Pratik Agile

Extreme Programming

Özcan Acar

PRATİK AGILE
EXTREME PROGRAMMING

Pratik Agile

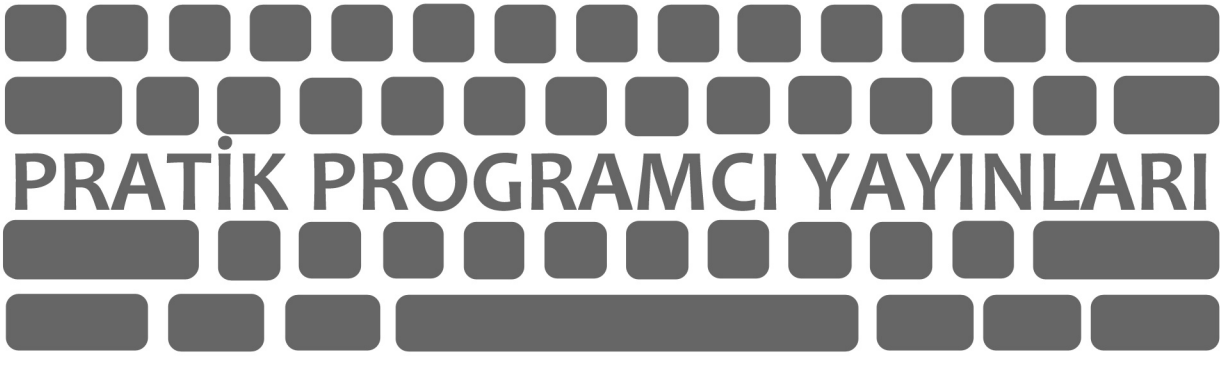
Yazılımcılar için Extreme Programming ile ve yazılım geliştirme rehberi

Özcan Acar

PRATİK PROGRAMCI YAYINLARI

Pratik Programcı 2

Yazar:	Özcan Acar
Yayınevi:	Pratik Programcı Yayınları
İlk sürüm:	Nisan 2014
Kapak tasarımı:	Ahmet Yıldırım
Düzelti:	Davut Ozanoğlu
Satış:	http://www.pratikprogramci.com



pratikprogramci.com

Beni gözü gibi seven dedem Cemal Köse'nin anısına ...

Bilgi Paylaşım İle Çoğalır

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

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

Bölüm Başlıkları

Kitap 18 bölümden oluşmaktadır. Ana bölüm başlıkları şunlardır:

- 1.Bölüm - Çevik Süreç Extreme Programming
- 2.Bölüm - Proje Planlama
- 3.Bölüm - İletişim
- 4.Bölüm - XP Çalışma Ortamı
- 5.Bölüm - XP Projesi
- 6.Bölüm - Çalışma Ortamı Kurulumu
- 7.Bölüm - Çevik Tasarım
- 8.Bölüm - Birim Testleri
- 9.Bölüm - Test Güdümlü Yazılım
- 10.Bölüm - XP ile Shop Sistemi İmplementasyonu
- 11.Bölüm - Onay/Kabul Testleri
- 12.Bölüm - Spring Çatısı
- 13.Bölüm - Spring MVC
- 14.Bölüm - Sürekli Entegrasyon
- 15.Bölüm - Yazılım Metrikleri
- 16.Bölüm - Subversion ile Versiyon Kontrolü
- 17.Bölüm - Proje Takibi
- 18.Bölüm - XP Hakkında Sorular ve Cevapları

İçindekiler

Yazar Hakkında

1974 İzmir doğumluyum. İlk ve orta öğrenimimi İzmir'de tamamladıktan sonra Almanya'da bulunan ailemin yanına gittim. Doksanlı yılların sonunda Almanya'nın Darmstadt şehrinde bulunan FH Darmstadt üniversiteden bilgisayar mühendisi olarak mezun oldum. 2014 yılında Pratik Spring isimli kitabım Pratik Programcı Yayınları tarafından yayımlanmıştır.

KurumsalJava.com ve Mikrodevre.com adresleri altında blog yazıyorum. Kurduğum BTSoru.com'da bana yazılımla ilgili sorularınızı yöneltebilirsiniz.

Önsöz

Danışman ve programcı olarak birçok projede çalışma imkanı buldum. 2003 öncesi projelerine daha çok geleneksel şelale (waterfall) yazılım metotları hakimdi. Bu tarz projelerde genellikle proje öncesinde müşteri gereksinimleri en son detayına kadar kağıda dökülür. Bu yüzlerce sayfadan oluşan bir spesifikasyonun oluşturulması anlamına gelmektedir. Çoğu zaman bu ön çalışma altı ay ya da bir sene sürmektedir. Programcı olarak görevim diğer ekip arkadaşlarımla bu spesifikasyonda yer alan gereksinimleri implemente etmekti. Çoğu zaman akıllarda oluşan soru işaretlerine bu spefisikasyon cevap verebilecek durumda değildi, çünkü analizcinin aklına gelmeyen bazı detaylar bu spesifikasyonda yoktu.

Herhangi bir şekilde sahip olduğumuz sorular cevaplarını bulurdu ve implementasyonu doğru olduğunu düşündüğümüz şekilde devam ettirirdik. Birçok projede müşteriyi bir kez bile gördüğümü hatırlamam. Sanki proje ekibiyle müşteri arasında görünmeyen bir duvar vardı. Ne bizim ona oluşmamız mümkündü, ne de müşterinin programcılarla temas etmesi isteniyordu. Sebebi belliydi: programcıların zaman kaybetmeden ve başka görevler üstlenmek zorunda kalmadan spesifikasyonu implemente etmesi gerekiyordu. Hiç kimse programcıları rahatsız edememeliydi.

Geleneksel tarzda yapılan proje planlarında tespit edilen görevler mevcut kaynaklara dağıtılır. Örneğin ben programcı olarak haftalık 5 iş gününü temsil eden bir kaynağım. Bu durumda proje menajeri oluşturduğu proje planında önündeki üç ya da altı ay için bana uygun gördüğü görevleri atar. Her görevin bir başlangıç ve bitiş tarihi vardır ve bana düşen bu zaman diliminde bana atanan görev yerine getirmektir. Görevi bu zaman diliminde

tamamlayamazsam yandı gülüm keten helva :-). Birçok projede planlama ve tahminler yanlış olduğu için hafta sonları çalışmak zorunda kaldığımı iyi hatırlarım. Hafta içinde normal mesai saatlerinden fazla çalışmak yetmiyormuş gibi, arka arkaya hafta sonları çalışmak zorunda bırakılmak bir çalışanı hem çok yorar hem de çalışma verimini azaltır. Tabi programcı olarak sizin göreviniz her zaman çalışır durumda olmaktır. Siz insan mısınız, robot mu bu yöneticileri pek ilgilendirmeyen bir sorudur. Kaynak görevini yerine getirmek zorundadır! O kadar!

En büyük sorunları projelerin sonlarına doğru yaşadım. Birçok programcı haftalar ya da aylarca çalışma sonunda oluşturdukları modülleri proje sonunda bir araya getirerek, entegre çalışan bir sistem oluşturmaya çalışırlar. İlk entegrasyonda denemesinde çalışan bir sistem hiç görmedim desem yalan olmaz! Çoğu zaman hiçbir iletişime ihtiyaç duyulmadan oluşturulmuş modüllerin beraber çalışmaları bir mucizedir. En büyük tartışmalar da bu yüzden proje sonlarında yaşanır. Suçlu birileri aranır ve mutlaka bulunur! İnsanın bazen içinden işe gitmek bile gelmiyor!

Bu durum 2003 den sonra katıldığım projelerde kısmen değişmeye başladı. 2000 yılında oluşan ve popüler olan çevik süreçler Avrupa'da yaygınlaşmaya başladı. Çevik süreçlerin uygulandığı projelerde daha müşteri odaklı çalışma fırsatı buldum. Müşteri gereksinimlerini tespit edebilmek için müşteri ile devamlı bir diyalog içinde olmanız gerekiyor. Çevik süreçler müşteriyi proje sürecine dahil eder. Müşteri programcıların sorularına devamlı cevap verecek yakınlıktadır. Ayrıca çevik projelerde proje öncesi çok detaylı spesifikasyonlar oluşturulmaz. Müşteri tarafından dile getirilen özellikler kullanıcı hikayesi (user story) olarak hikaye kartlarına (story card) yazılır. Müşteri bu kullanıcı hikayelerine öncelik sırası vererek, hangi gereksinimlerinin öncelikli olarak implemente edilmesi gerektiğini belirler. Kullanıcı hikayelerinin implementasyon süresi programcılar tarafından tahmin edilir. Bu müşteriye hangi isteğinin hangi zaman diliminde gerçekleştirilebileceğini anlamasında yol gösterici olur. Bu veriler doğrultusunda gerçeği büyük oranda yansıtan sürüm planları (release planning) oluşturulur. Belli zaman aralıklarında müşteriye çalışır bir sistem sunulur. Bunun için programcıların ürettikleri kodu sürekli entegre etmeleri gerekir. Bu sayede müşteriden geribildirim sağlanarak, ne oranda müşteri isteklerinin karşılandığı anlaşılır. Bu şekilde işleyen projelerde çalışmak büyük bir zevktir. Kesinlikle fazla mesai yapılmasına izin verilmez ve hafta sonları çalışılması talep edilmez. Programcılara kaynak değil, insan gözüyle bakılır ve sorumluluk alarak takımın bir parçası olmaları sağlanır.

Çevik yöntemlerin kullanımı gerçekten bir programcı olarak benim hayatımı ve yazılım sürecine olan negatif bakış tarzımı değiştirdi. Projelerde edindiğim tecrübeleri bu kitapta sizinle paylaşmak istedim. Çevik süreçler, özellikle Extreme Programming tamamen ekip işine ve iletişime dayanan çalışma yöntemlerine sahiptir. Bunun yanı sıra müşteri sürecin merkezinde görülür. Bu müşterinin istekleri doğrultusunda yazılım yapılmasını kolaylaştırır. Oluşan yazılım sistemi %100 müşterinin istediği özelliklere sahiptir ve değişikliklere ve yeniliklere açıktır.

Kitabın İçerigi Nedir?

Kitabın ana konusu çevik süreç olan Extreme Programming in uygulanış tarzını tanıtmaktır. Kitabın ilk bölümlerinde Extreme Programming hakkında teorik bilgiler yer almaktadır. Extreme Programming yöntemlerini uygulayabilmek için bu temel teorik bilgilerin alınmasında fayda vardır. Kitap 18 bölümden oluşmaktadır. Bu bölümlerin içerikleri özetle şöyledir:

Bölüm 1

Birinci bölümde şelale tarzı yazılım yöntemleri ve projelerde oluşan sorunlar yer almaktadır. Bu sorunları gidermek için çevik süreçlerin kullanımı tavsiye edilmektedir. Extreme Programming (XP) bir çevik süreç olarak projelerde oluşan sorunlara cevap verebilecek yazılım metotlarına sahiptir. Bu bölümde Extreme Programming in sahip olduğu değerler, prensipler ve teknikler tanıtılmaktadır.

Bölüm 2

XP projelerinde sürüm ve iterasyon planları oluşturularak, projenin gidişatı belirlenir. Planlama oyunu olarak isimlendirilen süreçte müşteri tarafından implemente edilecek özellikler seçilir. Programcılar gerekli tahminleri yaparak, müşterinin implementasyon için gerekli süre hakkında fikir sahibi olmasını sağlarlar. Bu bölümde sürüm ve iterasyon planlarının nasıl oluşturulduğu tematize edilmektedir.

Bölüm 3

XP iletişime dayalı bir süreçtir. Ekip çalışanları arasında iletişimi güçlendirmek için günlük Stand-up toplantılar düzenlenir. Bunun yanı sıra belirli aralıklarla retrospective (geriye bakış) toplantılarında projeye geri bakış sağlanarak, oluşan

hatalar üzerinde tartışılır ve çözümler aranır.

XP projelerinde programcılar pair programming (eşli programlama) metoduyla eşli çalışırlar. Bu programcıların kısa sürede teknik alanda aynı seviyeye gelmelerini kolaylaştırır. XP projelerinde çalışma ortamının iletişim aspekti göz önünde bulundurularak oluşturulması gerekmektedir. Üçüncü bölümde Stand-up ve retrospective toplantıları yanı sıra, pair programming ve çalışma ortamının oluşturulması konuları tematize edilmektedir. Ayrıca proje hakkında bilgilerin paylaşıldığı Wiki sistemleri hakkında bilgi verilir. Trac ve Bugzilla gibi araçlar kullanılarak bilgi ve hata yönetimi sağlanır.

Bölüm 4

Bu bölümde XP projelerinde çalışma ortamlarının nasıl yapılandırıldığı ve ne tür araçlardan faydalanıldığı konusu incelenmektedir.

Bölüm 5

Teorik bilgilerin ardından, XP nin nasıl uygulandığını göstermek amacıyla beşinci bölümde örnek bir XP projesi yer almaktadır. Bu bölümde müşteri gereksinimlerinin nasıl tespit edildiği ve sürüm ve iterasyon planlarının nasıl oluşturulduğu bir örnek üzerinde gösterilmektedir.

Bölüm 6

Proje öncesinde sıfırıncı iterasyonda programcılar ihtiyaç duydukları teknik ortamı oluşturmaya başlarlar. Altıncı bölümde Eclipse, Ant, Tomcat, Subclipse, JUnit, HSQL, DBUnit gibi araçların kullanımı ve kurulumu incelenmektedir.

Bölüm 7

Yazılım sürecinde uygulanması gereken tasarım prensipleri bu bölümün ana konusudur. Esnek bağ, açık kapalı prensibi, tek sorumluluk prensibi, Liskov yerine geçme prensibi, bağımlılıkların tersine çevrilmesi prensibi, arayüz ayırma prensibi ve paket tasarım prensipleri detaylı ve örnekli olarak bu bölümde incelenmektedir.

Bölüm 8

Birim (unit) testleri oluşturularak sistem komponentleri yazılım esnasında test edilir. Onay/kabul (acceptance) testleri, entegrasyon testleri, regresyon testleri, performans testleri gibi değişik türde testler oluşturmak mümkündür.

Java projelerinde JUnit test çatısı kullanılarak testler oluşturulur. Sekizinci bölümde JUnit kullanılarak birim testlerinin nasıl oluşturulduğu uygulamalı olarak gösterilmektedir.

Bölüm 9

XP projelerinde yazılım test güdümlü (Test Driven Development – TDD) yapılır. Dokuzuncu bölümde TDD sürecinin nasıl çalıştığı ve programcıların TDD yöntemleriyle implementasyonu nasıl ilerlettikleri yer almaktadır.

Bölüm 10

Onuncu bölümde pratik uygulamalı olarak shop sisteminin login modülü implemente edilmektedir. İmplementasyon için onay/kabul testlerinden yola çıkılarak, veri tabanına kadar uzanan yapının adım adım TDD kullanılarak nasıl implemente edildiği gösterilmektedir. Bu bölümde sistem komponentlerinin simülasyonu için mock sınıfların kullanımı yer almaktadır.

Bölüm 11

İmplementasyonun çalışır durumda olduğunu kanıtlamak için onay/kabul testleri oluşturulur. Bu testler kullanıcı hikayeleri gibi müşteri tarafından tanımlanır ve programcılar tarafından implemente edilir. On birinci bölümde onay/kabul testlerinin hangi teknik ve araçlar kullanılarak implemente edildiği konusu incelenmektedir.

Bölüm 12

On ikinci bölümün konusu Spring dir. Spring çatısı ile tasarımı ve teknik altyapısı güçlü, bakımı kolay ve kolay genişletilebilir programlar oluşturmak mümkündür. Spring sunduğu altyapı hizmetleriyle programcıların hayatını kolaylaştırır ve programın test edilebilirliğini yükseltir.

Bölüm 13

Web uygulamalarının geliştirilmesi için Spring MVC çatısı kullanılabilir. Bu bölümde Spring MVC nin nasıl kullanıldığı incelenmektedir.

Bölüm 14

Sürekli entegrasyon (continous integration) XP projelerinde uygulanan bir tekniktir. Programcılar tarafından kod üzerinde değişiklik yapıldığı anda mevcut kod derlenerek, sistem testleri çalıştırılır. Bu işlem sonunda kod

üzerinde kırılmalar oluşmuşsa, programcıların konu hakkında e-posta aracılığıyla dikkati çekilir ve hatanın bir an önce giderilmesi talep edilir. Sürekli entegrasyon ile her zaman çalışır bir sistemin olması sağlanır. On dürdüncü bölümde sürekli entegrasyon konusu incelenmektedir.

Bölüm 15

İmplemenatasyonun hangi yöne gittiğini tespit edebilmek için yazılım metrikleri kullanılır. Bunlar sistemin belirli özelliklerinin ölçülmesi sonucu ortaya çıkan değerlerdir. On beşinci bölümde sistem metriklerinin nasıl ve hangi araçlar kullanılarak ölçüldüğü gösterilmektedir.

Bölüm 16

Kod paylaşımını kolaylaştırmak için versiyon kontrol sistemleri kullanılır. On altıncı bölümde açık kaynaklı olan Subversion versiyon kontrol sisteminin kullanımı açıklanmaktadır.

Bölüm 17

Projenin gidişatını kontrol edebilmek için proje takip planları oluşturulur. Bu planlarda Burndown grafikleri kullanılarak görsellik sağlanır. Enformasyon radyatörü olarak tanımlanan metot ve yöntemler kullanılarak proje ekibinin ve diğer şahısların projenin gidişatı hakkında bilgi sahibi olması sağlanır. On yedinci bölümde sürüm ve iterasyon takibi konuları incelenmektedir.

Bölüm 18

Son bölümde XP hakkında soru ve cevapları yer almaktadır.

Kitabın İçeriği Ne Degildir?

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

Kitapta anlatımı kolaylaştırmak için UML diyagramları kullanılmıştır. Okuyucunun temel UML bilgisine sahip olması, verilen örneklerin anlaşımını kolaylaştıracaktır. Kitabın amacı UML i tanıtmak ya da öğretmek değildir. UML

i öğrenmek için diğer kaynaklar tavsiye edilmektedir.

Kitap Kim İçin Yazıldı?

Bu kitap Extreme Programming hakkında bilgi sahibi olmak isteyen proje menajerleri, programcılar ve konuyla ilgilenen diğer şahıslar için yazılmıştır. Kitabın ilk bölümlerinde yer alan teorik bilgiler, konu hakkında yeterli bilginin edinilmesini sağlayacak niteliktedir. Kitabın büyük bir bölümü programcılara hitap edebilmek amacıyla pratik uygulamalı tarzda şekillendirilmiştir.

Kitap Nasıl Okunmalı?

Kitabın ilk dört bölümü Extreme Programmıng in temellerini göstermektedir. Bu sebepten dolayı bu ilk dört bölümün mutlaka tayin edilen sıraya göre okunmasında fayda vardır. Diğer bölümler okuyucunun isteği doğrultusunda okunabilir.

Yazar İle İletişim

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

PratikProgramci.com

PratikProgramci.com kaleme aldığım ve bundan sonra almayı planladığım kitapları dijital formatta sizlerle buluşturmak istediğim yeni bir eğitim platformu. Kitaplarım yanı sıra belli, başlı yazılım konularını kapsayan görsel öğrenim (screencast) modülleri oluşturarak, beğeninize sunmak istiyorum. Bunun ilk örneğini Profesyonel Java oluşturuyor. Gelişmeleri pratikprogramci.com adresinden takip edebilirsiniz.

Kitapta Yer Alan Kod Örnekleri

Kitapta kullanılan kod örneklerini Eclipse projesi halinde http://www.pratikprogramci.com/?wpdmact=process&did=MTYuaG90bGluaw== adresinden edinebilirsiniz.

1. Bölüm

Çevik Süreç Extreme Programming

Giriş

Yazılım sektörü yıllardan beri kan kaybediyor. Ama artık taze kan bulundu ve hastalığın tedavisi kolaylaştı. Çözüm çevik süreçler!

Günümüze kadar uzanan süreçte yazılım sektöründe yapılan projeler nereye varacağı belli bile olmayan büyük maceralar haline gelmiştir. Bunun başlıca sebebi kullanılan yazılım yöntemlerinin gereksinimlere cevap verecek yapıda olmamasıdır. Çevik süreçler bu sorunu çözecek nitelikte. Bu bölümde

- yazılım yaparken hedefin ne olduğunu,
- çevikliğin ve çevik sürecin ne olduğunu,
- çevik manifesto ve prensiplerinin ne anlama geldiğini,
- çevik sürecin diğer yazılım metotlarına kıyasla hangi farklılıkları beraberinde getirdiğini,
- hangi çevik süreç türlerinin mevcut olduğunu,
- bir çevik süreç olan Extreme Programming in ne olduğunu,
- Extreme Programming in hangi değer, prensip ve teknikler üzerine kurulu olduğunu

yakından inceleyeceğiz.

Hedef

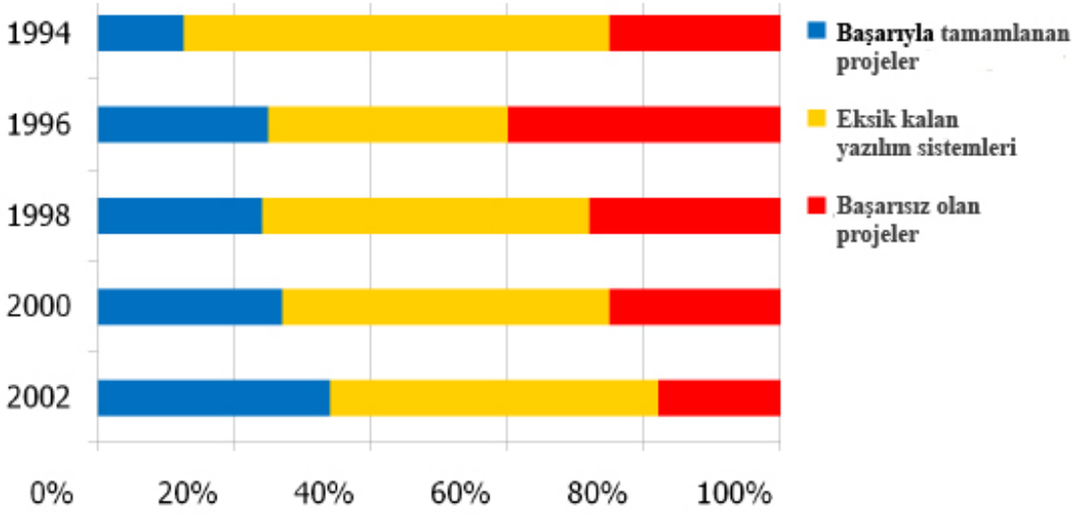
Ana hedef, müşteri gereksinimlerini tatmin eden, sabit bir bütçe ve belirli bir zaman diliminde, hatalardan arındırılmış ve müşterinin piyasadaki rekabet etme yeteneğini kuvvetlendirecek bir yazılım sistemi geliştirmektir. Yazılım ve proje geliştirme süreçleri bu hedefe ulaşmak için kullanılan metotları ihtiva eder.

Hedefin net bir şekilde tarifinin yapılabilmesine rağmen, günümüzde uygulanan birçok projenin başarıyla tamamlanamadığını ya da oluşturulan yazılım sistemlerinin müşteri gereksinimlerini tatmin etmediğini görmekteyiz. Bunun sebepleri:

- Geleneksel yazılım metotları proje başlangıcında müşterinin tüm gereksinimlerini tespit eder veya tespit ettiğini düşünür ve akabinde implemente eder. Yazılım süreci zaman içinde değişikliğe uğrayan müşteri gereksinimlerine ayak uyduracak şekilde organize edilmemiştir. Müşteri tarafından istenen değişiklikler hoş karşılanmaz. Bu sebepten dolayı

geliştirilen yazılım sisteminin müşteri gereksinimlerini %100 tatmin etme şansı yok gibidir.

- Müşteri istekleri doğrultusunda yapılmak zorunda kalınan değişiklikler proje maliyetlerini yükseltir, çünkü bu beraberinde basit olmayan yapısal değişiklikler getirebilir.
- Proje yöneticilerinin programcılardan insan üstü beklentileri, projenin büyük bir bölümünün sorumluluk ve riskini omuzlarında taşıyan bu bireylerin fazla mesai yapmalarına ve ruhen ve bedenen yorulmalarına sebep vermektedir. Motivasyonu düşük olan programcıların yaratıcı yönleri azalmakta ve kodun kalitesi buna orantılı olarak düşmektedir. Bu programlardaki hata oranının artması anlamına gelmektedir.
- Projelerde takım çalışması ve bireyler arası iletişimin kuvvetlendirilmesi desteklenmez. Her programcı yazılımını yaptığı program parçasından sorumlu olduğu için projeden ayrılan programcılar proje için risk haline dönüşür.
- Proje başlangıcında müşteri gereksinimleri en son detayına kadar tespit edilir. Ayrıca yazılım sistemi için teknik mimari ve uygulanacak tasarım belirlenir. Bunlar proje başlangıcında çok zaman kaybedilmesine neden olur. Ayrıca bir zaman sonra tasarımın implemente edilen gereksinimlere cevap vermez hale geldiği ve yapılan her değişiklik sonucunda tasarım çıkmaza girdiği görülür.
- Oluşturulan ve uygulanan proje planı bir zaman sonra geçerliliğini yitirir. Plana mutlaka uyulması gerektiği düşüncesi, plan dışına çıkılmasını önlemek için fazla mesai yapılmasını zorunlu kılar.



Resim 1.1 Standish Group tarafından yapılan Chaos Study istatikstikleri

Bu ve bunun gibi sıralayabileceğimiz bir çok nedenden dolayı projelerin büyük bir bölümü başarısızlığa mahkum olmaktadır. Bahsedilen sorunları ortadan kaldırmak mümkün olabilmelidir. Bunun cevabını çevik süreçler verir.

Çevik Süreç

Türk Dil Kurumu'nun online sözlüğünde çevik kelimesi şu şekilde tanımlanmaktadır:

```
Kolaylık ve çabuklukla davranan, tetik, atik
```

Çevik süreçler yazılım sektöründe kullanılan mevcut geleneksel yöntemlere (örneğin Rational Unified Process – RUP ya da V-Model) alternatif olarak geliştirilmiş modern ve bürokrasiye mesafeli yazılım yöntemlerini ihtiva ederler. Çevik yazılım (Agile Development) bir yandan bir değer sistemini, diğer yandan da somut yazılım metotlarını içerir. Çevik yazılıma yazılım sektöründe yeni bir filozofi akımı ya da yeni bir yazılım metamodeli olarak bakabiliriz. Bu yeni filozofinin somut örnekleri arasında Extreme Programming, Scrum ve Lean Development bulunmaktadır. Bu kitabın konusu Extreme Programming (XP) dir.

Çevik Sürecin Geçmişi

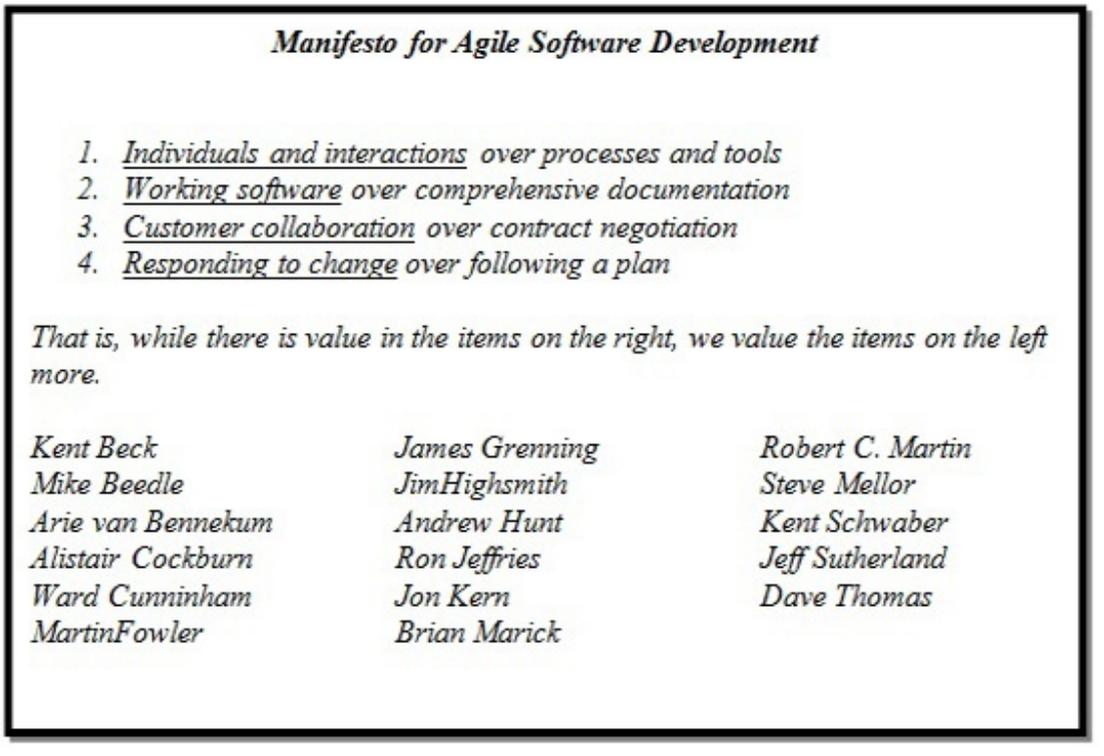
2000 senesinde Kent Beck ve arkadaşlarının yer aldığı bir toplantı düzenlendi. Kent Beck ve arkadaşları belli bir süredir çevik yazılım metotlarını projelerde uyguluyorlardı ve fikir alışverişi için böyle bir toplantının faydalı olacağını düşündüler. Smaltalk gibi nesneye yönelik modern programlama dilleri sayesinde iteratif yazılım metotları geliştirilmiş ve değişik ekipler tarafından uygulanmaktaydı. Smalktalk ile başlayan bu yeni akım doksanlı yılların ortalarına doğru iyice kuvvetlenmişti. Çevik (Agile) kelimesi ilk kez bu toplantıda değişik iteratif yazılım metotlarını bir çatı altında toplamak için kullanıldı.

Toplantıya katılanlar tarafından bir manifesto (Agile Manifesto) hazırlandı. Yeni bir yazılım filozofisi doğmuştu. Çevik sürecin mimarları olan katılımcılar kısa bir zaman sonra Agile Alliance organizasyonunu kurdular. Bu organizasyon ile çevik süreç ve çevik yazılımın desteklenmesi, geliştirilmesi ve uygulanması hedef alındı. Konuyla ilgilenen kurum, kuruluş ve şahıslar için bu

organizasyon merkezi bir buluşma noktasıdır. Agile Alliance organizasyonu ilgilenenlere çevik süreç ve çevik yazılım hakkında geniş bir literatür sunmakta ve her yıl bu konuda konferans düzenleyerek, katılımcıları bilgiledirmektedir.

Çevik Manifesto (Agile Manifesto)

2000 senesinde Kent Beck ve 16 arkadaşı tarafında aşağıda yer alan çevik manifesto oluşturulmuştur.



Manifesto for Agile Software Development

1. *Individuals and interactions over processes and tools*
2. *Working software over comprehensive documentation*
3. *Customer collaboration over contract negotiation*
4. *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck	*James Grenning*	*Robert C. Martin*
Mike Beedle	*JimHighsmith*	*Steve Mellor*
Arie van Bennekum	*Andrew Hunt*	*Kent Schwaber*
Alistair Cockburn	*Ron Jeffries*	*Jeff Sutherland*
Ward Cunninham	*Jon Kern*	*Dave Thomas*
MartinFowler	*Brian Marick*	

Türkçesi:

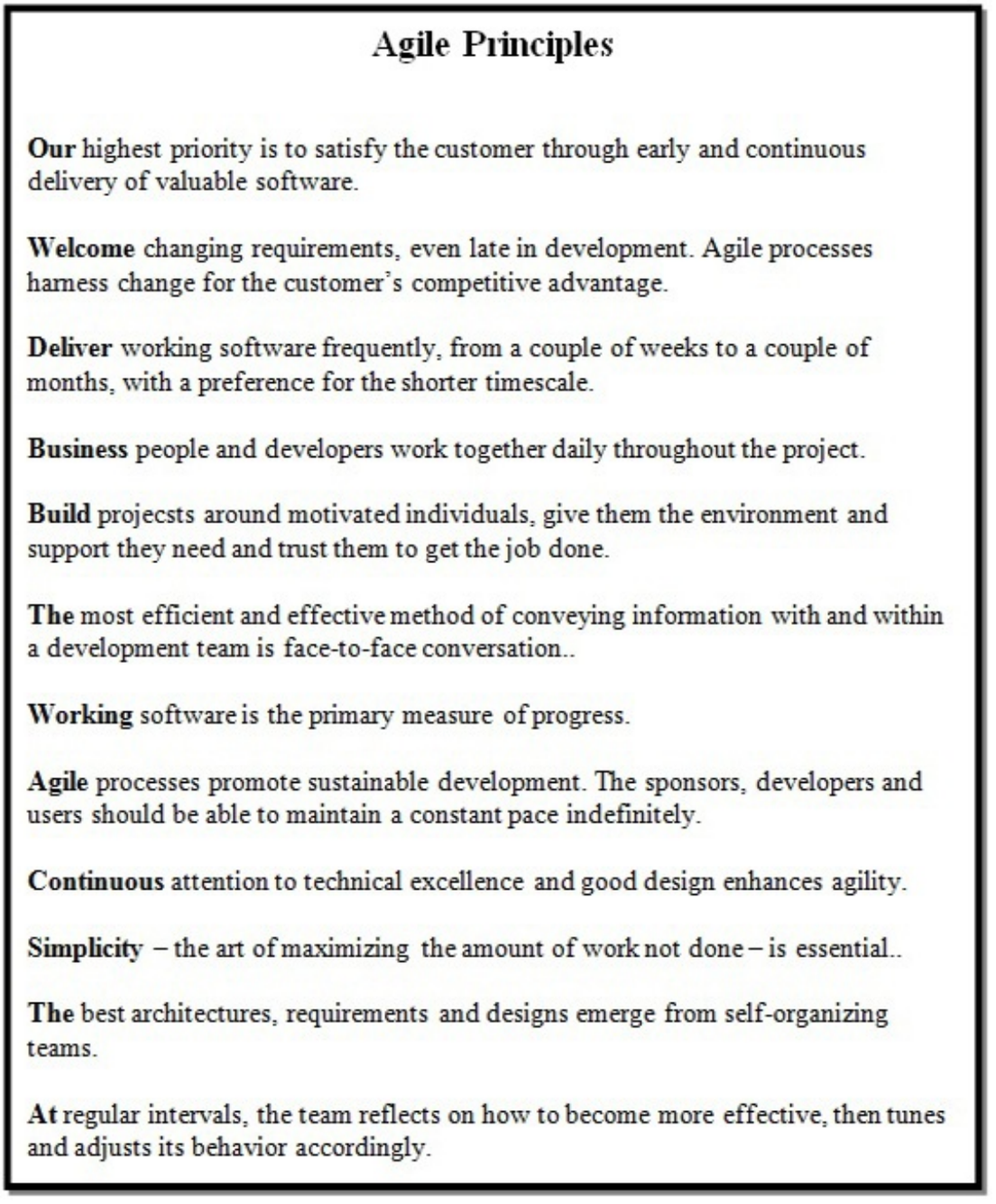
1. Kişiler ve iletişim süreç ve araçlardan önce gelir.
2. Çalışır durumda olan program detaylı dokümantasyondan daha önceliklidir.
3. Müşteri ile beraber çalışmak sözleşmelerden ve anlaşmalardan daha önceliklidir.
4. Değişikliklere ayak uydurmak bir planı takip etmekten daha önemlidir.

Sağ bölümde yer alanlar (altı çizili olmayanlar) değer taşımakla beraber, sol bölümde altı çizili olan değerler bizim için daha kıymetlidir.

Bu manifesto ile çevik sürecin bir değer sistemine sahip olması ve geleneksel yazılım metotlarından elde edilen tecrübeler doğrultusunda daha pratik ve çevik bir yapıda olması gerektiği dile getirilmiştir.

Çevik Prensipler (Agile Principles)

Çevik manifestoda yer alan ifadeler on iki prensip ile somut hale getirilmekte ve açıklanmaktadır. Bunlar:



Agile Principles

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.

Business people and developers work together daily throughout the project.

Build projecsts around motivated individuals, give them the environment and support they need and trust them to get the job done.

The most efficient and effective method of conveying information with and within a development team is face-to-face conversation..

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity – the art of maximizing the amount of work not done – is essential..

The best architectures, requirements and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

En önemli öncelik erken ve sürekli olarak kullanılabilir programlar oluşturarak, müşteriyi tatmin etmektir.

Bu prensip ile aslında programın müşteri tarafından talep edildiğini ve müşteriyi sadece istekleri doğrultusunda geliştirilmiş ve çalışır durumda olan bir programın tatmin edebileceği dile getirilmektedir. Peki müşteri nasıl tatmin edilebilir? Bunun için proje ekibinin proje başlangıcından itibaren ve sürekli olarak çalışır durumda olan programlar oluşturarak, müşteriye sunması ve görüşlerini alması gerekmektedir. Bu sayede müşteri inceleyebileceği ve çalışır durumda olan bir program aracılığıyla gereksinimlerinin ne oranda implementasyonla örtüştüğünü kontrol edebilir. Gereksinimlerin değişmesi ya da müşterinin istekleri dışında bir yazılım yapılması durumunda müşteri yazılım sürecine müdahale edebilir ve gerekli değişiklikleri talep edebilir. Bu durum proje sonunda müşteri gereksinimleri ile yüksek oranda örtüşen bir programın oluşmasını sağlar.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Yazılımın ilerleyen dönemlerinde gelse bile talep edilen değişiklikler hoş karşılanmalıdır. Çevik süreçler değişiklikleri müşterinin rekabetteki avantajını korumak ve sağlamak için kullanırlar.

Geleneksel yazılım metotlarının uygulandığı projelerde mimarinin ve planlamanın büyük bir bölümü implementasyon öncesi oluşturulur. Yazılım, hazırlanan planlardan sapmadan gerçekleştirilir. Bu müşteri tarafından talep edilen değişikliklerin göz ardı edilmesi anlamına gelmektedir. Böyle bir sürecin sonunda müşteri isteklerini tatmin etmeyen ve müşterinin piyasadaki rekabet kabiliyetini sınırlayan programlar oluşur. Bunu engellemek için her zaman müşteriden gelen değişiklik taleplerinin implementasyon esnasında dikkate alınması gerekmektedir. Değişiklik ne zaman gelirse gelsin, implementasyon bu değişiklik doğrultunda adapte edilebilmelidir.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.

Kısa sürelerde (birkaç haftadan, birkaç aya kadar sürebilen zaman dilimlerinde) çalışır programlar ortaya koy. Seçim zaman diliminin kısa tutulması yönünde olmalıdır.

Çevik süreçlerde programlar hem iteratif hem de inkrementel olarak geliştirilir.

Bir iterasyon yazılım için gerekli tüm safhaları ihtiva eden belirli bir zaman biriminde, örneğin 2 hafta implementasyonun gerçekleştirildiği süreçtir. Program ayrıca inkrementel oluşturulur, çünkü programcı ekip her iterasyonda müşterinin seçtiği gereksinimlere konsantre olarak programı yavaş yavaş oluşturur. Her iterasyon ardından program müşteriye sunularak, görüşleri alınır.

4. Business people and developers work together daily throughout the project.

Müşteri ve programcılar proje süresince beraber çalışırlar.

Çevik projelerde müşteri ile proje ekibinin beraber çalışması doğaldır. Programdan olan beklentileri en iyi müşteri bilebileceği için sürekli müşteriden geribildirim alınması gerekmektedir. Bunun en kolay yolu müşteri ile proje ekibinin beraber çalışmasıdır.

Proje ekibi, özellikle programcılar müşteri tarafından dile getirilen gereksinimlerin (requirement) fizibilitesini (yapılabilirlik) araştırırken, her gereksinim için implementasyon zamanını tahmin ederler. Bu doğrultuda müşteri kendi tanımlamış olduğu gereksinimlere bir öncelik sırası vererek, hangi gereksinimlerin öncelikli olarak implemente edilmesi gerektiğini tayin eder. Programcılar bu konularda müşteriye yardımcı olarak, gereksinimlerin daha somut hale getirilmelerini sağlarlar. Bunlar genellikle birkaç cümleden oluşan kullanıcı hikayelerine (user story) dönüştürülür. Sürekli müşteri ve programcılar arasındaki diyalog yanlış anlaşılmaları ortadan kaldırır. Bu yüzden müşterinin her gün programcıların erişebileceği bir mesafede olması gerekmektedir.

Geleneksel yazılım metotlarında bunun böyle olmadığını görmekteyiz. Adeta programcı ekip ve müşteri arasına görünmez bir duvar örülür. Projenin ilerleyen safhalarında müşteri tarafından talep edilen değişiklikler olumlu karşılanmaz, çünkü bu proje başlangıcında yapılan anlaşmalara ters düşmektedir. Başlangıçta ne yapılmasına karar verildiyse, programcılar rahatsız edilmeden mevcut gereksinimlerin implementasyonuna odaklanabilmelidirler. Tabii böyle bir sistem gölgesinde oluşan programın müşteri gereksinimlerini ne ölçüde tatmin edebileceğini düşünebilirsiniz.

5. Build projecsts around motivated individuals, give them the environment and support they need and trust them to get the job done.

Projelerin motivasyonu yüksek bireyler tarafından yapılmasını sağla, onlara ihtiyaç duydukları ortamı ve desteği ver ve işi bitirebileceklerine inan.

Çevik projeler bireylerden oluşur ve her programcı kendi bireysel karakteri ile takımın bir parçasıdır. Ekip elemanlarının değişik karakterde olmaları doğaldır. Programcıların onlara duyulan güvenin hissettirilmesiyle motivasyonları artırılır. Onların sorumluluk almaları sağlanarak, öz güvenlerinin pekişmesi sağlanır. Programcılar birbirlerine yardım ederler. Onlar arasında kıdem farkı yoktur. Bilen bilmeyene öğreterek, kısa bir zamanda aynı teknik seviyeye gelmeleri sağlanmış olur.

6. The most efficient and effective method of conveying information with and within a development team is face-to-face conversation.

Bilgi alışverişinde en verimli ve efektif yöntem takım içinde yüz yüze konuşmaktır.

Çevik projelerde bilgi alışverişi yüz yüze gerçekleşir, çünkü bilginin transfer esnasında en az hasara uğradığı yöntem budur. Programcılar arasındaki kişisel konuşmalar güveni artırır ve yanlış anlaşılmaları ortadan kaldırır. Sorunlar hemen açıklığa kavuşturulabilir.

7. Working software is the primary measure of progress.

Çalışır durumda olan program ilerlemenin ana göstergesidir.

Almanca'da kullanılan bir deyim vardır: "torbada kedi satın almam!". Torbanın içini görmeden satın aldığımızda, beklentimiz haricinde (beklentimiz örneğin bir horoz olabilir) bir şeyle (örneğin kedi – deyimde değersiz anlamında kullanılıyor) karşılaşabiliriz. Bu durum bir program siparişi veren müşteri içinde geçerlidir. Müşteriye kısa aralıklarla çalışır durumda olan lakin henüz tamamlanmamış bir program sunulduğunda, müşteri, mevcut programın kendi gereksinimlerini tatmin edecek durumda olup, olmadığını kontrol edebilir. Program yalan söylemez. Bu yüzden müşteri çalışır durumda olan programı kullandıktan sonra torbanın içine bakmış ve torbanın içindeki şeyin kedi mi yoksa horoz mu olduğunu görebilmiştir.

8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.

Çevik süreçler etkili yazılım yöntemlerini destekler. Müşteri, programcılar ve kullanıcılar sabit bir tempoda beraber çalışabilmelidirler.

Çevik projelerde sabit bir çalışma temposunun oluşturulması büyük önem taşımaktadır. Programcılar arasında görev eşit bir şekilde paylaştırılır. Fazla mesai yapılması hoş karşılanmaz. Bu ayrıca çalışanların motivasyonu olumlu etkiler. Proje ilerledikçe iş azalır. Sonradan programcıları kötü sürprizler beklemez, çünkü ortada iyi test edilmiş ve çalışır durumda olan bir program vardır.

9. Continuous attention to technical excellence and good design enhances agility.

Devamlı teknik mükemmelliğe özen gösterilmesi ve iyi tasarım çevikliği kuvvetlendirir.

Çevik projelerde kalite beklentisi yüksektir. Yazılım temin edilen en iyi araç ve yetenekli programcılarla gerçekleştirilir. Bu süreçte programın tasarımı sürekli optimize edilir. Programcılar yeniden yapılandırma (refactoring) esnasında buldukları tasarım hatalarını hemen giderirler.

10. Simplicity – the art of maximizing the amount of work not done – is essential.

Sadelik (basitlik) esastır.

Mümkün olan en basit tarzda implementasyonu gerçekleştirmek, programın karmaşıklığını azaltır. Karmaşıklık oranı düşük olan bir uygulamanın bakımı ve geliştirilmesi kolaydır. Programcılar yazılım esnasında sadece kendilerinden o anda olan beklentiyi tatmin edecek kadar kod yazarlar. Bu gereksiz kodun oluşmasını engeller ve test edilebilir bir yapının ortaya çıkmasını sağlar.

11. The best architectures, requirements and designs emerge from self-organizing teams.

En iyi mimariler, gereksinimler ve tasarımlar kendi kendine organize olabilen takımlardan çıkar.

Çevik takımlar kendi başlarına organize olabilme özelliğine sahiptir. Böyle takımlar görevleri kendi aralarında eşitçe paylaşırlar. Takımlar ve bireyler arasındaki görüşmeler ve bilgi alışverişi sonucunda mimari ve tasarım gözden geçirilir ve optimize edilir. Ayrıca bireyler arası yapılan konuşmalar müşteri

tarafından dile getirilen gereksinimlerin açıklığa kavuşturulmalarını ve daha iyi anlaşılmalarını sağlarlar.

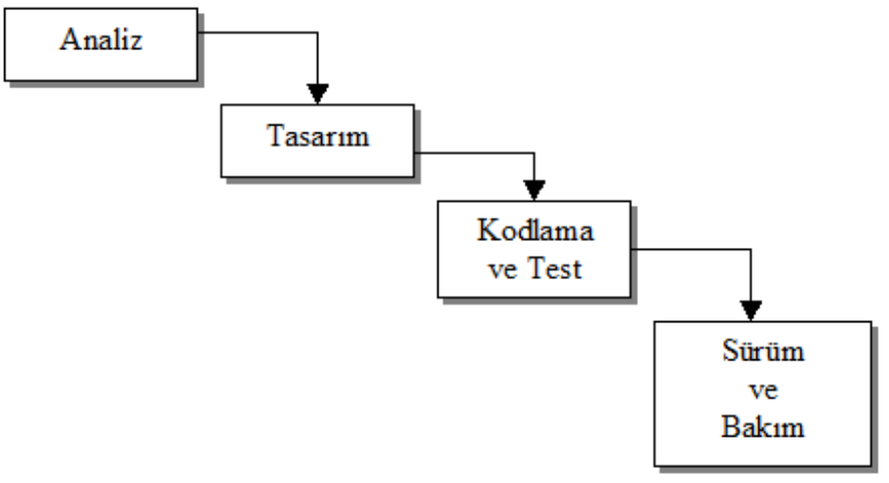
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Belirli zaman dilimlerinde takım daha nasıl verimli olabileceği konusunda kendini sorgular ve edindiği bilgiler doğrultusunda çalışma tarzını adapte eder.

Çevik takımlarda bilgi alışverişi ve devamlı öğrenme çok önemlidir. Belirli zaman aralıklarıyla takım çalışanları bir araya gelerek, fikir alışverişinde bulunurlar ve çalışma yöntemlerini sorgularlar. Edinilen tecrübeler doğrultusunda yazılım sürecinde adaptasyonlar gerekebilir. Nihai amaç en verimli çalışabilmek ve müşteri tarafından kabul görmüş bir program oluşturabilmektir.

Çevik Sürecin Farkı

Yazılım geliştirme süreci analiz, tasarım, kodlama, test, sürüm ve bakım gibi safhalardan oluşur. Geleneksel yazılım metotlarında bu safhalar şelale modelinde olduğu gibi doğrusal (linear) olarak işler. Her safha başlangıç noktasında bir önceki safhanın ürettiklerini bulur. Kendi bünyesindeki değişikler doğrultusunda teslim aldıklarını bir sonraki safhanın kullanabileceği şekilde dönüştürür.



Resim 1.2 Şelale modeli

Şelale modelinin özelliklerini şu şekilde sıralayabiliriz:

1. Şelalenin her basamağında yer alan aktiviteler eksiksiz olarak yerine

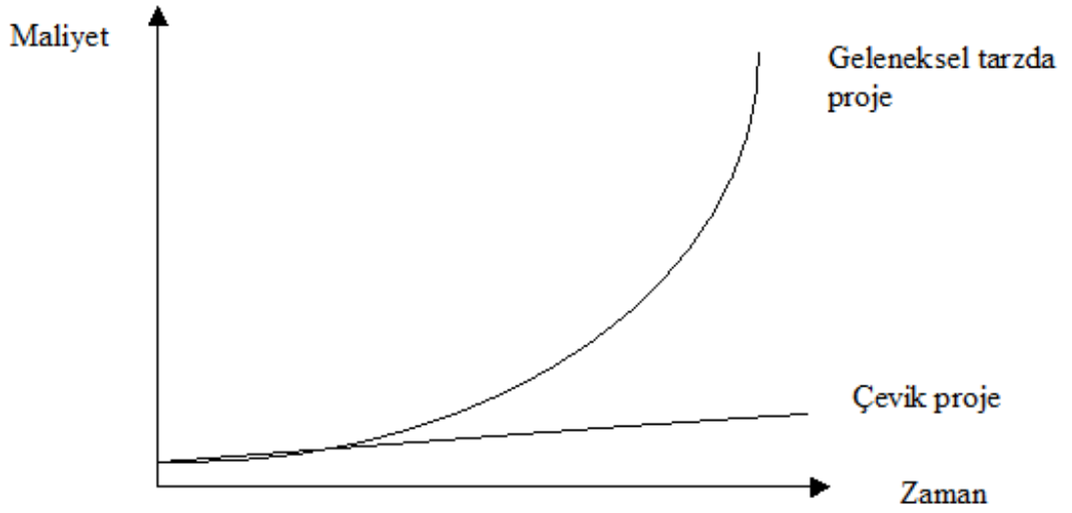
getirilir. Bu bir sonraki basamağa geçmenin şartıdır.
2. Her safhanın sonunda bir doküman oluşturulur. Bu yüzden şelale modeli doküman güdümlüdür.
3. Yazılım süreci doğrusaldır, yani bir sonraki safhaya geçebilmek için bir önceki safhada yer alan aktivitelerin tamamlanmış olması gerekir.
4. Kullanıcı katılımı başlangıç safhasında mümkündür. Kullanıcı gereksinimleri bu safhada tespit edilir ve detaylandırılır. Daha sonra gelen tasarım ve kodlama safhalarında müşteri ve kullanıcılar ile diyaloğa girilmez.

Bu modelin beraberinde getirdiği problemleri şu şekilde sıralayabiliriz:

1. Safhaların birbirinden kesin olarak ayrı tutulmaları gerçekçi değildir. Projelerde safhalar arasındaki bu sınırlar yok olabilir.
2. Teoride safhalar birbirlerini takip edeler. Projelerde bunun bazen mümkün olmadığını ve önceki safhalara geri dönülmek zorunda kalındığını görebiliriz.
3. Safhalar arası geri bildirim yetersizdir. Model değişikliğe açık değildir.
4. Müşteri gereksinimlerinin proje öncesi detaylı olarak kağıt üzerinde oluşturulması ilerde sorun yaratabilir. Müşteri gereksinimleri değişikliğe uğrayabileceği için yazılım sisteminin de yapısal değişikliğe uğraması kaçınılmaz olabilir. Böyle bir durum maliyeti artırır, çünkü yeni ve değişen gereksinimleri implemente edebilmek için modelde yer alan safhaların birkaç kere uygulanması gerekebilir.
5. Sistemin kullanılır hale gelmesi uzun zaman alabilir.
6. Başlangıçta yapılan hataların tespiti çok uzun zaman alabilir. Bu hataların giderilmesi maliyeti yükseltir.
7. Modül implementasyonları için zaman tahminleri proje planlarını oluşturan yöneticiler tarafından yapılır. Teknik bilgiye sahip olmayan şahıslar tarafından yapılan bu tahminler çoğu zaman doğru değildir. Bu durum proje planlama sürecini negatif etkiler.

Proje başlangıcında her detayı göz önünde bulundurmak mümkün olmadığı için şelale modeliyle geliştirilen yazılım sistemlerinin müşteri gereksinimlerini tam tatmin etmediğini görmekteyiz. Bunun önüne geçebilmek için projenin başlangıç safhasında analiz için çok zaman harcanır ve müşteri gereksinimleri en ince detayına kadar tespit edilir. Aslında proje başlangıcında oluşturulan dokümanlar geçerliliklerini yitirmiştirler, çünkü müşteri gereksinimleri piyasa ve rekabet koşulları gereği değişikliğe uğramış olabilir. Ne yazık ki şelale modeli bunları dikkate almaz ve müşterinin talep ettiği değişiklikleri aza indirmeye

çalışır. Bunun bir sebebi de sonradan gelen değişiklik taleplerinin maliyeti yükseltmesidir, çünkü bu durumda şelale modelinde yer alan safhaların birkaç kere uygulanması gerekebilir.

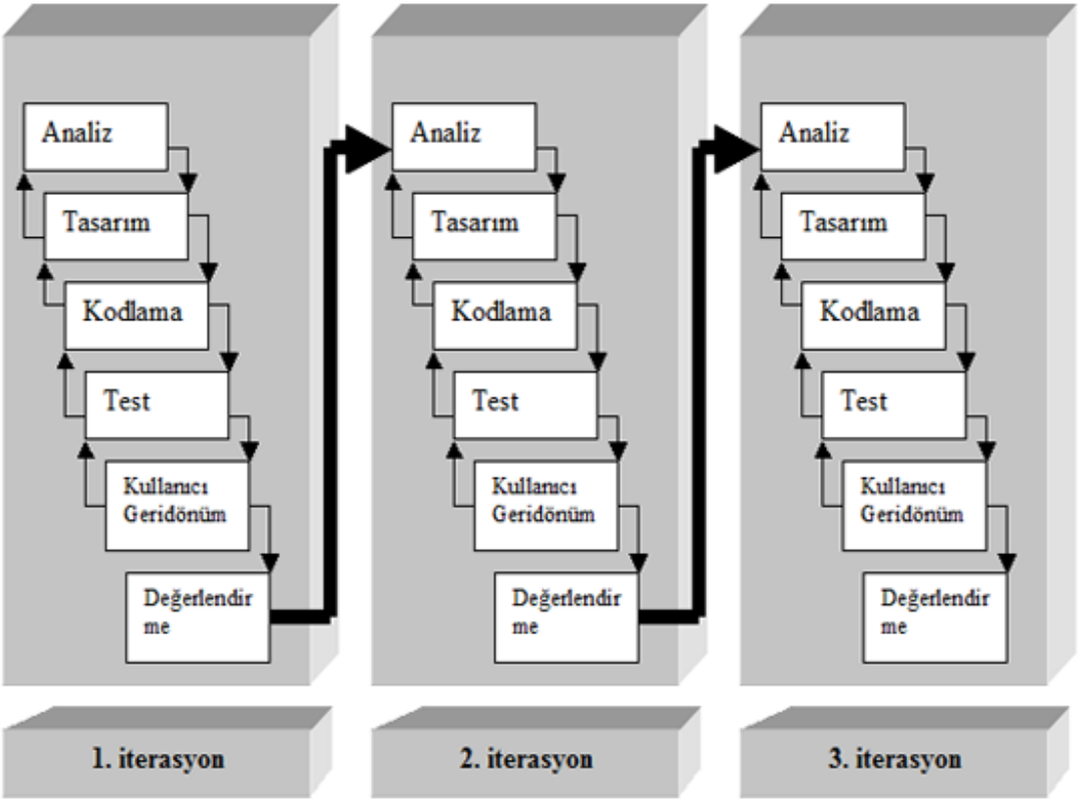


Resim 1.3 Geleneksel tarzda yapılan projelerde gerekli değişikliklerin yapılması maliyete zamanla yükseltir.

Bu çerçeveden bakıldığında proje sonunda oluşan program müşterinin güncel gereksinimlerini tatmin etmez durumdadır. Program daha çok müşterinin proje başlangıcında sahip olduğu gereksinimleri tatmin edecek şekilde tasarlanmıştır. Projelerin birkaç sene boyunca sürebileceğini düşünürsek, aslında bu süreç sonunda oluşan program güncel değildir.

Çevik süreçlerde durum farklıdır. Çevik süreç değişimi kabul eder ve onunla yaşamayı kolaylaştırmak için yeni yazılım metotları sunar.

Çevik süreçlerde iterasyon bazında çalışmalar sürdürülür. Her iterasyon bir ile dört haftalık zaman dilimlerinden oluşur ve şelale modelinde yer alan safhaları ihtiva eder. Aslında her iterasyona bir mini şelale modeli ihtiva ediyor diyebiliriz.



Resim 1.4 Çevik süreç iterasyon modeli

Çevik süreç modelinin avantajları şöyledir:

1. Çevik projelerde müşteri gereksinimleri (requirement) ve bu gereksinimlerin karşılığı olan uygulama paralel olarak gelişir. Müşteri projenin gidişatına her zaman müdahale etme yetkisine sahiptir. Bu uzun süren projelerde önem taşımaktadır, çünkü çoğu zaman proje başlangıcında müşteri kendi gereksinimlerini tam olarak bilmeyebilir. Zaman içinde oluşan ilk prototipler müşterinin kafasında nasıl bir sistem istediği hakkında daha net bir resmin oluşmasını sağlar. Projedeki geri bildirim mekanizmalarıyla müşteri gereksinimleri her zaman değişikliğe uğrayabilir.
2. Bu modelde geri bildirim merkezi bir rol oynamaktadır. Çeşitli safhalarda sağlanan geri bildirim ile projenin hangi durumda olduğu saptanır. Programcılar hazırladıkları testler aracılığıyla geri bildirim sağlayarak, implemente ettikleri komponentlerin hangi durumda olduklarını tespit ederler. Sürekli entegrasyon yapılarak programın hangi durumda olduğu geri bildirim olarak elde edilir. Müşteriye kısa aralıklarla programın yeni sürümü sunularak, geri bildirim sağlanır.
3. Proje başlangıcında detaylı dokümantasyon ve tasarım oluşturulmaz. Programcılar test güdümlü (Test Driven Developlement) çalışır ve oluşan tasarımı her yeni testle gözden geçirirler. Eğer tasarım yetersiz kalırsa,

gerekli tasarım değişikliklerini ilerideki testlerin çıkmaza girmesini engellemek için uygularlar.

4. Her iterasyon başlangıcında müşteri tarafından dile getirilen gereksinimler analiz edilir ve implemente edilecek olanlar seçilir. Müşteri her gereksinim için bir öncelik sırası belirler. Öncelik sırası yüksek olan gereksinimler öncelikli olarak implemente edilir. Her iterasyon sonunda gerekli değerlendirmeler yapılarak, oluşan problemler tartışılır ve tekrar meydana gelmelerini engellemek için gerekli önlemler alınır.
5. Test güdümlü çalışıldığı için kod kalitesi çok yüksek olur. Gün boyunca programcılar kendilerine değişik bir programcıyı takım arkadaşı olarak seçerek (pair programming), implementasyonu gerçekleştirirler. Kısa bir zaman sonra programcılar arasındaki teknik bilgi aynı seviyeye ve her programcı programın herhangi bir bölümünde çalışacak hale gelir. Bu şekilde bir programcının kod hakkında bilgi monopolüne sahip olması engellenir.
6. Programcılar aktif olarak proje planlamasında yer alırlar. Onlar gereksinimlerin tespitinde müşteriye yardımcı olurlar ve zaman tahminlerinde bulunarak, proje planlaması için gerekli verilerin oluşturulmasını sağlarlar. Bu programcılara belirli bir sorumluluk yükler. Kendisine güvenildiğini bilen ve sorumluluk sahibi bir programcının öz güveni ve motivasyonu artar.
7. Çevik projelerde iyi bir çalışma ortamının ve temposunun oluşturulması fazla mesai yapılmasını engeller. Fazla mesai yapılmayacak diye bir kural yoktur. Lakin fazla mesai bir kural haline gelmemelidir. Bu durum tüm ekibin motivasyonunu negatif etkiler. Hatalar genelde isteksizce yer alınan fazla mesailerde meydana gelir. Proje çalışanları sekiz saat olan ve fazla mesai yapılmayan iş günlerinde daha verimli olurlar.
8. Müşteriye kısa aralıklarla çalışabileceği bir sürüm sunulur. Program tamamlanmamış olsa bile, müşteri hazır bölümleri kullanarak, yaptığı yatırımın hızlı şekilde geri dönmesini sağlanır.
9. Programcılar ve müşteri arasında devamlı iletişim vardır. Programcılar soru ve sorunları müşteri ile paylaşarak, kısa sürede çözüm üretebilirler.
10. Müşterinin piyasadaki değişikliklere ve bununla birlikte rekabet ortamına ayak uydurabilmesi önemlidir. Çevik süreç hızlı reaksiyon göstererek, bu değişikliklere ayak uydurulmasını sağlar. Rekabete ayak uydurabilmek için hızlı reaksiyon gösterebilmek hayatı bir önem taşımaktadır.

Çevik Süreç Türleri

Zaman içinde bir metamodel olarak kabul edebileceğimiz çevik süreci implemente eden değişik çevik süreç türleri oluşmuştur. Bunların çoğu çevik manifestodan sonra oluşmuştur. Ama bazıları çevik manifesto öncesinde mevcuttu ve kullanılmaktaydı.

Önemli çevik süreç türlerini şunlardır:

- ***Scrum*** - Scrum seksenli yıllarda Kent Schwaber ve Jeff Sutherland tarafından geliştirilmiş bir çevik süreçtir. Scrum Rugby oyununda kullanılan bir terimdir. Oyuncular kısa bir süre için bir araya gelerek, bir sonraki oyun hamlesi hakkında fikir alışverişinde bulunurlar, yani kısa bir toplantı yaparlar. Scrum daha çok proje yönetim metotlarına konsantre olmaktadır. Yazılımın nasıl yapılması gerektiği hakkında detay ihtiva etmez. Birçok projede Scrum Extreme Programming (XP) ile kombine edilir.
- ***XP*** - Extreme Programming (XP) doksanlı yılların sonunda Kent Beck, Ron Jeffries ve Ward Cunningham tarafından Chrysler için yapılan bir proje sonrasında oluşmuş bir çevik süreçtir. XP Scrum dan esinlenilerek geliştirilmiş bir çevik süreçtir. XP Scrum ın aksine daha çok yazılım metotlarına konsantre olmaktadır. Bu sebepten dolayı Scrum ve XP bir projede kombine edilebilir.
- ***IXP*** - XP den doğan IXP nin (Industrial XP) amacı XP yi geliştirmek ve XP de yer alan metot ve tekniklerin daha büyük organizasyonlar için adapte etmektir.
- ***FDD*** - Jeff DeLuca tarafından doksanlı yılların sonunda geliştirilmiş bir çevik süreç türüdür (FDD = Feature Driven Development). FDD yazılım özelliği (feature, function) güdümlü çalışır. Sisteme yeni bir özellik kazandırılmadan önce detaylı bir tasarım çalışması yapılarak bu özelliği kapsayan mimarik yapı oluşturulur. Bu yüzden FDD daha çok tasarım odaklı işleyen bir çevik süreçtir.

Extreme Programming (XP)

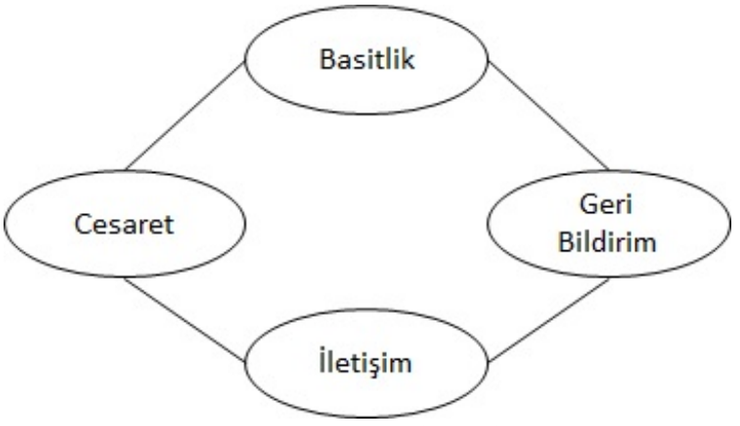
En popüler çevik süreçlerden (Agile Process) birisi XP olarak bilinen Extreme Programming dir. Kent Beck ve arkadaşları tarafından 1996 yılında Chrysler firmasında yapılan bir proje bünyesinde oluşan XP ihtiva ettiği basit ama bir o kadar etkili yöntemlerle yazılım sektöründe yeni bir rüzgarın esmesini sağlamıştır.

XP ile oluşturulan çevik süreçte müşteri ve gereksinimleri merkezi bir rol

oynamaktadır. Yazılım esnasında XP ile tam belirli olmayan ve çabuk değişikliğe uğrayan müşteri gereksinimlerine ayak uydurulabilir. Bu konvansiyonel yazılım metotlarda mümkün değildir, çünkü proje öncesi müşteri gereksinimleri en son detayına kadar kağıda dökülmüştür. Oluşan bir dokümantasyon baz alınarak, yazılım gerçekleştirilir. Proje ilerledikçe müşteri tarafından yapılması istenen değişikliklerin maliyeti çok yüksek olacaktır, çünkü mevcut yapı (tasarım – design) istenilen değişiklerin yapılmasını engelleyebilir ya da yeni bir yapılanmaya gidilmesi gerekebilir. XP, kullanıldığı projelerde formalite ve bürokrasinin mümkün en az seviyeye çekilmesine önem verir. Çevik olabilmek için az yükle yola çıkılması gerekmektedir. Bu yüzden proje öncesi geniş çapta tasarım ve dokümantasyon oluşturulmasına izin verilmez.

XP Değerleri

XP dört değer üzerine kuruludur: Basitlik (Simplicity), İletişim (Communication), Geri Bildirim (Feedback) ve Cesaret (Courage).



Resim 1.5 XP değerleri

XP nin özü bu dört değer içinde yatmaktadır. Bu değerler yaşandığı taktirde XP öğrenimi ve kullanımı kolaylaşır. Bu değerlerin geçerlilik bulmadığı ortamlarda XP nin uygulandığı söylenemez. XP ile verimli bir çevik süreç oluşturabilmek için bu değerlerin hepsinin kabul görmesi ve uygulanması gerekmektedir.

Bu değerlerin ne anlama geldiğini yakından inceliyeyim. XP basit yöntemler aracılığıyla sonuca ulaşmak ister, çünkü sadece bu şekilde hızlı ve düşük maliyetli projeler gerçekleştirilebilir. Bunun yanı sıra basit çözümlerle oluşturulan programın bakımı ve geliştirilmesi kolaydır. Basit çözümler kolay anlatılır ve adapte edilir. Bu zaman kazanılması anlamına gelmektedir.

Yazılımda en önemli konulardan birisi kalite kontrolüdür. XP projelerinde kalite kontrolü geri bildirim üzerinden sağlanır. Programcılar yazdıkları testlerden geri bildirim alarak kaliteyi sağlarlar. Kısa zamanlarla yeni sürüm oluşturularak müşteri ve kullanıcılardan geri bildirim aracılığıyla programın gereksinimleri tatmin edip, etmediği kontrol edilir. Yazılım esnasında sürekli entegrasyon yapılarak, programın en son durumu hakkında geri bildirim sağlanır. XP nin uygulanabilmesi için değişik katmanlarda geri bildirim mekanizmalarının oluşturulması gerekmektedir. İnsanlar için su ne ise, XP için geri bildirim odur.

Tüm proje çalışanlarının sürekli olarak aralarında iletişim kurmaları gerekmektedir. Bireyler arası yüz yüze görüşmeler büyük önem teşkil etmektedir. Sadece bu sayede sağlıklı bilgi transferi gerçekleşebilir. Böylece yanlış anlaşılmalar ve bilinmeyenler ortadan kaldırılır. Eğer takım içinde iletişim ve interaksiyon güçlü ise, dokümantasyon oluşturma ve kullanma gereksiz hale gelebilir. Bu zaman kaybını önler. Dokümantasyon yazılımı ve kullanımı başka sebeplerden dolayı gerekli olabilir, ama projenin başarısı için dokümantasyon öncelikli rol oynamamaktadır.

Basit çözümler, geri bildirim ve iletişim için cesaret gereklidir. Bu saydığımız değerler bireyler arası interaksiyonu ve iletişimi artıracağı için bireylerin kendi iç dünyalarını terk edip, takımın bir parçası olmalarını kolaylaştırırlar. Bu kişisel gelişmeyi sağlar ve temelinde kişisel cesaret yatar.

XP Prensipleri

XP değerlerinden yola çıkarak on beş XP prensibi oluşturulmuştur. Bunlar:

1. Rapid Feedback

Hızlı geri bildirim

Sık ve hızlı geri bildirim edinmek, projenin gidişatını olumlu etkiler. Geri bildirim sayesinde yanlış anlaşılmalar ve hatalar ortadan kaldırılır.

2. Assume Simplicity

Basitliği tercih etmek

Basit çözümler kolay implemente edilir ve kısa zamanda oluşturulur. Bu geri bildirimin de hızlı bir şekilde gerçekleşmesini sağlar. Basit çözümlerin

kavranması ve anlatılması daha kolaydır. XP programcılardan o anki gereksinimi tatmin etmek için basit çözümü bekler. Programcı gelecekte oluşabilecek eklemeleri ve değişiklikleri düşünmemeli, sadece ve sadece kendisinden o an için bekleneni en basit haliyle implemente etmelidir.

3. Incremental Change

Inkrementel değişiklik

Basit çözümler uygulasak bile, yazılım sistemleri zaman içinde karmaşık bir yapıya dönüşebilir. Yapılan en ufak bir değişiklik bile sistemin düşünmediğimiz bir bölümü üzerinde hata oluşmasına sebep verebilir. Oluşabilecek bu hataları kontrol altında tutabilmek için değişikliklerin ufak çapta olması gerekmektedir. Büyük değişiklikler beraberinde büyük sorunları getirebilir. Bu sebepten dolayı değişikliklerin ufak çapta ve sıklıkla yapılması gerekmektedir.

4. Embracing Change

Değişimi istemek

İlerleyebilmek için kendimize bir yön tayin etmemiz ve yeniliklere açık olmamız gerekiyor. Yeniliklere açık olmak cesaret gerektirir. Bilinmeyenle uğraşmak, rahatsız edici olabilir, ama başarıyı elde edebilmek için değişimi istemek gerekir.

5. Quality Work

Kaliteli iş

XP projelerinde kaliteli işin ortaya konabileceği bir ortamın oluşturulması gerekmektedir. Hiçbir programcı hatalı program yazmak istemez. Çalışma ortamınında etkisiyle yüksek kalitede yazılım yapmak hem programcının öz güvenini artırır hem de müşteriyi tatmin edici ürünlerin ortaya konmasını sağlar.

6. Teach Learning

Öğrenmeyi öğret

XP programcı takımlarında tertipçilik ve kıdem farkı yoktur. Tecrübeli programcılar bilgilerini daha az tecrübeli programcılarla paylaşarak, hem bilginin çoğalmasını sağlarlar hem de takım arkadaşları ile teknik olarak aynı seviyeye gelirler. Programcılara komutlar vererek iş yaptırmak yerine, kendiliğinden bazı şeyleri öğrenerek, görevlerini yerine getirmeleri

sağlanmalıdır.

7. Small Initial Investment

Az başlangıç yatırımı

XP en modern ve pahalı araç gereçlerle projeye başlanmasını beklemez. Başlangıç giderleri ne kadar düşük tutulabilirse, projenin iptali durumunda kayıplar o oranda az olacaktır. Başlangıçta tüm takımın dar bir finansman korsası giymesi sağlanarak, proje için daha önemli görevlere odaklanmaları sağlanır. Bunu bir örnekle açıklayabiliriz. Proje başlangıcında en son Oracle veri tabanı ve kullanıcı araçları (Toad gibi) satın alınarak, tüm ekibe bu veri tabanı ve araçları nasıl kullanacaklarına dair eğitim verilebilir. Bu çok masraflı ve bir o kadar da gereksiz bir şeydir. Projeye açık kaynak ve ücretsiz olan HSQL ya da PostgreSQL veri tabanı ile başlanabilir. Programcı ekip böylece ne bir hafta tanımadıkları bir ürün için harcamış, ne de büyük masraflar yapılarak şu an için gerek olmayan bir altyapı komponenti satın alınmış olur. Gerekli araçlar zamanı geldiğinde edinilmelidir.

8. Play to win

Kazanmak için oyna

XP takımları kazanmak için oynar. Her zaman gözlerinin önünde nihai sonuç vardır: programı tamamlamak ve müşteriye teslim etmek. XP programcı takıma tünelin sonundaki ışığı görmek için gerekli tüm imkanları sunar.

9. Concrete Experiments

Somut denemeler

Verdiğimiz kararların sonuçlarını kontrol edebilmek için denemeler yaparız, çünkü alınan kararlar her zaman doğru olmayabilir. Bir kontrol mekanizmasına ihtiyacımız olduğu belli. Bu da somut denemeler aracılığıyla nerede olduğumuzu tespit etmekten geçer. Bu somut denemeler yazılım sistemleri içinde geçerlidir. Örneğin testler hazırlayarak, oluşturduğumuz mimari ve tasarımı kontrol ederiz.

10. Open, honest Communication

Açık ve samimi iletişim

Projenin başarılı olabilmesi için bireyler arasında açık ve samimi türde bir

iletişim olması gerekmektedir. Birçok projede bu böyle değildir. Çoğu zaman bireylerin korkuları, deneyimsiz olmaları ya da kendilerini çok beğenmeleri ve diğerlerini kendilerinden alt safhada görmeleri, açık ve samimi bir iletişim ortamının oluşmasını engeller.

11. Work with people's instincs, not against them

Takımın içgüdülerini kullan, onlara karşı koyma

Bireysel içgüdü yanı sıra, bireylerin oluşturduğu takımların da içgüdüsü vardır. Eğer takım bir şeylerin doğru gitmediği hissine sahipse ve bunu dile getiriyorsa, o zaman bir şeyler yolunda gitmiyor demektir. Takımın içgüdüsüne kulak verilmelidir. Bunun göz ardı edilmesi, proje için olumsuz sonuçlar doğurabilir.

12. Accepted Responsibility

Sorumluluk üstlenmek

Sorumluluk birilerine verilmemeli, bireyler kendileri sorumluluk üstlenmeliler. Eğer bir bireye ya da bir takıma yapılması zor bir projenin sorumluluğu yüklenirse, bu birey ya da takım için motivasyonun düşmesini ve kaybetme korkusunun pekişmesini hızlandırır. Eğer bireyler ya da takımlar kendi sorumluluklarını kendileri seçerlerse, hem yaptıkları işte kendilerini iyi hissederler, hem de yüksek motivasyon ile üstlendikleri işi başarıyla tamamlarlar.

13. Local Adaptations

Sürecin ortam şartlarına adapte edilmesi

Büyük bir ihtimalle her takımın XP yi Kent Beck'in anlattığı tarzda harfiyen uygulaması mümkün değildir. Atalarımızın da dediği gibi her yiğidin yoğurt yiyiş tarzı başkadır. Amaç XP yi harfiyen uygulamak değildir, amaç kısa bir zamanda projeyi başarılı bir sonuca ulaştırmaktır. Eğer proje XP de yapılacak değişiklerle başarıya ulaşacaksa, o zaman süreç üzerinde bu değişiklikler yapılmalı ve uygulanmalıdır. Bunda bir sakınca yoktur.

14. Travel light

Az yükle yolculuk yapmak

Projede hızlı ilerleyebilmek için fazla bir yükle yola çıkılmaması gerekmektedir. Beraber çalışmayı kolaylaştırmak için kullanımı kolay araç ve gereçler

seçilmelidir. Formalitelerden uzak durulmalıdır.

15. Honest Measurement

Doğru ölçüm

Proje gidişatını kontrol edebilmek için değişik türde ölçümlerin yapılması gerekmektedir. Örneğin hazırlanan birim testleri ile sınıfların işlevleri kontrol edilir. Yapılan ölçümler doğru ve samimi yapıldığı taktirde kontrol mekanizması olarak kullanılan ölçümlerin bir anlamı vardır. Programcılar tarafından samimi ve doğru yapılmayan ölçümler projenin gidişatını olumsuz yönde etkiler.

XP Teknikleri (XP Practices)

Dört XP değer ve on beş XP prensibi on dört XP tekniği ile desteklenmektedir. XP teknikleri programcıların XP değer ve prensiplerini uygulamada yardımcı olur. Kent Beck tarafından hazırlanan ilk XP versiyonunda on iki teknik yer almaktaydı. Diğer çevik süreçlerin de etkisiyle Standup-Meeting ler ve retrospektif toplantılar XP teknikleri arasına katıldı.

Bunlar:

1. On-site Customer

Programcıya yakın müşteri olarak tercüme edilebilir.

XP projeleri müşteri gereksinimlerine odaklı ilerler. Bu yüzden müşteri ve sistem kullanıcılarının projeye dahil edilmeleri gerekmektedir. Müşteri gereksinimlerini ekibe bildirir. Programcıların implementasyonu gerçekleştirebilmesi için müşteri tarafından dile getirilen gereksinimleri anlamaları gerekmektedir. Yanlış anlaşılmaları ve hataları gidermek için programcıların müşteri ve sistem kullanıcıları ile diyalog halinde olabilmesi gerekmektedir. Bu sebepten dolayı müşteri veya sistem kullanıcılarının programcıların erişebileceği bir uzaklıkta olmaları gerekir. Tipik XP projelerinde müşteri ve programcılar aynı odada beraber çalışırlar. Müşteri ekibin sorularını zaman kaybı olmadan cevaplar ve projenin ilerlemesine katkıda bulunur.

2. Standup-Meeting

Ayakta toplantı

Proje çalışanları her gün 15 dakikayı aşmayan ve ayakta yapılan toplantılarda bir araya gelirler. Bu toplantının amacı, projenin gidişatı hakkında bilgi alışverişinde bulunmaktır.

3. Planning Game

Planlama oyunu

XP projeleri iteratif ve inkrementel yol alır. Bir sonraki iterasyonda yapılması gereken işleri planlama oyununda görüşülür ve sürüm ve iterasyonun içeriği tespit edilir. Planlama oyununa müşteri, kullanıcılar ve programcılar katılır. Müşteri ve kullanıcılar daha önce kullanıcı hikayesine (user story) dönüştürdükleri isteklerine öncelik sırası verirler. Programcılar her kullanıcı hikayesi için gerekli zamanı tahmin ederler. Kullanıcı hikayelerinin öncelik sırası bu tahmine bağımlı olarak değişebilir. Planlama oyunlarında sürüm ve iterasyon planları oluşur.

4. Short Releases

Kısa aralıklarla yeni sürüm

XP projelerinde yeni implemente edilen ve değişikliğe uğrayan komponentler yeni sürümler oluşturularak müşteri ve kullanıcının beğenisine sunulur. Bu sayede hem müşteriler çalışır durumda olan programdan faydalanabilir hem de yeni sürümü inceleyerek, gereksinimleri ile örtüşüp, örtüşmediğini kontrol edebilirler. Eğer yeni sürüm müşteriyi tatmin edecek durumda değilse, gereksinimler değişikliğe uğrayabilir. Bu değişiklikler bir sonraki iterasyonda göz önünde bulundurularak, müşteri istekleri ile yüksek derecede örtüşen bir programın oluşturulması sağlanır.

5. Retrospective

Geriye bakış

Proje çalışanları düzenli aralıklarla geriye bakarak, meydana gelen sorunları gözden geçirirler. Buradaki amaç gelecekte bu sorunların tekrarını önlemektir. Geriye bakış bir ile altı aylık zaman birimleri için tüm proje çalışanları ya da seçilen bireyler tarafından yapılır. Geriye bakış toplantıları yarım gün ile üç gün arasında sürebilir.

6. Metaphor

Mecaz

XP projelerinde hazırlanan program için bir veya birden fazla, programın nasıl bir işlevi olacağını ekibin gözünde canlandırmalarını sağlayacak mecazi isim, öğe ya da resimler kullanılır. Bunlar proje çalışanlarının ortak bir payda da buluşarak, ne yapılması gerektiği hakkında bir fikir sahibi olmalarını kolaylaştırır. Örneğin bir shop sistemi yazılımı yapılacak. Burada metaphor olarak alışveriş sepeti kullanılabilir. Alışveriş sepetini duyan her programcının aklında bir shop sisteminin programlanması gerektiği fikri doğar.

7. Collective Ownership

Ortak sorumluluk

XP projelerinde programcılar ortak sorumluluk taşırlar. Bu her kod parçasının herhangi bir programcı tarafından gerekli durumlarda değiştirilebileceği anlamına gelir. Böylece yapılması gereken işler aksamaz, çünkü belli kod bölümlerinden belli programcılar sorumlu değildir. Aksine her programcı programın her bölümü üzerinde çalışma hakkına sahiptir. Bir programcının işe gelmemesi durumunda, başka bir programcı kolaylıkla onun görevlerini üstlenebilir.

8. Continuous Integration

Sürekli entegrasyon

Sistem değişiklikleri ve yeni komponentler hemen sisteme entegre edilerek test edilir. Sürekli entegrasyon sayesinde yapılan tüm değişiklikler her programcının sistem üzerinde yapılan değişiklikleri görmesini sağlar. Ayrıca sistem entegrasyonu için gerekli zaman azaltılır, çünkü oluşabilecek hatalar erken teşhis edilerek, ortadan kaldırılır.

9. Coding Standards

Kod standartları

Programcılar tarafından aynı kalitede kod yazılımı yapılabilmesi için kod yazarken kullanılacak kuralların oluşturulması gerekmektedir. Kodun nasıl formatlanacağı, sınıfların, metot isimlerinin ve değişkenlerin nasıl isimlendirileceği kod standartlarında yer alır.

10. Sustainable Pace

Kalıcı tempo

XP projelerinde programcılar haftalık belirli mesai saatlerini aşmazlar. Gereğinden fazla çalıştırılan ve yorulan bir programcıdan verimli iş yapması beklenemez. Programcıların motivasyonun ve çalışma enerjilerinin yüksek olması için günde sekiz saatten fazla çalışmalarına izin verilmemelidir. Bazen fazla mesai saatlerine ihtiyaç olabilir. Eğer durum devamlı böyle ise, bu proje gidişatında bazı olumsuzlukların göstergesi olabilir.

11. Testing

Test etmek

Oluşturulan programların kalite kontrolünden geçmesi gerekmektedir. Bu yazılım esnasında oluşturulan testlerle yapılır. Programcılar komponentler için birim testleri hazırlar. Sınıf bazında yapılan bu testlerle komponentlerin işlevleri kontrol edilir. Müşteri gereksinimlerini test etmek için onay/kabul testleri hazırlanır. Komponentlerin entegrasyonunu test etmek için entegrasyon testleri hazırlanır.

12. Simple Design

Sade tasarım

Programcılar üstlendikleri görevleri (task) en basit haliyle implemente ederler. Bu programın basit bir yapıda kalmasını ve ilerde değiştirilebilir ve genişletilebilir olmasını sağlar. Sade bir tasarım yazılım sisteminin karmaşık bir yapıda olmasını önler. Bunun yanı sıra basit tasarımlar daha kolay ve daha hızlı implemente edilebilir. Basit bir implementasyonu anlamak ve anlatmak daha kolaydır.

13. Refactoring

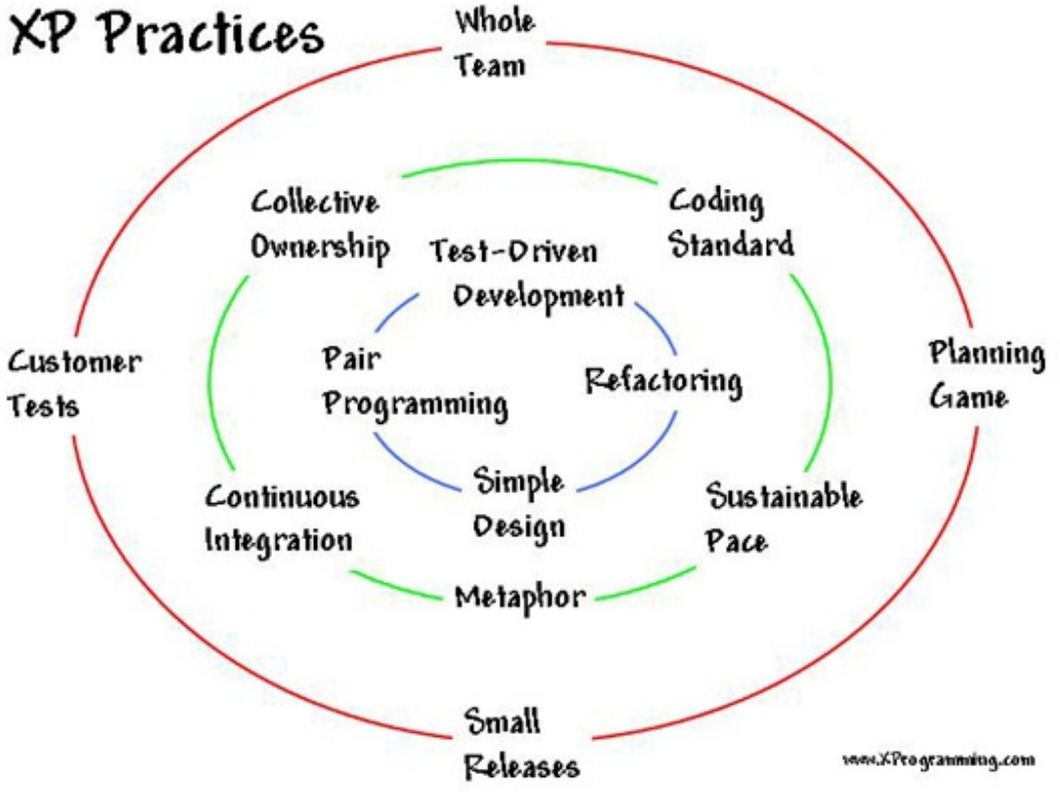
Yeniden yapılandırma

Tasarım hataları yazılım sisteminin daha ilerde tamir edilemeyecek bir hale dönüşmesine sebep verebilir. Bu yüzden bu hatalar hemen giderilir. Bu yeniden yapılandırma işlemine refactoring ismi verilir. Hazırlanan birim testleri ile yapılan değişikliklerin yan etkileri kontrol edilir. Bu açıdan bakıldığında birim testi olmayan bir sistem üzerinde yeniden yapılandırma işlemi hemen hemen mümkün değildir, çünkü değişikliklerin doğurduğu yan etkileri tespit etme mekanizması bulunmamaktadır.

14. Pair Programming

Eşli programlama

XP projelerinde iki programcı aynı bilgisayarda çalışır. Bu sayede programcıların kısa bir zaman içinde aynı seviyeye gelmesi sağlanır. Ayrıca bu kalitenin yükselmesini sağlar.



Resim 1.6 XP teknikleri

XP Rolleri

Bir çevik projede ekip çalışanlarının sorumluluk alanlarını tanımlamak için roller tayin edilir. Her rol beraberinde bazı sorumluluklar ve tanımlanmış haklar getirir. Bu roller sabit değildir. Ekip içinde değişik kişilere değişik roller verilebilir ve daha sonra rol değişikliği yapılabilir. Proje gereksinimleri doğrultusunda yeni rollerin oluşturulması mümkündür.

Müşteri (Customer)

Projenin var olma sebebi müşteridir. Müşteri ihtiyaç duyduğu ve gereksinimlerine cevap verebilecek bir yazılım sistemi için yatırım yapan kişidir.

XP projelerinde şu atasözümüz geçerlidir: “parayı veren düdüğü çalar!” ;-).

Proje bünyesinde ne programlanması gerektiğini müşteri tayin eder. Müşteri yapılması gerekenleri kullanıcı hikayeleri (user story) oluşturarak ifade eder. Programcılar müşteriye bu süreçte yardımcı olurlar. Ama kullanıcı hikayelerinin oluşturulma sorumluluğu büyük ölçüde müşteriye aittir. Her kullanıcı hikayesi yazılım sisteminin bir özelliğini tanımlar. Programcıların implementasyonu gerçekleştirebilmeleri için kullanıcı hikayesini anlayabilmeleri gerekmektedir. Sadece bu durumda implementasyon süreci için bir tahminde bulunabilirler. Ayrıca oluşturulan kullanıcı hikayelerinin test edilebilir yapıda olması gerekmektedir.

Müşteri çalışma alanı (domain knowledge) hakkında bilgiye sahip olan kişidir. Programcılar müşteriye karşılaştıkları sorunları çözmek için sorular sorabilirler. Bu soruların cevabını en iyi verebilecek şahıs müşteridir.

Hangi kullanıcı hikayelerinin implemente edileceğine müşteri karar verir. Bu konuda müşteriye herhangi bir sınırlama getirilmez. Müşteri seçer ve programcılar implemente eder.

İmplemente edilen kullanıcı hikayelerini kontrol etmek amacıyla müşteri onay/kabul testleri tanımlar. Bu testler programcı ya da testçi tarafından implemente edilir. Onay/kabul testleri kullanıcı hikayesini doğru implemente edilip, edilmediğini kontrol edici bir mekanizmadır.

Programcı

Sistem analizi, tasarım, test ve implementasyon programcılar tarafından yapılır.

Müşteri tarafından hazırlanan kullanıcı hikayelerinin implementasyon süresi programcılar tarafından tahmin edilir. Bu programcıların XP projelerinde proje planlama sürecine dahil edildikleri anlamına gelmektedir. Geleneksel projelerde bu tahminleri teknik bilgiye sahip olmayan yöneticiler yapmak zorundadır. Bu yüzden bu tahminler genelde gerçekleri yansıtmaz.

Her programcı test güdümlü ve bir takım arkadaşıyla beraber çalışır. Pair programming olarak bilinen, iki programcının birlikte yazılım yapması, kısa zamanda kod hakkındaki bilginin tüm programcılar tarafından paylaşılmasını kolaylaştırır. Ayrıca pair programming programcılar arasında iletişimi ve takım içinde çalışabilme özelliğini artırır. Test güdümlü çalışmak bakımı ve geliştirilmesi kolay kodun oluşmasını sağlar. XP projelerinde programcılar

herhangi bir satır kod yazmadan önce gerekli test sınıflarını oluşturarak implementasyona başlarlar.

Programcılar oluşturdukları testler yardımıyla her gün bir veya birden fazla şekilde modül entegrasyonu gerçekleştirirler. Sürekli entegrasyon programcılar tarafından oluşturulan modülleri entegre eden bir süreçtir. Her programcı bu süreçten geri bildirim sağlayarak, kendi yaptıklarının ne derecede sisteme entegre olduğunu ölçebilir.

Proje Menajeri

Proje menajeri müşteri ve programcıları bir araya getirir. Onların beraber çalışabilecekleri ortamların oluşmasını sağlar. XP proje menajeri tek başına proje planlamasından sorumlu değildir. Programcılara görev atamaz, onların kendi başlarına seçim yaparak, sorumluluk almalarını kolaylaştırır. Toplantı ve diğer buluşmaları koordine eder, takımın karşılaştığı sorunları ortadan kaldırmak için gerekli olanları yapar.

Koç

Çevik süreci tanıyan ve nasıl uygulanması gerektiğini bilen experdir. Koçun görevi proje başlangıcında çevik takımı oluşturmak ya da bir araya getirmek ve onlara belirli bir süre rehberlik yapmaktır. Koç projede sorun çıktığı zaman ya da takım XP yöntemlerinin dışına çıktığında müdahale eder. Zaman zaman koç implementasyonda aktif olarak rol alır. Örneğin birim testlerin nasıl doğru bir yapıda oluşturulabileceğini diğer programcılara gösterebilir.

Testçi

Müşteri tarafından oluşturulan onay/kabul testlerini implemente eden programcıdır. Aynı zamanda birim ve entegrasyon testlerinin implementasyonunda takım arkadaşlarına yardımcı olur.

Haklar ve Sorumluluklar

Proje çalışanları çoğu zaman içgüdüsel korku hissedebilirler. Bunun en büyük sebebi belirsizliktir. Ne ve nasıl yapılması gerektiği bilinmediği taktirde ekip içinde huzursuzluk doğar. Bu projenin başarısını negatif etkiler.

Projenin başarılı olabilmesi için çalışanların hissettikleri korkunun aza indirilmesi ya da yok edilmesi gerekmektedir. XP bu konuda proje çalışanlarına

bir takım hakların tanınması gerektiğini belirtir. Hangi rollerin hangi haklara sahip olduğunu yakından inceleyelim.

Müşteri Hakları

Müşterinin sahip olduğu haklar şu şekilde özetlenebilir:

- Müşteri bütçe ve zaman planlaması yapabilmek için neyin yapılabilir olduğunu ve hangi zaman biriminde yapılabileceğini bilme hakkına sahiptir.
- Müşteri fikir değiştirerek, gereksinimler üzerinde değişiklik yapma ve yeni gereksinimlerin implementasyonunu talep etme hakkına sahiptir.
- Müşteri programcılar tarafından sağlanabilecek en yüksek verimi ve değeri elde etme hakkına sahiptir.
- Müşteri projede somut ilerlemeyi görme hakkına sahiptir. Bu kısa aralıklarla oluşturulan yeni sürüm ve onay/kabul testlerine olumlu cevap veren yeni implementasyonlarla sağlanır.
- Müşteri bir sonraki sürümde implemente edilecek kullanıcı hikayelerini seçme hakkına sahiptir.

Programcı Hakları

Programcının sahip olduğu haklar şu şekilde özetlenebilir:

- Programcı takım arkadaşlarına soru sorma ve cevap alma hakkına sahiptir.
- Programcı kullanıcı hikayeleri için implementasyon zaman tahmini yapma hakkına sahiptir. Programcı tahminler üzerinde değişiklik yapma hakkında sahiptir.
- Programcı, kendisine görev verilmesi yerine sorumluluk alma hakkında sahiptir.
- Programcının her zaman yüksek kalitede iş çıkarma hakkı vardır.
- Programcının hangi öncelik sırasına göre neyi yapması gerektiğini bilme hakkı vardır.

Süreç İşleyişi

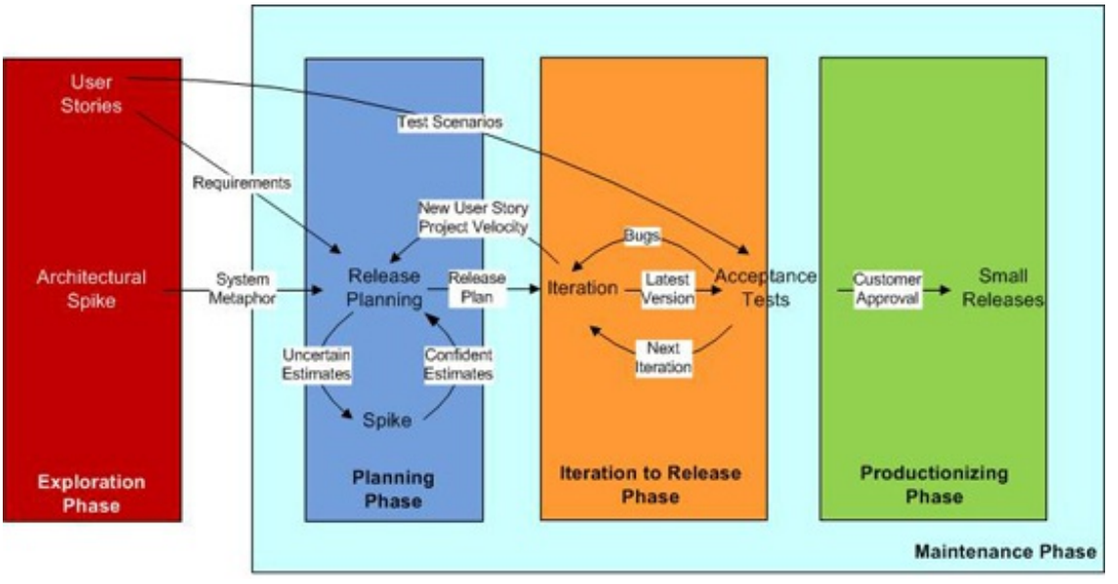
XP projelerinde projenin gidişatını genel olarak şu şekilde özetleyebiliriz:

- Müşteri gereksinimleri ihtiva eden kullanıcı hikayelerini oluşturur. Bunlara öncelik sırası atar. Programcılar her kullanıcı hikayesi için implementasyon zamanını tahmin ederler

- Müşteri programcılarla beraber iterasyon(1-2 hafta) ve sürüm (1-2 ay) planını hazırlar. Her sürüm birden fazla iterasyon ihtiva eder ve müşteri tarafından kullanılacak özellikte çalışır bir sistemdir.
- Müşteri ilk iterasyon (1 hafta) için gerekli kullanıcı hikayelerini programcı tahminleri de dikkate alarak seçer.
- Programcılar iterasyon için seçilen kullanıcı hikayelerini implemente ederler. Kafalarda oluşan sorular müşteriye danışılarak cevaplandırılır.
- İterasyon sonunda programcılar müşteriye çalışır bir sistem sunarlar. Müşteri sistemi değerlendirerek programcılara geri bildirim sağlar.
- Edinilen tecrübeler ışığında bir sonraki iterasyon planlanır. Eğer müşteri yeni gereksinimlerin implemente edilmesini isterse, bunlar için tekrar kullanıcı hikayeleri oluşturulur ve tahminler yapılır. Eğer yeni kullanıcı hikayeleri yoksa, mevcut kullanıcı hikaye listesinden en yüksek öncelik sırasına sahip olanlar seçilir ve bir sonraki iterasyondan implemente edilir.
- İlk sürüm sonunda oluşan yazılım sistemi müşteri tarafından kullanıma alınır. Başka sürümler planlandıysa bir sonraki iterasyonla devam edilir.

XP Proje Safhaları

Bir XP projesi değişik safhalardan oluşur. Her safha, bünyesinde kendine has aktiviteler ihtiva eder. Resim 1.7 de bir XP projesinde olması gereken safhalar yer almaktadır.



Resim 1.7 XP proje sahfaları ve aktiviteler
(http://www.agilemodeling.com/essays/agileModelingXPLifecycle.htm)

XP projesi şu safhalardan oluşur:

- ***Keşif safhası (Exploration Phase)*** - Projenin başlangıcında keşif safhasını oluşturan aktiviteler yer alır. Bu safhada müşteri kullanıcı hikayelerini (user story) oluşturur. Programcılar teknik altyapı için gerekli deney (spike) ve araştırmayı yaparlar.
- ***Planlama Safhası (Planning Phase)*** - Keşif safhasını planlama safhası takip eder. Bu safhada müşteri programcılar yardımıyla iterasyon ve sürüm planlarını oluşturur. İterasyon planlaması için oluşturulan kullanıcı hikayelerinin implementasyon süresi programcılar tarafından tahmin edilir. Müşteri kullanıcı hikayelerine öncelik sırası vererek, iterasyonlarda hangi kullanıcı hikayelerinin öncelikli olarak implemente edilmeleri gerektiğini tespit eder. Programcılar tarafından herhangi bir kullanıcı hikayesinin implementasyon süresi tahmin edilemezse, programcılar spike solution olarak bilinen basit bir çözüm implemente ederek, kullanıcı hikayesinin gerçek implementasyonu için gerekli zamanı tahmin etmeye çalışırlar.
- ***İterasyon ve Sürüm Safhası (Iterations to Release Phase)*** - Kullanıcı hikayelerinin implementasyonu iterasyon ve sürüm safhasında gerçekleşir. Bir iterasyon bünyesinde implemente edilmesi gereken kullanıcı hikayeleri müşteri tarafından belirlenir. İmplementasyonunun işlevini kontrol etmek için müşteri tarafından onay/kabul testleri belirlenir. Bu testler programcılar ya da testçiler tarafından implemente edilir. Her iterasyon sonunda müşteriye çalışır bir yazılım sistemi sunulur. Bu şekilde müşterinin sistem hakkındaki görüşleri alınır (geri bildirim). İterasyon son bulduktan sonra çalışma hızını tahmin etmek için bir önceki iterasyonda elde edilen tecrübeler kullanılır ve iterasyon planı bu değerler doğrultusunda gözden geçirilir. Bir önceki iterasyonda oluşan hatalar bir sonraki iterasyonda gözden geçirilmek ve giderilmek üzere planlanır.
- ***Bakım Safhası (Maintenance Phase)*** - Bu programın bakımının ve geliştirilmesinin yapıldığı safhadır. Bu safhada kullanıcılar için eğitim seminerleri hazırlanır ve küçük çapta eklemeler ve sistem hatalarının giderilmesi için işlemler yapılır. Müşterinin istekleri doğrultusunda bir sonraki büyük sürüm için çalışmalara başlanır. Bu durumda tekrar kesif safhasına geri dönülmesi ve oradan işe başlanması gerekmektedir.

2. Bölüm

Proje Planlama

Giriş

Bir geminin rotası sefer öncesi kaptanı tarafından planlanır. Bu planda geminin demir atacağı limanlar ve seyahatin son noktası olan hedef liman yer alır. Böyle bir planın oluşturulması görevlerin dağıtımı ve hedefin tayini açısından önem taşımaktadır. Yolculuk esnasında dış etkilerden dolayı rotada değişiklikler meydana gelebilir. Kaptan ulaşmak istediği hedefi bildiği için gerekli değişiklikleri yaparak rotayı hedefe uyumlu hale getirir. Bir projenin gidişatını bir geminin rotasıyla kıyaslayabiliriz. Projeninde bir rotası olması gerekir, aksi taktirde hedefe ulaşmak için gerekli çalışmaların nasıl ve hangi zaman biriminde yapılması gerektiği konuşunda yanlış anlaşılmalar oluşur. Projenin rotası proje planında yer alır. Bu bölümde

- proje planının ne olduğunu,
- sürüm ve iterasyon planlamasının nasıl yapıldığını,
- sürüm ve iterasyon planlarının online ve offline ortamlarda nasıl tutulduğunu

yakından inceleyecegiz.

Proje Planı

Proje planı hangi sürede nelerin yapılacağını ihtiva eden bir dokümandır. Her proje öncesi böyle bir planın oluşturulması gerekmektedir. Proje planı olmayan bir proje rotası belli olmayan bir gemi gibidir. Hangi sürede hangi yolun katedileceğini kimsenin kestirmesi mümkün değildir. Bunun yanı sıra hedefin ne olduğu belli değildir.

Proje planının temelinde oluşturulacak yazılım sisteminin vizyonu yatar. Bu vizyondan gereksinimler doğar ve implementasyon bu gereksinimler doğrultuşunda gerçekleşir. İmplementasyon gereksinimlerin program koduna dönüştürüldüğü işlemdir. Programcılar müşteri gereksinimlerinden ve dolaylı olarak vizyondan doğan gereksinimleri implemente ederler. Gereksinimin ne olduğunu müşteri söyler. Proje planı, hangi gereksimin hangi zaman diliminde ve hangi sıraya göre implemente edileceğini ihtiva eden bir dokümandır. Bunun yanı sıra proje planı vizyonun hayata geçirilmesi için zamansal bir sınırlama getirir. Bu hedefe ulaşmak için gerekli zaman biriminin tanımlanmış halidir.

Proje planının eksikliği vizyonun hayata geçirilmesi için konulması gereken

zamansal sınırın ekliği anlamına gelir. Bu durumda hedefin ne olduğu bilinemez. Hedefin bilinmemesi rotası belli olmayan bir gemide seyahat etmek gibidir.

Sürüm Planlaması (Release Planning)

Bir projenin gidişatını kontrol edebilmek için proje planına ihtiyaç duyulmaktadır. XP projelerinde proje planlaması sürüm planlarının oluşturulmasıyla gerçekleşir. Bu konuda detaya girmeden önce bir sürümün ne olduğu konuşuna açıklık getirmemiz gerekiyor.

Sürüm, bir yazılım sisteminin bir veya birden fazla özellik implementasyonunu ihtiva eden bir versiyonudur. Her sürüm bir ile üç aylık bir yazılım sürecinden sonra oluşan özellikleri ihtiva eder. Her sürüm müşteri tarafından kullanılabilir yapıdadır. Bazı teknik sebeplerden dolayı kullanılamayan sürümlerde olabilir. Ama yeni bir sürüm oluşturulmasının ana amacı, müşteriye kullanabileceği bir sistemin sunulmasının hedeflenmesidir.

Sürüm planlama aktivitesi müşteriye hangi gereksinimlerinin bir sonraki sürümde olması gerektiği konusunda yardımcı olur. Bunun yanı sıra programcılar sürüm planlaması esnasında implementasyon için gerekli teknik analizi yapma imkanı bulurlar. Programcılar bu süreçte tahminlerde bulunarak sürüm planının oluşumunda aktif rol oynarlar. Sürüm planlama aktivitesi bir ile iki haftalık bir zaman diliminde gerçekleşir.

Sürüm planı projenin yol haritasıdır. Bu planda özelliklerin hangi sıraya göre implemente edileceği ve hangi tarihte yeni sürümlerin oluşturulacağı yer alır.

XP projelerinde sürüm planları sürüm planlama oyunlarında oluşturulur. Bu oyunun nasıl oynandığını yakından inceleyelim.

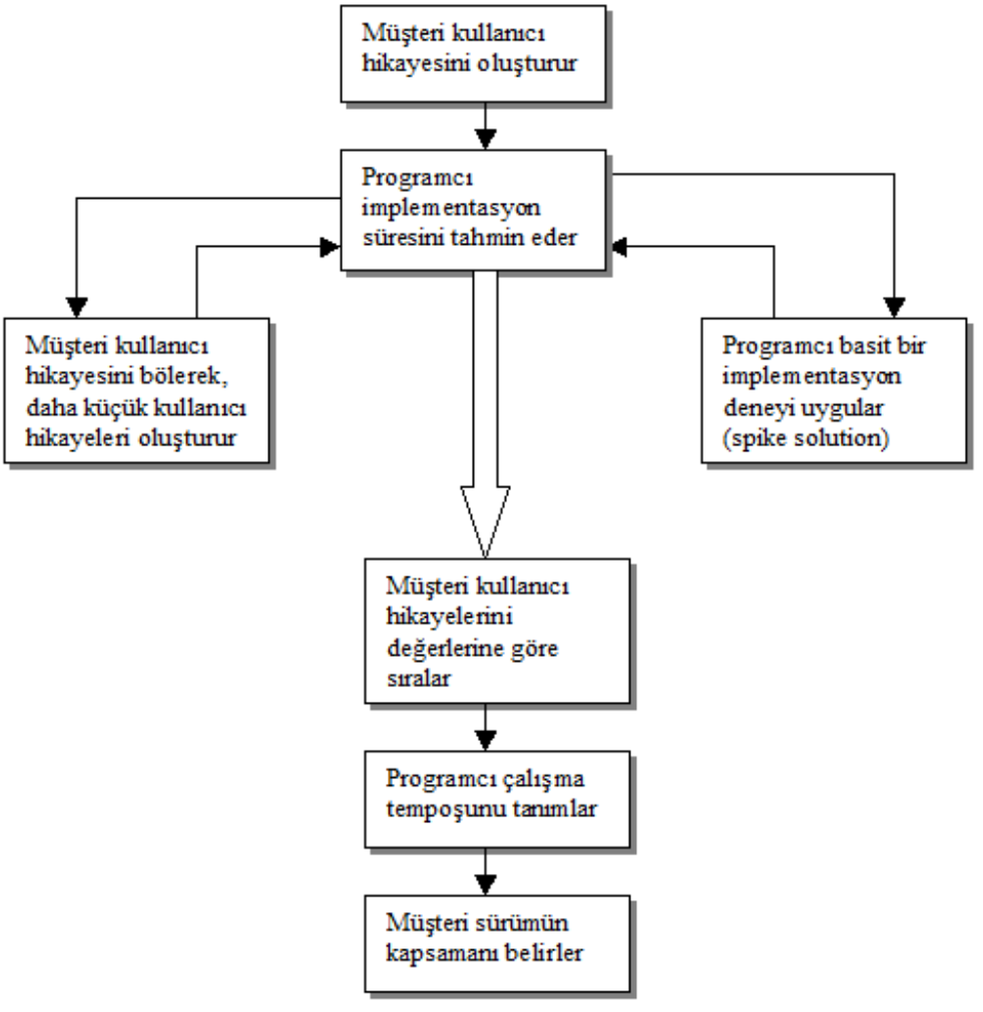
Sürüm Planlama Oyunu (Release Planning Game)

Proje start aldıktan sonra müşteri ve programcılar sürüm planlama oyununda bir sonraki sürümü planlamak için bir araya gelirler.

Bu rekabetin olduğu bir oyun değildir. Daha çok oyuncular birbirlerine yardım ederek, oyunu sonuçlandırmayı hedeflerler. Oyun sürüm planını ihtiva eden dokümanın oluşturulmasıyla son bulur.

Resim 2.1 Programcılar ve müşteri tarafından oynanan sürüm planlama oyunu

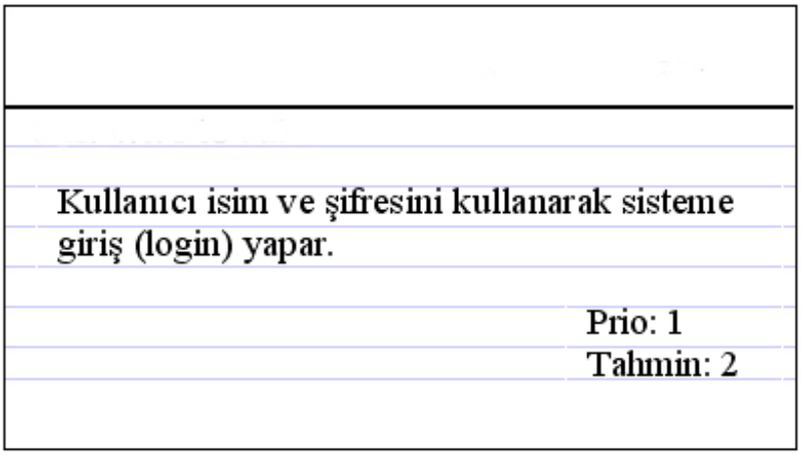
Sürüm planlama oyunu öncesi müşteri gerekli gördüğü tüm kullanıcı hikayelerini oluşturur. Sürüm planlama oyununu için hikaye kartlarına yazılan kullanıcı hikayeleri baz alınır. Oyun içinde yapılabilecek hamleler resim 2.2 de yer almaktadır.



Resim 2.2 Sürüm planlama oyununda yapılabilecek hamleler

Müşteri Kullanıcı Hikayesini Yazar

Sürüm planlama oyunu hikaye kartlarına yazılan (story card) kullanıcı hikayeleri (user story) ile oynanır. Kullanıcı hikayeleri müşteri tarafından oluşturulur ve bir ya da iki cümle ile sistem özelliklerini anlatımda kullanılır. Oluşturulan kullanıcı hikayelerinin programcılar tarafından anlaşılır yapıda olmaları gerekmektedir, çünkü programcılar implementasyonu kullanıcı hikayesini baz alarak yapalar. Bunun yanı sıra oluşturulan kullanıcı hikayelerinin test edilebilir yapıda olması gerekir. Kullanıcı hikayesinin bir örneği resim 2.3 de yer almaktadır. Kullanıcı hikayeleri A5 ya da A6 formatında kalın kartonda yapılmış kartlar (story card) üzerine yazılır.



Resim 2.3 Kullanıcı hikaye kartı (story card)

Planlama esnasında hikaye kartları önemli bir enstrümandır. Hikaye kartları müşteri ve programcılar arasında gereksinimler hakkında fikir alışverişini destekler. Programcılar kullanıcı hikayeleri hakkında tartışarak implementasyon süreleri hakkında fikir edinebilirler. Oluşan sorular müşteriye hemen yöneltilebilir.

Her kullanıcı hikayesi için müşteri öncelik sırası (Prio) belirler ve hikaye kartını tahmin yapılmak üzere programcılara verir.

Programcı Tahmin Eder

Programcı bir kullanıcı hikayesini implementasyon öncesi tüm detaylarıyla bilmek zorunda değildir. Bir kullanıcı hikayesini en son detayına kadar kavramaya çalışmak zaman kaybı olabilir, çünkü kullanıcı hikayesinde yer alan müşteri gereksinimi değişikliğe uğrayabilir. Bu genelde müşterinin programın ilk sürümlerini görmesiyle gerçekleşir. Çalışır bir program aracılığıyla müşteri gereksinimlerini daha iyi kavrayacak ve gerekli değişiklikleri talep edecektir. Bu sebepten dolayı implementasyona başlamadan önce kullanıcı hikayesinin ihtiva ettiği tüm detayları tespit etmek faydalı olmayacaktır, çünkü kullanıcı hikayesi değişikliğe uğrayabilir. Programcı ekibin kullanıcı hikayesi hakkında implementasyon için gerekli zamanı tahmin edebilecek kadar bilgiye sahip olması yeterli olacaktır.

Programcılar müşteri tarafından seçilen kullanıcı hikayesinin implementasyon süresini tahmin ederler. Bu tahminler **planlama pokerinde** yapılır.

Resim 2.4 Planlama poker kartları

Planlama pokeri yapılmayan tahminler bazen sağlıklı sonuçlar vermeyebilir. Bir programcı tarafından yapılan tahmin diğer programcıları etkileyebilir. Ya da bazı programcılar herhangi bir sebepten dolayı tahmin etme sürecine aktif olarak dahil olmayabilirler. Bu gibi sebeplerden dolayı oluşan tahmin süreleri yanıltıcı olabilir. Daha geçerli tahminler elde edebilmek için planlama pokeri oynanır.

Planlama pokeri için kullanılan kartlar resim 2.4 de yer almaktadır. Her programcı bu kartların bir setine sahiptir. Planlama pokeri şu şekilde oynanır: Bir moderatör ilk kullanıcı hikayesini okur. Müşteri bu kullanıcı hikayesi için implementasyon süresinin ne olduğunu sorar. Programcılar kısa bir zaman düşündükten sonra hep beraber planlama poker kartlarından birisini seçerek gösterirler. Çoğu zaman kullanılan kartlardaki değerler farklı olacaktır. En çok süreyi ve en az süreyi tahmin eden programcılardan bu sonuca nasıl vardıklarının açıklanması istenir. Verilen bilgiler doğrultusunda ortak bir değer bulunur.

Resim 2.5 Planlama pokeri oynayan programcılar

Tahminler için **hikaye puanları** (story points) kullanılır. 1 hikaye puanı örneğin 1 iş günü (8 saat) olabilir. Programcılar her kullanıcı hikayesini kendi başına tahmin etmek yerine, kullanıcı hikayelerini birbirleriyle kıyaslayarak tahminde bulunurlar. Örneğin kullanıcı hikayesi A için 2 hikaye puanı tahmin edilmişse, kullanıcı hikayesi B bu değer göz önünde bulundurularak tahmin verilir. Eğer kullanıcı hikayesi B A dan üç katı daha büyükse, o zaman B için tahmin 2x3 = 6 hikaye puanı olarak verilir.

Load Factor

Bir kullanıcı hikayesinin ideal şartlarda implementasyonu için gerekli zaman dilimi ile normal şartlarda implementasyonu için gerekli zaman dilimi farklı

olacaktır. Örneğin programcılar gün boyunca yazılım haricinde toplantı, bilgi alışverişi gibi işler için zaman ayırmak zorundadırlar. Bir programcının sekiz saatlik bir iş gününde sekiz saat program yazabilmesi ideal zaman dilimi olarak tanımlanır. Toplantı ve diğer işler için kullanılan zaman ideal zaman diliminde çıkartıldığı zaman normal zaman dilimi elde edilir. Kullanıcı hikayelerinin tahminlerinde ideal ve normal zaman dilimlerinin göz önünde bulundurulması gerekmektedir, aksi taktirde kullanıcı hikayesi için yapılan tahmin gerçekleri yansıtmayacaktır. Gerçekçi bir tahmin yapabilmek için load factor olarak bilinen değer kullanılır. Bu değer bir kullanıcı hikayesinin implementasyonu için kullanılan zamanın ideal zamana bölünmesiyle elde edilir. Örneğin bir kullanıcı hikayesi için 1 iş günü (8 saat) tahmin edilmiş ve programcı kullanıcı hikayesini 2 iş gününde tamamlamış olsun. Bu durumda load factor 16 / 8 = 2 olacaktır. Load factor için 2 ile 5 arasında bir değer normaldir. Tahmin yapılırken tahmin edilen implementasyon zamanı load factor ile çarpılır. Örneğin programcı bir kullanıcı hikayesini 2 iş gününde implemente edebileceğini düşünüyorsa, kullanıcı hikayesi için tahmin süresi 2 değil, 2 iş günü x 2 load factor = 4 olmalıdır. Bu şekilde daha gerçekçi tahmin elde edilir.

Programcılar load factor değerini göz önünde bulundurarak tahminde bulunurlar.

Programcı Dener

Eğer programcılar bir kullanıcı hikayesi için tahminde bulunamazlarsa küçük çaplı bir demo implementasyonu yaparak implementasyon süresini tahmin etmeye çalışırlar. XP terminolojisinde bu işe spike solution ismi verilir. İdeal şartlarda bu işlemin sürüm planlama oyunundan önce yapılmış olması gerekir. Buradan sürüm planlama oyunu için kullanıcı hikayelerinin oyun öncesi hazırlanmış olması gerektiği sonucunu çıkartabiliriz.

Programcılar yaptıkları denemeler sonunda kullanıcı hikayesi için tahmin verebilecek duruma gelirler.

Müşteri Kullanıcı Hikayesini Böler

Her sürüm birden fazla iterasyondan oluşur. Her iterasyon bir ile iki haftalık zaman dilimini kapsar. Seçilen kullanıcı hikayelerinin bir iterasyon bünyesinde implemente edilebilir büyüklükte olması gerekmektedir. Aksi taktirde

implementasyon bir iterasyon bünyesinde yapılamaz. Bu istenmeyen bir durumdur. Bu durumda müşteriden kullanıcı hikayesini iki ya da daha fazla kullanıcı hikayesine bölmesi istenir. Müşteri kullanıcı hikayesini bölerek, yeni oluşan kullanıcı hikayelerini tekrar tahmin edilmek üzere programcılara verir.

Müşteri Kullanıcı Hikayelerinin Sırasını Belirler

Programcılar tarafından her kullanıcı hikayesi için tahminler yapıldıktan sonra, müşteri kullanıcı hikayelerini program açısından sahip oldukları değerlere göre sıraya koyar.

Tablo 2.1: Kullanıcı hikayeleri ve değerlilik sırası

Değer	Kullanıcı Hikayesi
Yüksek	A (3) B (2) C (1)
Orta	D (3) E (3) F(2)
Düşük	G (2) F (1) H (1)

Tablo 2.1 de müşteri tarafından gruplanan kullanıcı hikayeleri yer almaktadır. Her kullanıcı hikayesi için yapılan tahmin parantez içinde yer almaktadır.

İterasyon Süresi Belirlenir

Her sürüm birden fazla iterasyondan oluşur. Bir iterasyon bir ile iki haftalık bir zaman dilimini kapsar. İterasyon sonunda test edilmiş ve çalışır bir sistem müşteriye kullanılmak üzere sunulur.

İterasyonun uzunluğu bellidir. Bu bir hafta, iki hafta ya da dört hafta olabilir. İterasyon için seçilen kullanıcı hikayelerinin bu iterasyon süresinde implemente edilebilir olmaları gerekmektedir. İmplementasyonu bitirebilmek için iterasyon süresi dinamik olarak değiştirilmez. Eğer yetişmeyen kullanıcı hikayeleri varsa, bunlar yarım bırakılır ya da hiç başlanmadan bir sonraki iterasyona devredilir. Bir iterasyonda ne kadar kullanıcı hikayesinin implemente edilebileceği

programcıların çalışma hızına (velocity) bağlıdır.

Programcı Çalışma Hızını Bildirir

Programcıların bir iterasyon (1 ile 2 haftalık süreç) bünyesinde implemente edebildikleri kullanıcı hikayelerinin hikaye puan toplamı, ekibin o iterasyondaki hızıdır. (velocity). Örneğin ekip bir iterasyonda 3 kullanıcı hikayesini implemente etmiş ve bu kullanıcı hikayelerinin hikaye puan toplamı 5 ise (2+2+1), iterasyon çalışma hızı 5 dir. Sürüm planı oluşturulabilmesi için bu hızın bilinmesi gerekmektedir. Her iterasyon süresi belli olduğu için bu iterasyon süresini aşacak şekilde kullanıcı hikaye seçimi yapılamaz. Eğer bir önceki iterasyon çalışma hızı 5 hikaye puanına eşitse, bir sonraki iterasyon için 5 hikaye puanına eşit değerde kullanıcı hikayesi seçilir. Eğer ilk iterasyon için planlama yapılıyorsa, o zaman programcıların çalışma temposunu tahmin etmeleri gerekmektedir. Her yeni iterasyonla bu değer daha doğru bir hal alacaktır.

Müşteri Sürümün Kapsamını Belirler

Sürüm planlama oyununun son basamağında müşteri sürüm içinde implemente edilecek olan hikayeleri seçer.

Tablo 2.2: Sürüm Planı

İterasyon	Hikaye Puanı	Kullanıcı Hikayesi
1	3	A
	2	B
2	1	C
	3	D
	1	H
3	3	E
	2	F
4	2	G
	1	F

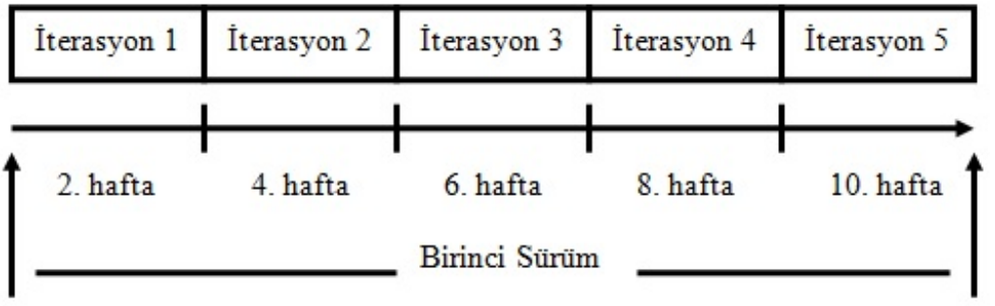
Sürüm planı projenin başlangıcında yapılan ve bir daha değişmeyen bir plan değildir. Müşteri herhangi bir iterasyonda yeni bir kullanıcı hikayesinin implementasyonunu talep edebilir. Bu durumda sürüm planının

güncelleştirilmesi gerekebilir. Sürüm planının düzenli aralıklarla kontrol edilmesi gerekmektedir. Özellikle her iterasyondan sonra programcıların çalışma hızı (velocity) tekrar hesaplanabilir olduğundan, bir sonraki iterasyon planı bu değer göz önünde bulundurularak tekrar gözden geçirilir.

İterasyon Planlaması (Iteration Planning)

Sürüm planı projenin uzun vadeli planlamasını ihtiva eder. Daha kısa zaman birimlerini daha detaylı planlayabilmek için iterasyon planları oluşturulur.

Yazılım sistemini tek solukta implemente etmek yerine XP projelerinde yapılacak iş belirli uzunlukta olan iterasyonlara bölünür. Her iterasyon iki ile dört haftalık zaman dilimlerini kapsar. İterasyon süresi sabittir. Eğer iterasyon bünyesinde seçilmiş olan kullanıcı hikayeleri implemente edilemezse, iterasyon süresi değiştirilmez. Bunun yerine implemente edilemeyen kullanıcı hikayeleri bir sonraki iterasyona devredilir.

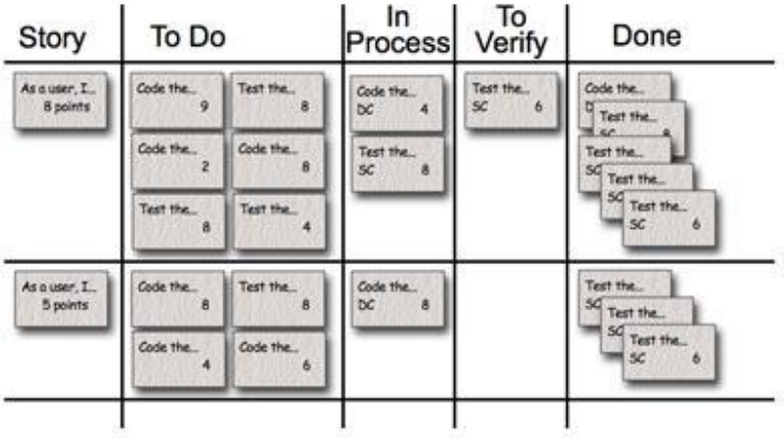


Resim 2.6 Sürüm ve iterasyon

İterasyon planını oluşturmak amacıyla iterasyon öncesi iterasyon plan toplantıları düzenlenir. Bu toplantılar bir ile dört saat arasında bir zaman diliminde gerçekleşir. Bu toplantılarda, iterasyon bünyesinde implemente edilmek üzere seçilmiş olan kullanıcı hikayeleri tekrar gözden geçirilir. Diğer iterasyonlarda edinilen tecrübeler doğrultusunda tahminler tekrar gözden geçirilir. İterasyon plan toplantılarına müşteri, programcılar, testçiler, tasarımcılar ve yazılım sisteminin oluşumunda sorumlu herkes katılır. Müşteri kullanıcı hikayelerinin sırasını tekrar gözden geçirebilir, istediği kullanıcı hikayesini iterasyondan çıkartıp, yeni bir kullanıcı hikayesini iterasyon planına ekleyebilir.

Resim 2.7 İterasyon plan toplantısı

Programcılar kullanıcı hikayelerini gözden geçirerek görev (task) listesi oluştururlar. Bu görevler görev kartlarına (task card) yazılır. Sürüm planında olduğu gibi görevlerin implementasyon süresi poker kartları kullanılarak programcılar tarafından tahmin edilir. Tahminler saat bazında yapılır. Görev kartları ait oldukları kullanıcı hikayesinin yer aldığı hikaye kartıyla gruplandırılır (resim 2.8).



Resim 2.8 Görev kartları kullanıcı hikaye bazında gruplanır

Tespit edilen görevler doğrudan programcılara atanmaz. Programcılar iterasyon başlamadan önce üzerinde çalışacakları ilişkili bir ya da iki görev kartı seçerler.

Tavsiye

İterasyon planlaması yapılırken kullanıcı hikayesinden yola çıkarak görevler yerine, kullanıcı hikayesinin implementasyonunu teyit eden onay/kabul test listesi oluşturulmalı ve implementasyon bu testlerle start almalıdır, yani görev kartları (task card) yerine test kartları oluşturulmalıdır. Görev listesine bakılarak yazılım sisteminde ne oranda bir gelişme olduğunu anlamak çoğu zaman mümkün değildir. Örneğin aşağıdaki görev listesinde yer alan üç numaralı göreve bir göz atınız. Sizce bu görev tamamlandığında ne orada bir ilerlemenin kaydedildiğini ifade ediyor? Oluşturulan görevler bazen implementasyonun gerçek amacının gözden kaçırılmasına sebep olabilirler. Görevler bize ne yapılması gerektiğini söylerler, ama somut olarak hangi adımların atılması gerektiğini söylemezler.

Görev listesi:

1. LoginController sınıfı işletme (business) katmanında bulunan LoginManager sınıfını kullanarak login işlemini gerçekleştirir. LoginManager sınıfını oluştur.
2. LoginManager sınıfı LoginDao üzerinden veri tabanı işlemlerini gerçekleştirir. LoginDao isminde bir sınıf oluştur.
3. Üye bilgileri veri tabanında customer isimli tabloda tutulur. Bu tablo yeterli mi, kontrol et.

Onay/kabul test listesi:

1. Kullanıcı login sayfasına gider. E-posta adresi ve şifre alanlarını boş bırakarak login butonuna tıklar. Kullanıcıya “Lütfen e-posta adresinizi ve şifrenizi giriniz!” hata mesajı gösterilir.
2. Kullanıcı login sayfasına gider. E-posta adresini girer ve şifre alanını boş bırakarak login butonuna tıklar. Kullanıcıya “Lütfen şifrenizi giriniz!” hata mesajı gösterilir.
3. Kullanıcı login sayfasına gider. E-posta adres alanını boş bırakarak, şifresini girer ve login butonuna tıklar. Kullanıcıya “Lütfen E-posta adresinizi giriniz!” hata mesajı gösterilir.
4. Kullanıcı login sayfasına gider. E-posta adresi ve şifreni girer ve login butonuna tıklar. E-posta adresi ve şifre doğrudur. Login işlemi gerçekleşir. Üye, hoş geldiniz sayfasına yönlendirilir.
5. Kullanıcı login sayfasına gider. E-posta adresi ve şifreni girer ve login butonuna tıklar. E-posta adresi geçersizdir. Kullanıcıya “ E-posta adresiniz geçersizdir, lütfen tekrar ediniz!” hata mesajı gösterilir.
6. Kullanıcı login sayfasına gider. E-posta adresi ve şifreni girer ve login

butonuna tıklar. E-posta adresi geçerli olmasına rağmen şifre hatalıdır. Kullanıcıya "Şifre hatalı, lütfen tekrar deneyiniz!" hata mesajı gösterilir.

Onay/kabul testleri görevlere kıyasla daha temsili ve somut yapıdadır. Onlar neyin nasıl yapılması gerektiği hakkında bilgi ihtiva ederler ve böylece programcının işini kolaylaştırırlar. Onay/kabul test listesinde bulunan birinci teste baktığınızda ne yapmanız gerektiğini hemen anladınız değil mi? Onay/kabul testleri yazılım sistemindeki özelliklerle birebir örtüşür ve bu yüzden ilerlemenin ölçülebilir olmasını sağlar. Ben bu yüzden iterasyon planlamasında görev yerine onay/kabul testleri oluşturulmasını ve implementasyonunu diğer bölümlerde yakından göreceğimiz gibi top down TDD tarzı yapılmasını tavsiye ediyorum.

Sürüm/İterasyon Planı Nerede?

XP projelerinde sürüm ve iterasyon planları tüm proje çalışanları tarafından görülebilecek bir yerdedir. Çoğu zaman duvar panoları ve beyaz tahtalar kullanılarak kullanıcı hikayeleri ait oldukları sürüm ve iterasyon belli olacak şekilde gruplanır. Resim 2.9 de bunun bir örneği yer almaktadır.

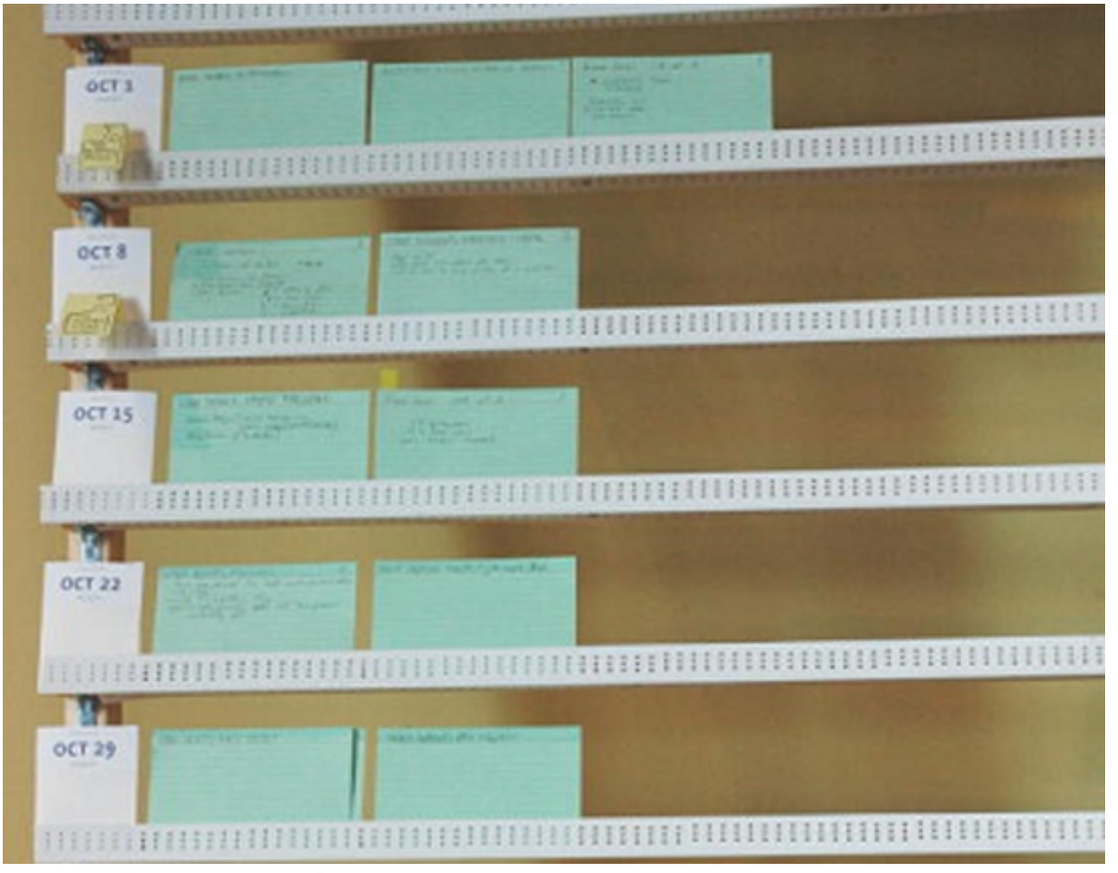
Resim 2.9 Sürüm ve iterasyon planını ihtiva eden pano

Panonun ilk üç katında tamamlanmış iterasyonlar ve ihtiva ettikleri kullanıcı hikayeleri yer almaktadır. Tamamlanan iterasyonlar, iterasyonun başlangıç

tarihini ve implementasyon için kullanılan zamanını (5 = 5 gün) ihtiva eden beyaz kartlarla birbirini takip etmektedir.

Resim 2.10 Sürüm ve iterasyon planını ihtiva eden pano

Panonun 4-12 katları planlanan 9 iterasyonu ihtiva eder. Her iterasyon 1 hafta olduğu için panonun bu katlarında 9 haftalık çalışma süresi planlanmıştır. 4. kat güncel iterasyondur. Her katın sol bölümünde o iterasyonda implemente edilecek olan kullanıcı hikayeleri yer alır. Her katın sağ bölümünde önemlilik derecesi düşük olan ve henüz iterasyon planlarında yer almamış kullanıcı hikayeleri yer alır.



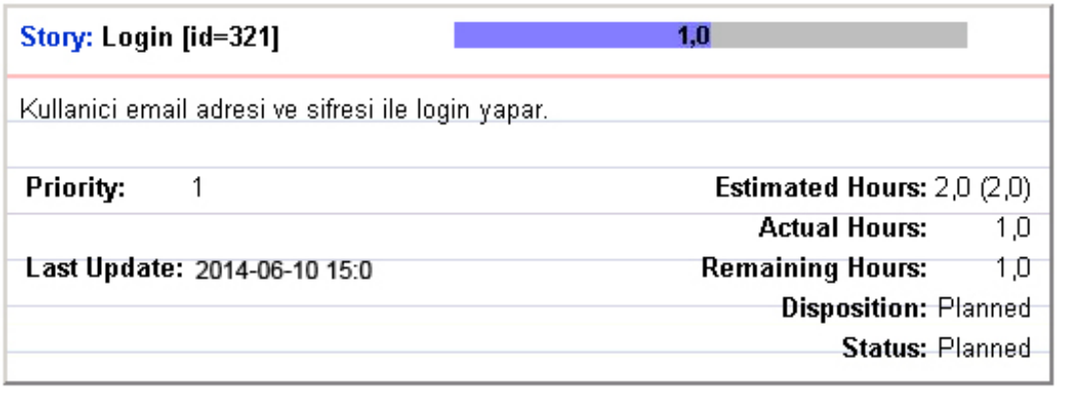
Resim 2.11 Sürüm ve iterasyon planını ihtiva eden pano

Resimlerde görüldüğü gibi hikaye kartları çok basit araçlar kullanılarak oluşturulabilirler. Hikaye kartlarının ana amacı programcı ekip ve müşteri arasındaki diyaloğu kuvvetlendirmek ve projede kaydedilen ilerlemeyi görsel olarak sağlamaktır. Bir bilgisayar programı kullanılarak ta hikaye kartları oluşturulabilir. Lakin bu yöntem hikaye kartları kadar çevik olmayacaktır, çünkü bir bilgisayarı açıp, hikaye kartlarına ulaşılana kadar az denilmeyecek bir zaman geçebilir. Oysa hikaye kartları bir pano üzerinde herkesin görebileceği şekilde dizilebilir ve kullanılabilirler. Değişik lokasyonlarda beraber çalışmak zorunda olan ekipler için dijital hikaye kart yönetim sistemleri düşünülebilir.

Dijital Hikaye Kartları

Hikaye kartları dijital ortamda da oluşturulabilir. Bu işlem için açık kaynaklı olan XPlanner programı kullanılabilir. XPlanner web tabanlı proje yönetim ve takip programıdır. XPlanner ile iterasyonlar ve ihtiva ettikleri dijital hikaye kartları oluşturulur. Her dijital kart bünyesinde görev (task) listesi oluşturulabilir. Ayrıca bu kartlar üzerine birim ve onay/kabul testleri not edilebilir. Böyle bir XPlanner dijital hikaye kartını bir sonraki resimde

görmekteyiz.



Resim 2.12 Dijital hikaye kartı

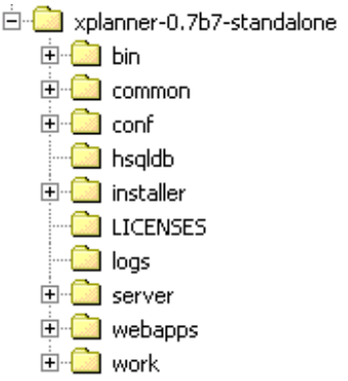
Sol üst köşede kullanıcı hikayesinin ismi (Story: Login) ve numarası (id=320) bulunmaktadır. Kartın ilk satırında kullanıcı hikayesi hakkında açıklama yer almaktadır: "Bir sistem kullanıcısı sahip olduğu e-posta adresi ve şifresi ile sisteme login yapabilir". Kullanıcı hikayesinin öncelik sırası müşteri tarafından 1 (Priority:1) olarak tanımlanmıştır. Hikaye kartının üzerinde bulunan diğer bölümler sırasıyla şöyledir:

- ***Estimated Hours*** - Bu Login isimli kullanıcı hikayesi için tahmin edilen implementasyon süresidir. Burada programcı ekip 2 saat implementasyon süresi belirlemiştir.
- ***Actual Hours*** - İmplementasyon için programcı ekip tarafından harcanan zamandır. Programcı ekip bu kullanıcı hikayesi için 1 saatlik bir çalışma yapmıştır.
- ***Remaining Hours*** - İmplementasyon için geriye kalan zaman dilimidir. Programcı ekip kalan 1 saat içinde bu kullanıcı hikayesini implemente etmelidir.
- ***Disposition*** - Burada Planned, Carried Over ve Added statüleri yer alabilir. Planned kullanıcı hikayesinin iterasyona, iterasyona başlanmadan önce eklenmiştir anlamına gelmektedir. Carried Over kullanıcı hikayesinin bir önce iterasyondan alındığını ve Added statüsü kullanıcı hikayesinin bu iterasyona başlandıktan sonra eklendiğini gösterir.
- ***Status*** - Kullanıcı hikayesinin belli aşamalardaki statüsünü gösterir. Burada Draft, Defined, Estimated, Planned, Implemented, Verified ve Accepted gibi bir değerler yer alabilir. Kullanıcı hikayesi ilk oluşturulduğunda Draft statüsündedir. Programcı ekip tarafından implementasyon tahminleri yapıldığı zaman Estimated statüsüne geçer. Kullanıcı hikayesi bir iterasyon içinde eklendiğinde Planned, programcı ekip

tarafından implemente edildikten sonra Implemented, birim testlerinin doğru çalışması sonucunda Verified ve onay/kabul testleri olumlu sonuç verdiğinde Accepted statüsüne geçer. Her kullanıcı hikayesinin ulaşması gerektiği en son statüs Accepted olmalıdır. Bu yüzden her kullanıcı hikayesi için en az bir adet onay/kabul testinin hazırlanması gerekmektedir.

XPlanner

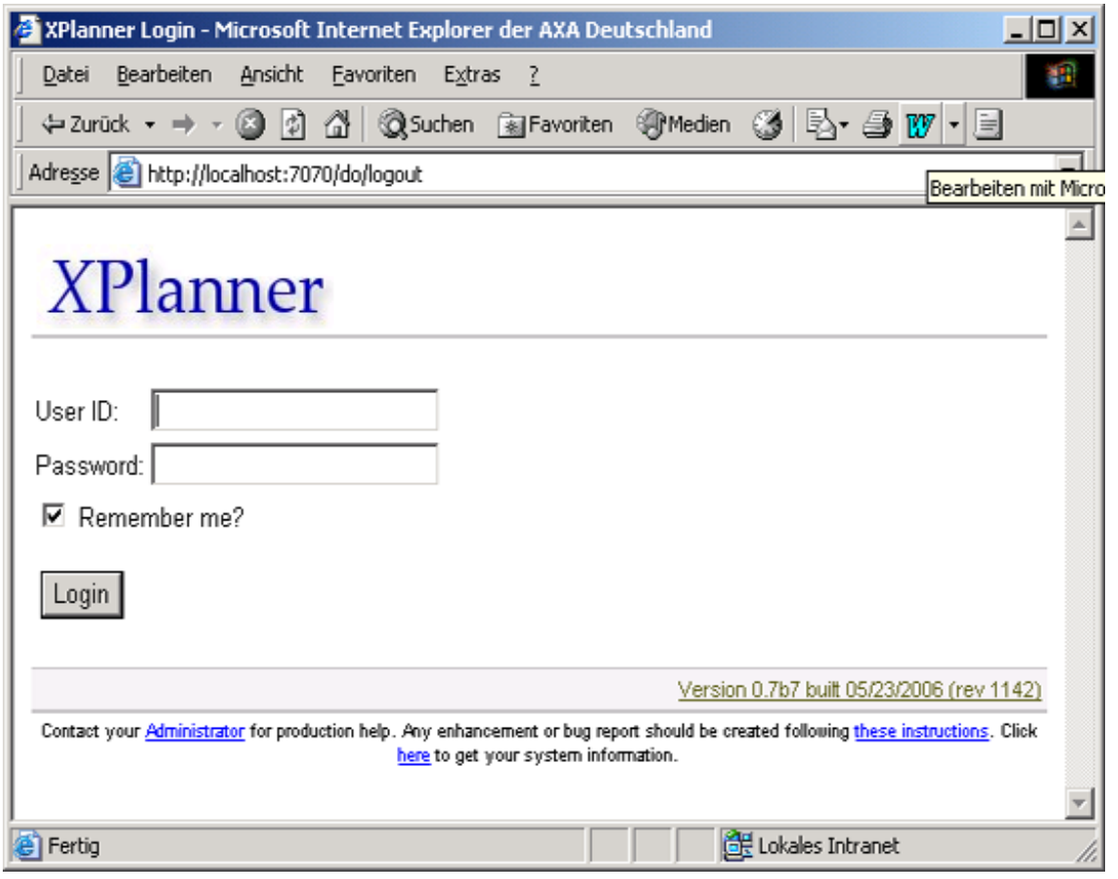
Xplanner ın güncel sürümünü bu adresten temin edebilirsiniz. Hızlı kurulum ve kullanım için Standalone (xplanner-0.7b7-standalone.zip) versiyonunu indirmenizi tavsiye ederim. Bu versiyon içinde Tomcat uygulama sunucusu ve HSQL veri tabanı bulunmaktadır. Paketi herhangi bir dizine açtıktan sonra, aşağıdaki dizin yapısı oluşacaktır:



Resim 2.13 Xplanner dizin yapısı

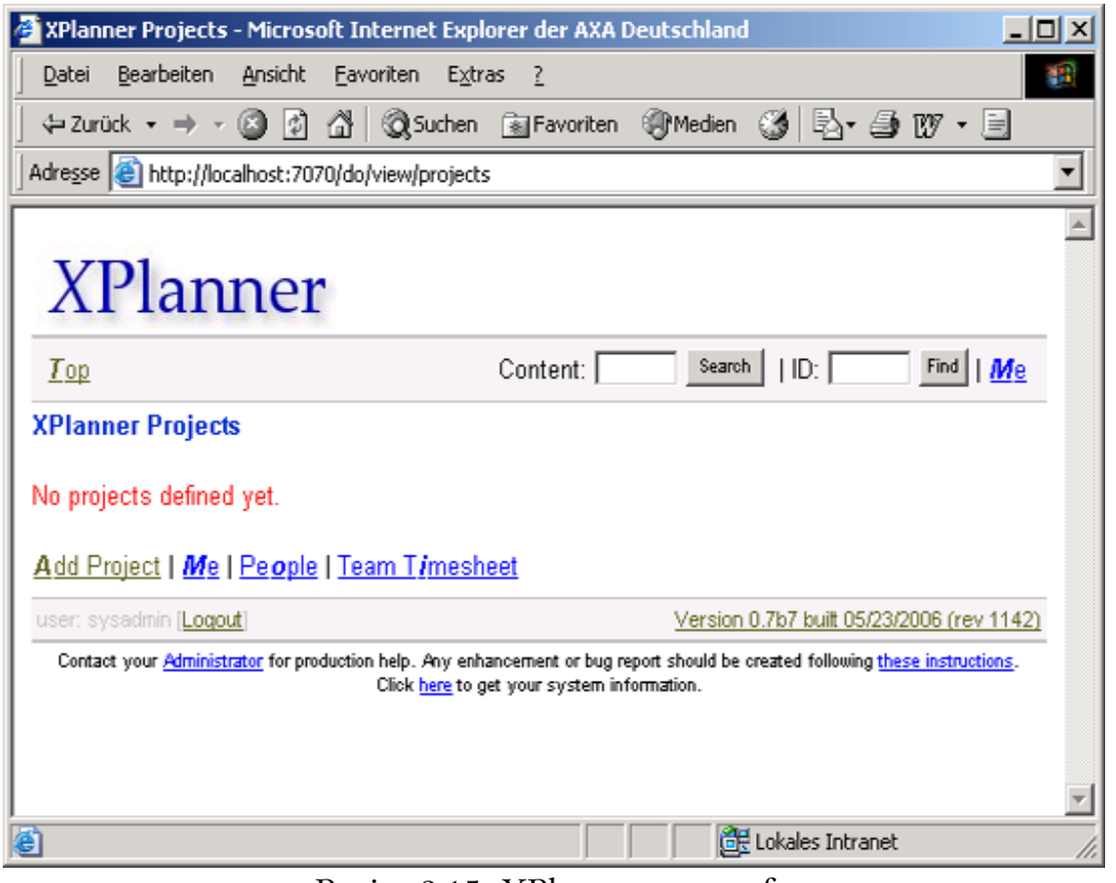
Bin dizininde yer alan startup.bat ile XPlanner çalıştırılır. Veri tabanı olarak MySQL ya da HSQL kullanılabilir. JDK 1.4 ve üstü gerekmektedir. XPlanner standalone sürümü HSQL ile çalışacak şekilde konfigüre edilmiştir. Bu yüzden startup.bat ile kurulu bir XPlanner sürümü çalıştırılabilir.

startup.bat ile lokal Tomcat çalıştırıldıktan sonra http://localhost:7070 adresinden XPlanner ın web tabanlı arayüzüne bağlanabiliriz.



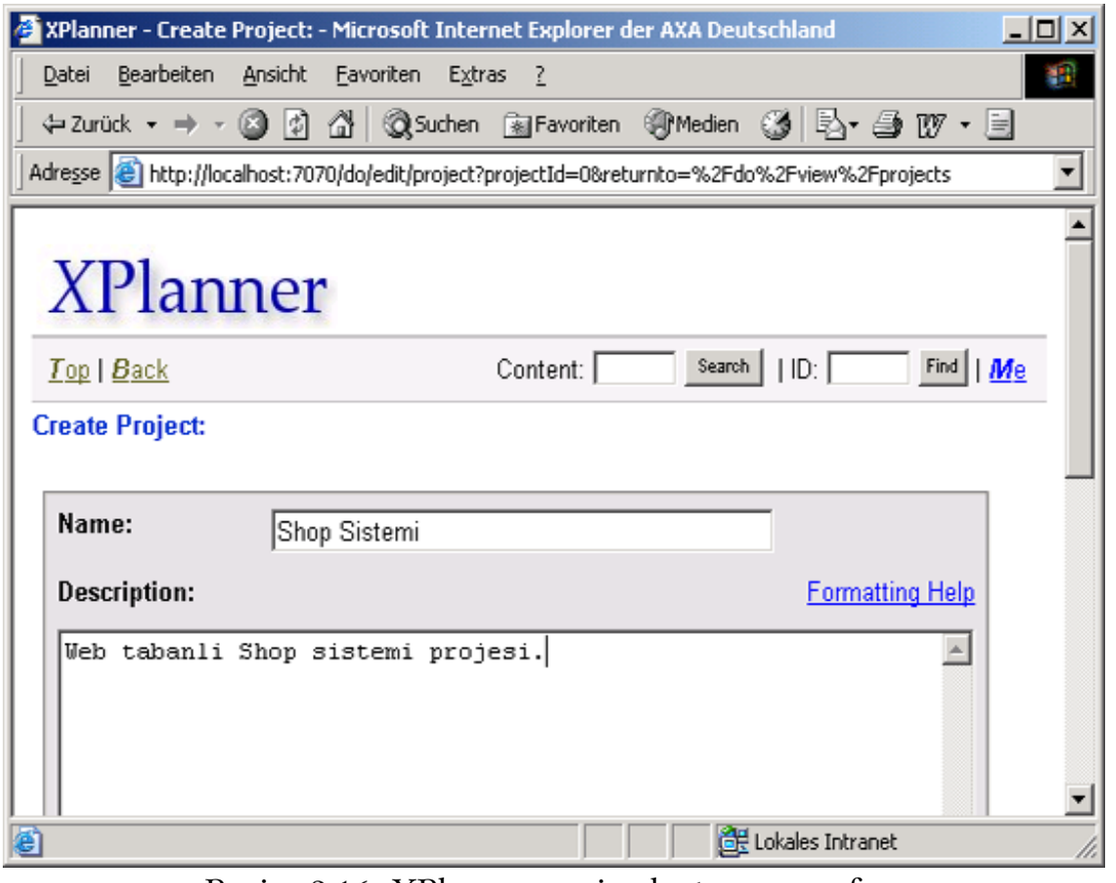
Resim 2.14 XPlanner login sayfası

sysadmin/admin kullanıcı ismi ve şifresi ile login yapılabilir. Login işleminin ardından yönetilen projelerin yer aldığı ilk sayfa görüntülenecektir. Henüz bir proje tanımlamadığımız için bu sayfada “No projects defined yet (henüz proje tanımlanmadı)” mesajı yer almaktadır.



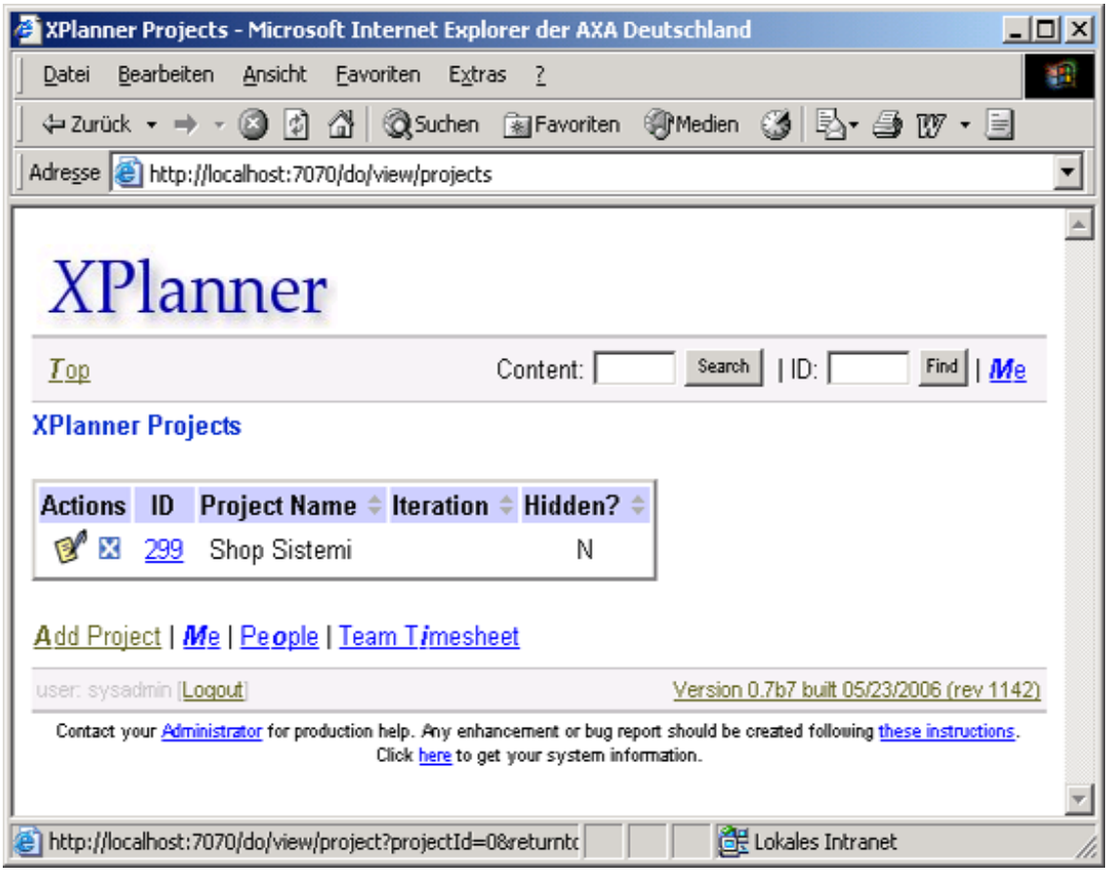
Resim 2.15 XPlanner anasayfa

Add Project linkine tıklayarak projemizi ekleyebiliriz.



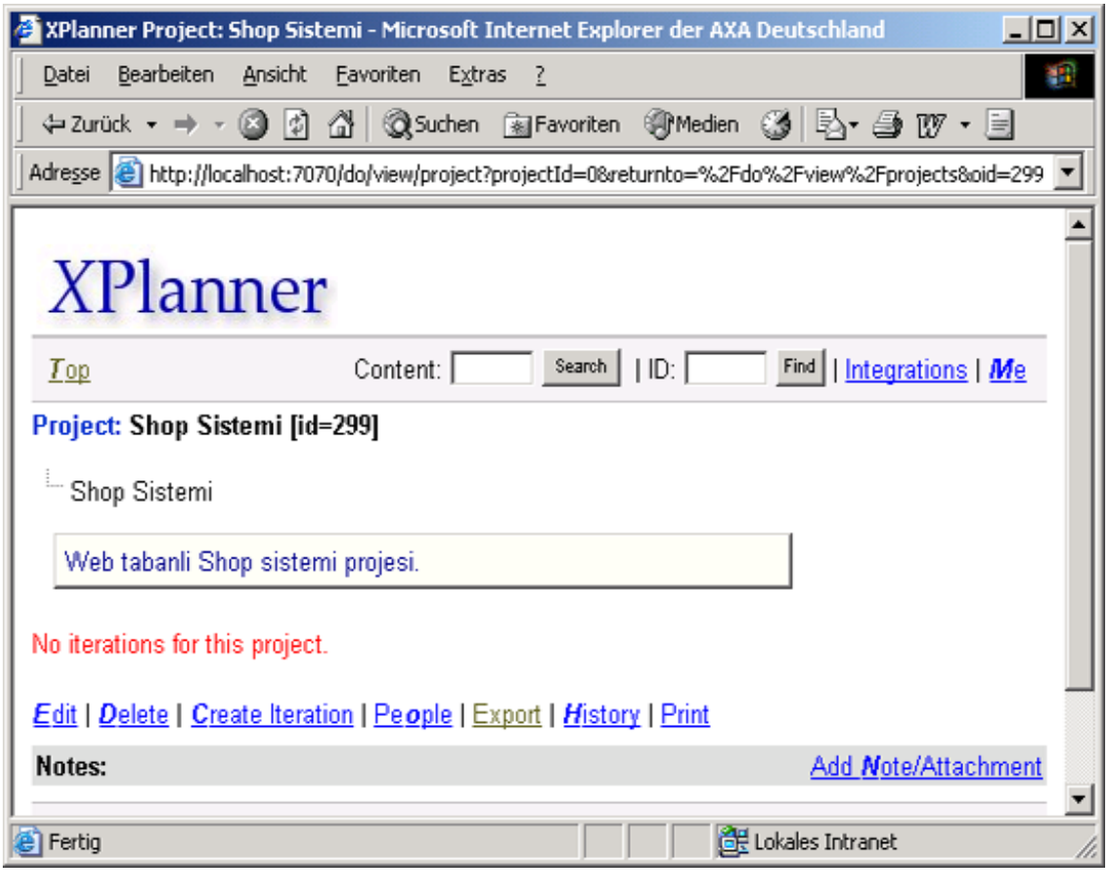
Resim 2.16 XPlanner proje oluşturma sayfası

Bu işlemin ardından Shop Sistemi isminde bir proje oluşturulacaktır. Proje aşağıdaki şekilde proje ana sayfasında görüntülenir.



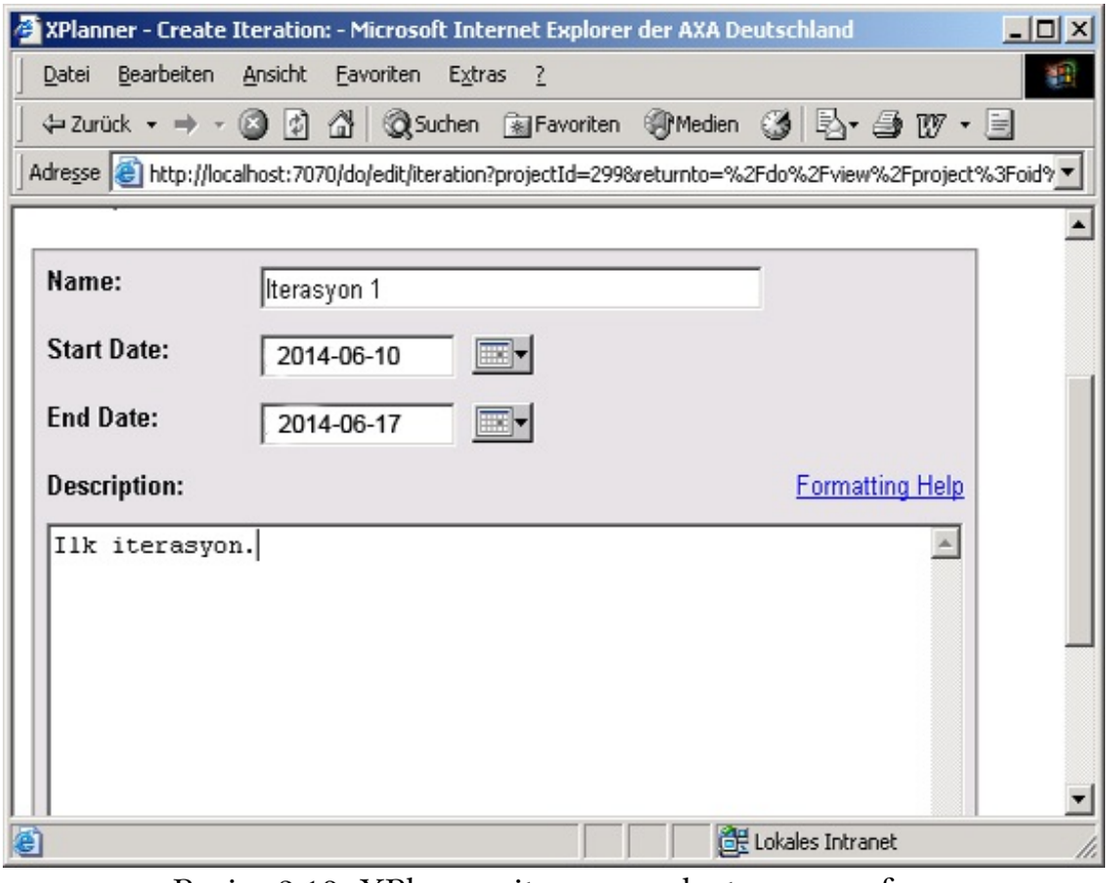
Resim 2.17 XPlanner anasayfa

XPlanner Shop Sistemi isminde bir proje oluşturmuş ve bu projeye 299 numarasını atamıştır. 299 rakamı üzerindeki linke tıklayarak proje detaylarını görebiliriz.



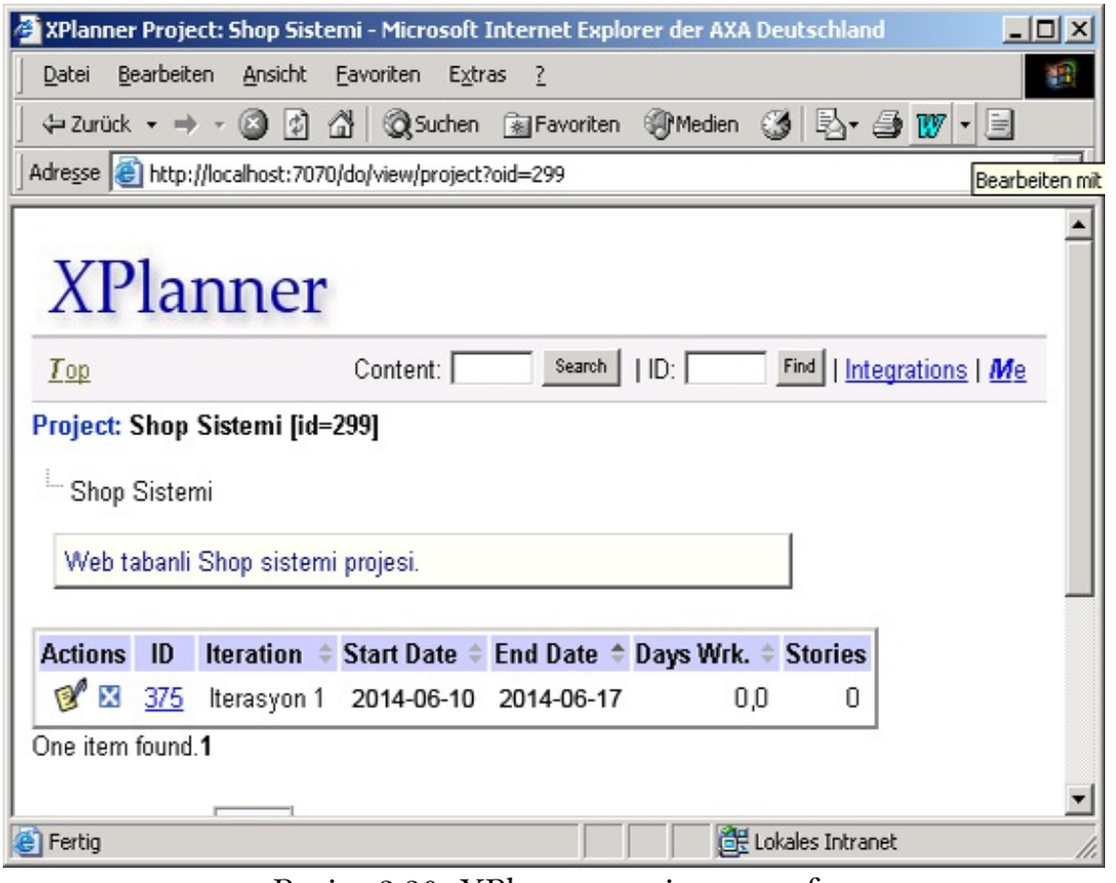
Resim 2.18 XPlanner proje detay sayfası

Her proje için bir veya birden fazla iterasyon oluşturulması gerekmektedir. Her iterasyon implemente edilecek kullanıcı hikayelerini, yani hikaye kartlarını ihtiva eder. Bir iterasyon oluşturmak için Create Iteration linkine tıklıyoruz.



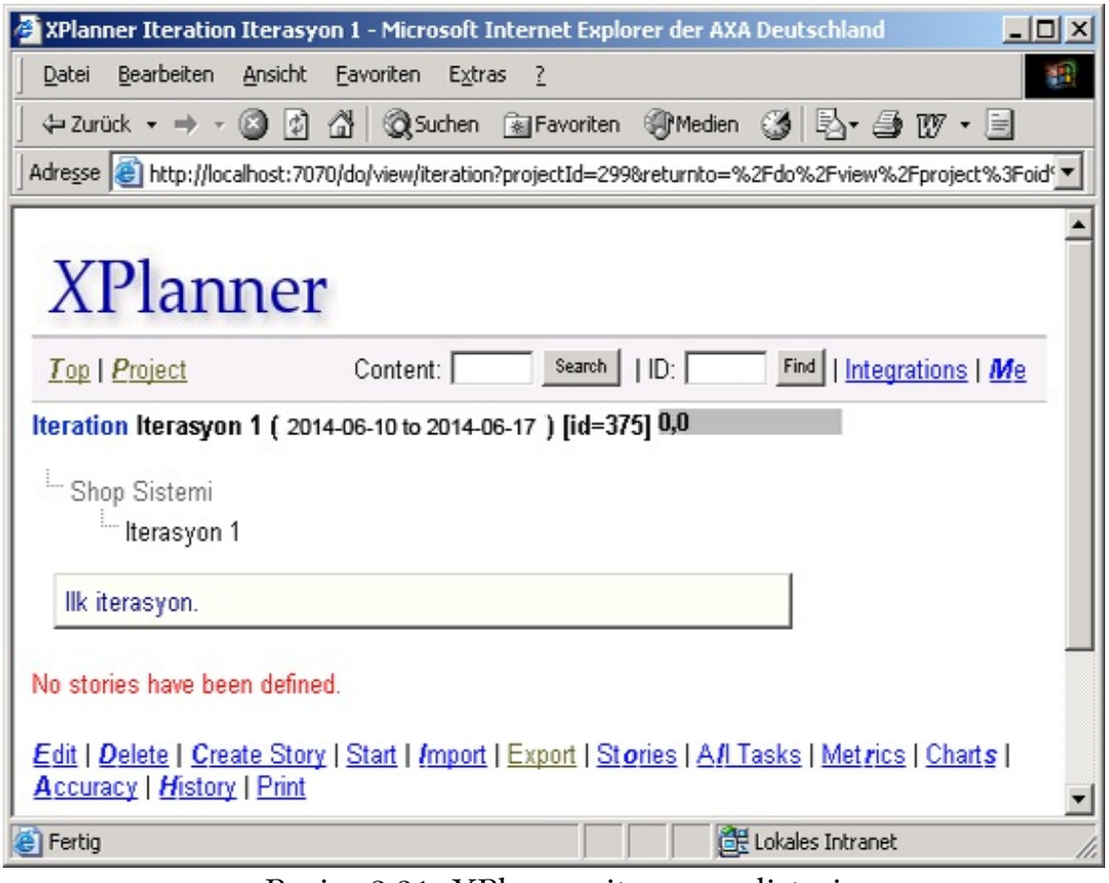
Resim 2.19 XPlanner iterasyon oluşturma sayfası

İlk iterasyonumuzu İterasyon 1 olarak isimlendiriyoruz. Her iterasyonun bir başlangıç ve bitiş tarihi vardır. Başlangıç tarihi olarak 10.6.2014, bitiş tarihi olarak 17.6.2014 seçiyoruz. Buradan da anlaşılacağı gibi iterasyon sürece 1 hafta olarak seçilmiştir.



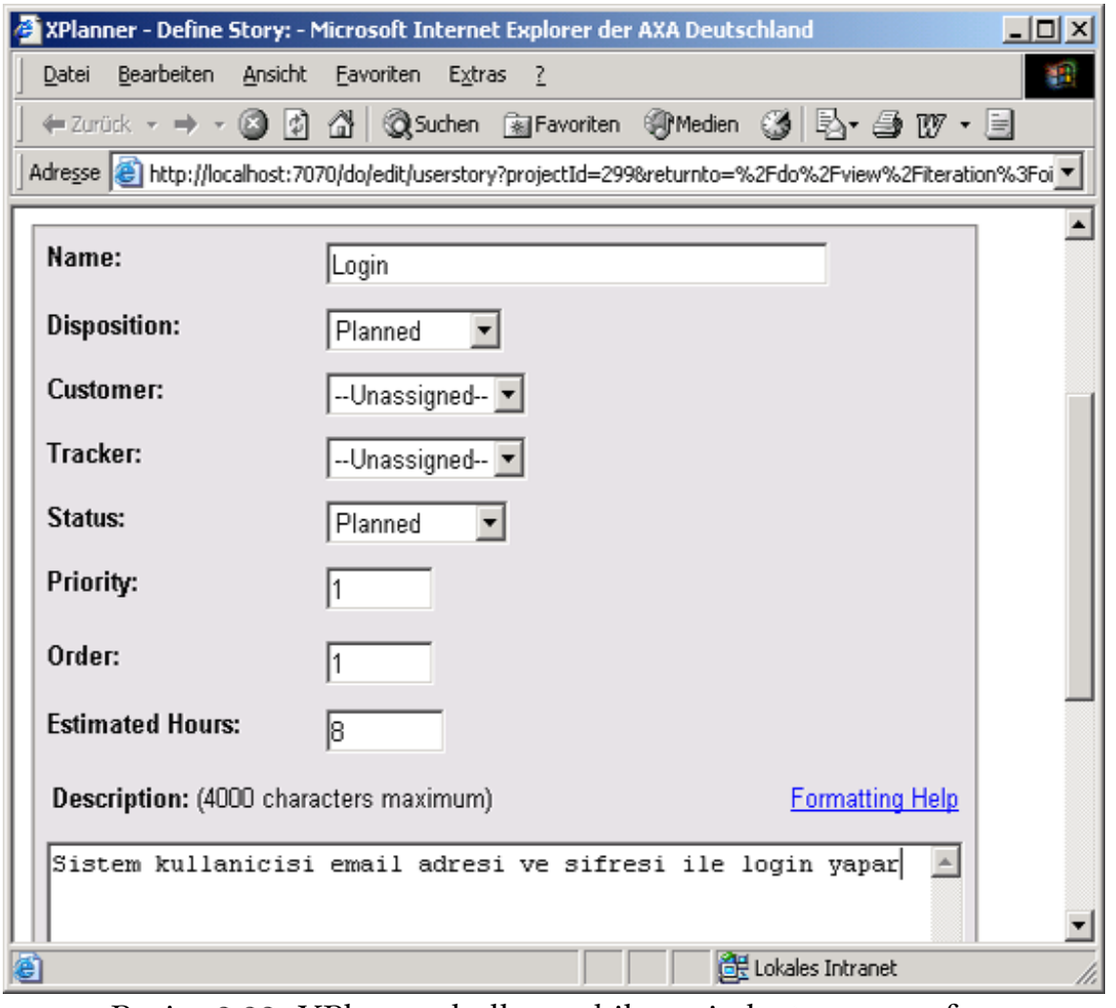
Resim 2.20 XPlanner proje anasayfa

Bu işlemin ardından XPlanner 375 numaralı ve İterasyon 1 ismini taşıyan ilk iterasyonu oluşturur. Şimdi bu iterasyon bünyesinde implementasyonu yapılacak olan kullanıcı hikayelerini oluşturmamız gerekiyor. 375 rakamı üzerindeki linke tıklayarak, iterasyon bölümüne geçiyoruz.



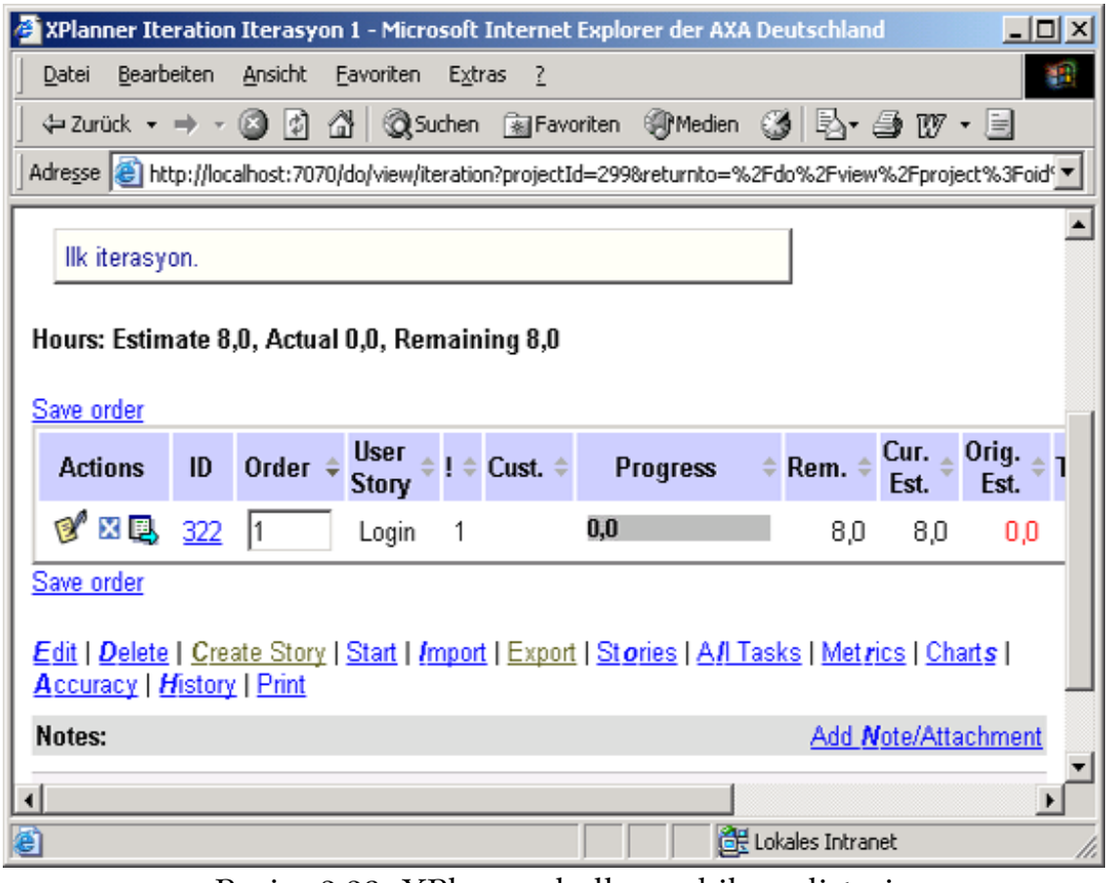
Resim 2.21 XPlanner iterasyon listesi

Bu iterasyon bünyesinde henüz kullanıcı hikayesi tanımlanmadığı için "No stories have been defined (kullanıcı hikayesi tanımlanmadı)" mesajı yer almaktadır. Create story linkine tıklayarak, ilk kullanıcı hikayesini oluşturuyoruz.



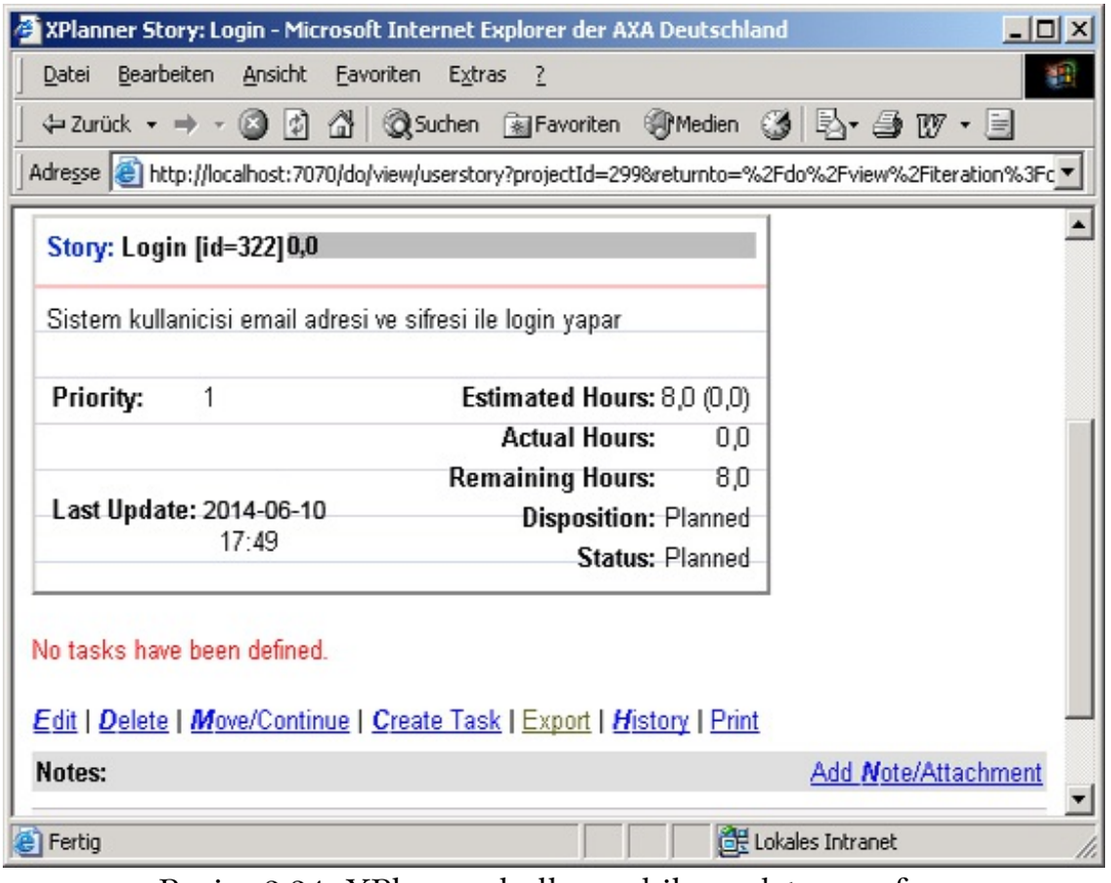
Resim 2.22 XPlanner kullanıcı hikayesi oluşturma sayfası

Bu işlemin ardından XPlanner ilk kullanıcı hikayesini aşağıdaki şekilde oluşturur.



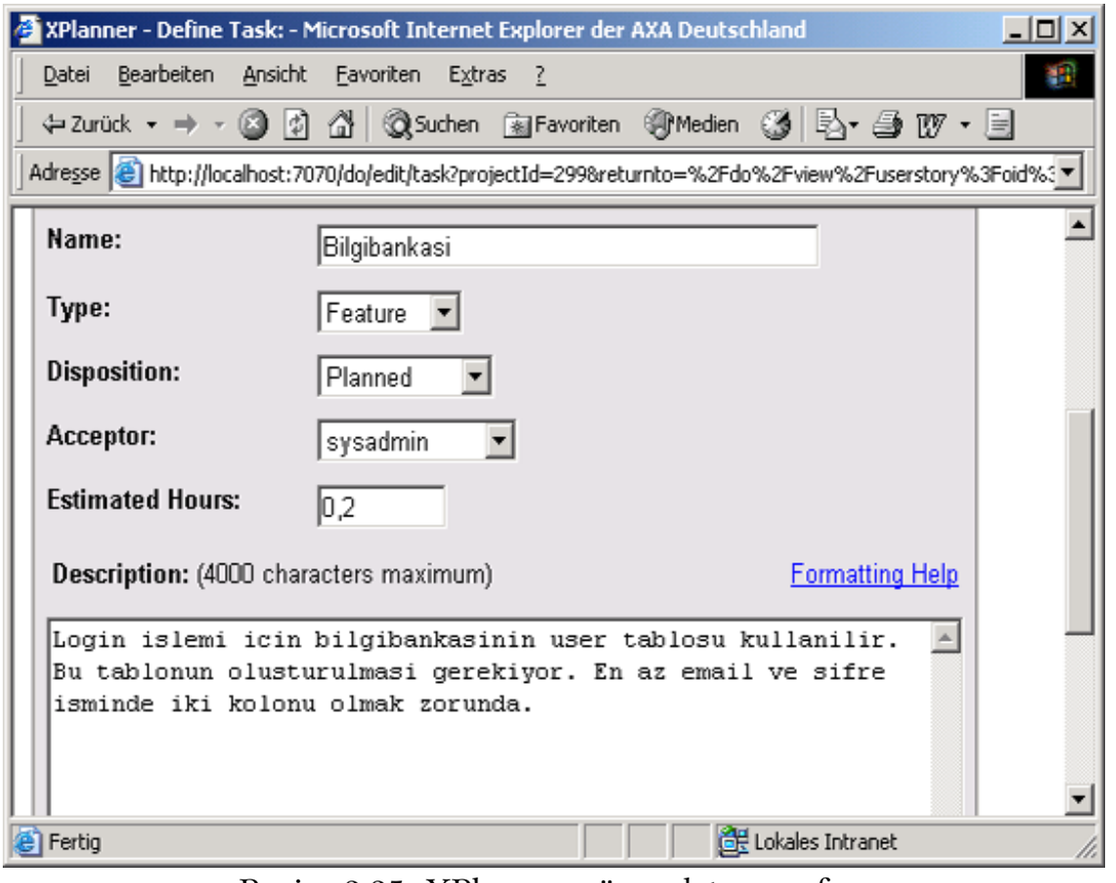
Resim 2.23 XPlanner kullanıcı hikaye listesi

Kullanıcı hikayesi 322 rakamını taşımaktadır. Bu rakamın üzerinde bulunan linke tıklayarak, kullanıcı hikayesinin detaylarını görebiliriz.

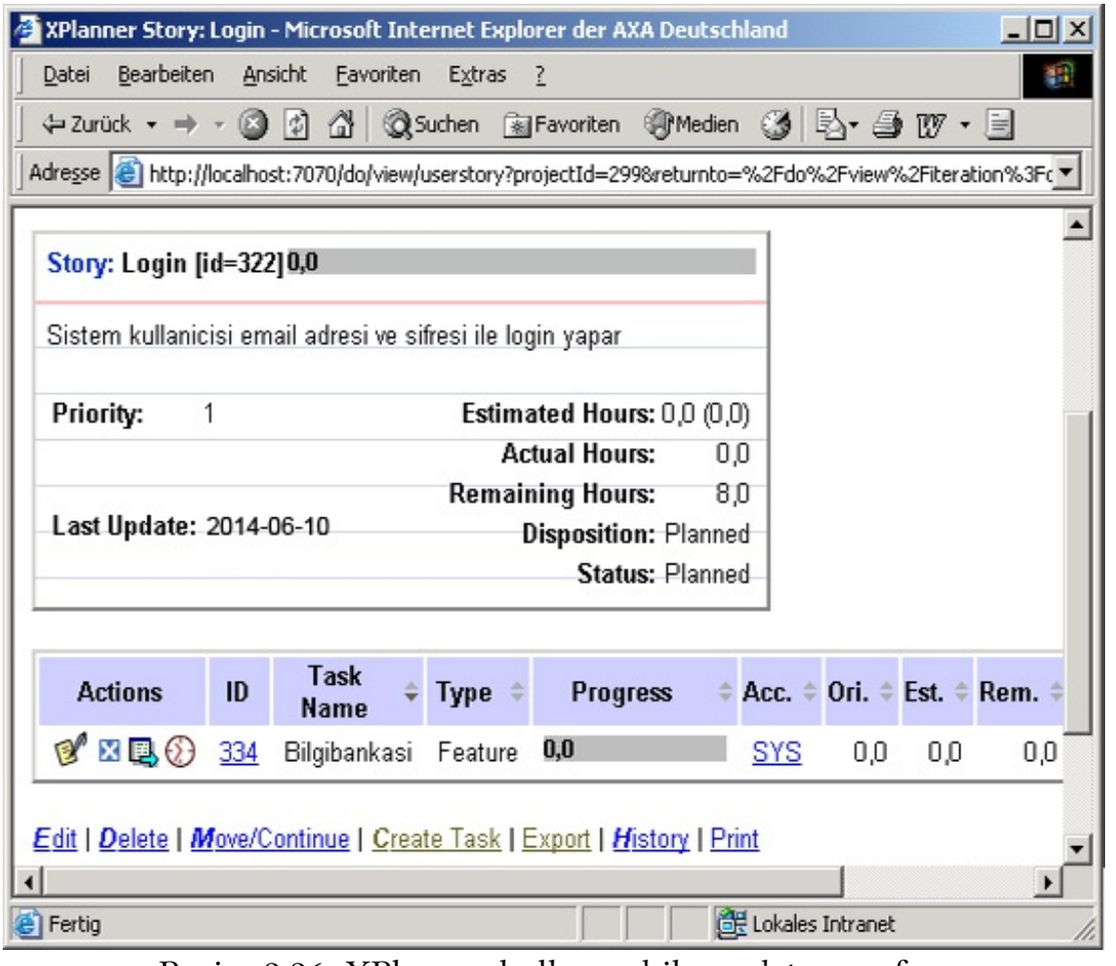


Resim 2.24 XPlanner kullanıcı hikaye detay sayfası

Burada 322 rakamını taşıyan bir hikaye kartı görmekteyiz. Login ismini taşıyan bu hikaye kartı, kullanıcı hikayesi için 8 saatin tahmin edildiğini ve öncelik derecesinin 1 olduğunu göstermektedir. Bu kullanıcı hikayesi bünyesinde görev listesi tanımlanmadığı için "No tasks have been defined (görev listesi tanımlanmamıştır)" mesajı görülmektedir. Programcılar kullanıcı hikayesinin implementasyon zamanını tahmin edebilmek için bu kullanıcı hikayesi için gerekli görev (task) listesi oluştururlar. Her görev için bir zaman dilimi tespit edilir. Bunların toplamı, kullanıcı hikayesinin tahmin edilen implementasyon (Estimated Hours) zamanıdır. Create Task linkine tıklayarak ilk görevi oluşturabiliriz.

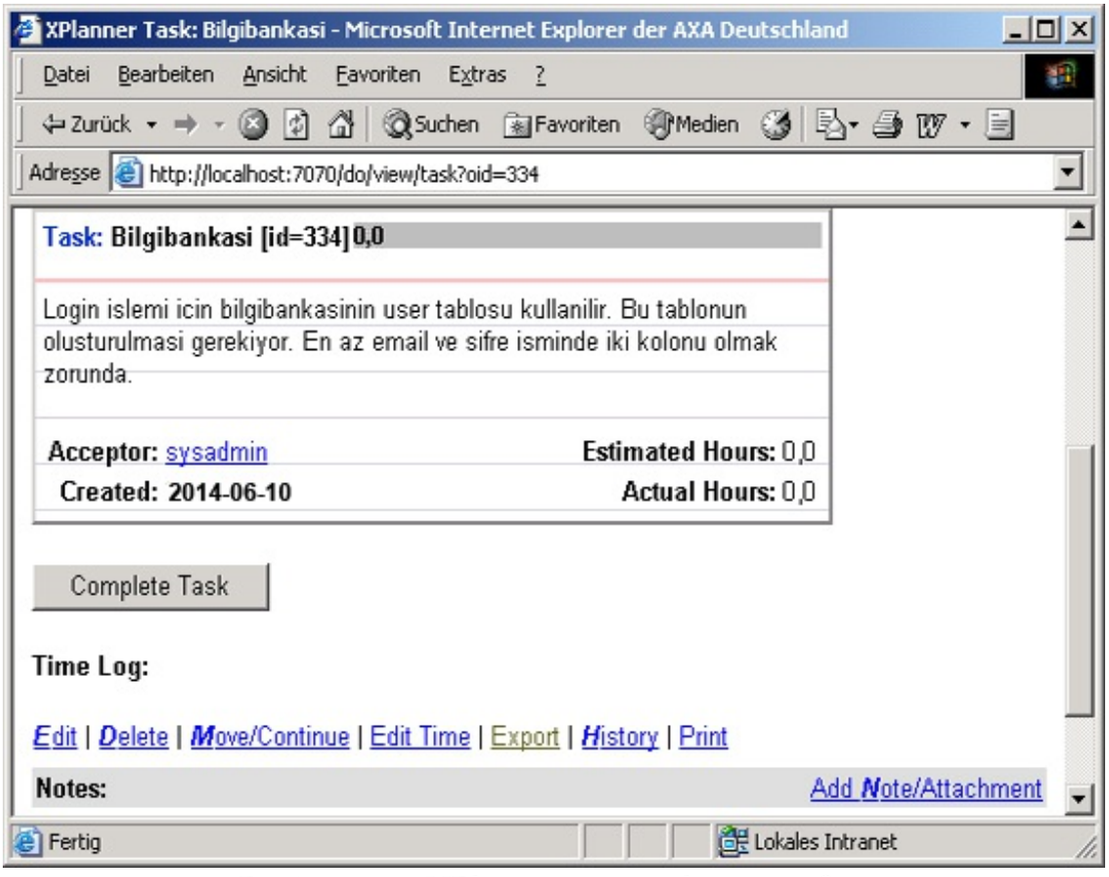


Resim 2.25 XPlanner görev detay sayfası



Resim 2.26 XPlanner kullanıcı hikaye detay sayfası

Hikaye kartına geri döndüğümüzde yeni görevin kartın alt bölümüne eklendiğini görebiliriz. Yeni göreve 334 rakamı verilmiştir. Bu linke tıklayarak görev detay bölümüne geçebiliriz.



Resim 2.27 XPlanner görev detay sayfası

Görev kartları (task card) hikaye kartlarına (story card) benzemektedir. Her görev kartının sahip olduğu başlığı (Task:Bilgibankası) ve numarası ([id=334]) bulunmaktadır. Görev kartına görev hakkında açıklama eklenir. Programcı ekip görevin hangi zaman diliminde bitirilebileceğini tahmin eder. Complete Task butonuna tıklayarak, görevin tamamlandığı belirtilir.

Gördügümüz gibi XPlanner kullanıcı hikayelerinin ve hikaye kartlarının dijital olarak oluşturulmasını kolaylaştırmaktadır. Tüm ekip bir web tarayıcısıyla projenin gidişatı hakkında bilgi sahibi olabilir. Lakin aynı odada çalışan bir ekip için XPlanner gibi bir program tavsiye edilmemektedir. Aynı odada çalışan programcı ekip kalemle yazılan hikaye kartları tercih etmelidir. Bu şekilde projenin ve güncel iterasyonun hangi safhada olduğu çok daha kolay anlaşılabilir. Programcılar karton kartlarını ellerine alarak, bilgi alışverişinde bulunabilirler ve kartlar üzerinde değişiklik yapabilirler. Bu şekilde bir çalışma sistemi, projenin daha hızlı bir şekilde gerçekleştirilmesini sağlayacaktır. Aynı odada çalışma imkanı bulamayan, değişik bina ya da şehirlerde bulunan programcı ekipler için XPlanner gibi bir sistemin kullanılması avantajlı olacaktır. Bu şekilde internet üzerinden çalışmalar koordine edilebilir.

3. Bölüm

İletişim

Giriş

Bilginin bireyler arası transferi iletişimle mümkündür. İletişim olmayan yerde bireyler izole olduklarından takım işi ve ruhu oluşmaz. Extreme Programming in temel değerlerinden birisi iletişimdir. Bu bölümde

- Stand-up toplantıların ne olduğunu ve nasıl yapıldığını,
- retrospective toplantıların ne olduğunu ve nasıl yapıldığını,
- çevik çalışma ortamlarının iletişim aspekti göz önünde bulundurularak nasıl yapılandırıldığını,
- iletişim aracı olan Wiki, Trac ve Bugzilla nın nasıl kullanıldığını

yakından inceleyeceğiz.

Stand-up Toplantı

Çevik projelerde her gün stand-up toplantı düzenlenir. 15 dakikayı geçmeyen bu toplantılarda her katılımcı şu soruların cevabını verir:

1. Dünden beri ne yaptım?
2. Ne gibi sorunlarla karşılaştım?
3. Yarına kadar ne yapıyorum?

Resim 3.1 Günlük Stand-up toplantı

Bu sayede ekip içindeki herkes takım arkadaşlarının yaptıkları hakkında bilgi sahibi olur. Bu toplantılar mesai başlangıcında gerçekleştirilir.

Bu toplantılarda teknik detaylar ve çözümler konuşulmaz, çünkü bu toplantının süresini uzatabilir. Daha önce de belirttiğim gibi toplantının süresi 15 dakikayı aşmamalıdır. Teknik detaylar toplantı sonrası küçük gruplar oluşturularak konuşulabilir. Toplantıya katılanların ayakta durması çok önemlidir, çünkü bu şekilde konuşlarını kısa tutarlar ve hemen işlerinin başına dönme isteğine sahip olurlar.

Projede ilk adımlar atılırken bir koçun XP tekniklerinin uygulanışını kontrol etmesinde fayda vardır. Çoğu zaman bir ekip XP tekniklerini uyguladığını düşünür, ama uygulanış detaylarına bakıldığında hataların yapıldığı görülür. Örneğin Stand-up toplantılarda çalışanların oturması ya da toplantının 15 dakikayı aşması yanlıştır. 15 dakika sonunda isteyen toplantıyı terk edebilmelidir. Yapılan başka bir hatada gelmeyen bir katılımcının beklenmesidir. Toplantı zamanında başlamalı ve geç gelen katılımcıya hafif bir ceza verilmelidir, örneğin her geç gelen 1 TL kumbaraya atmak zorundadır. Biriken bu paralar ile tüm takım için bir kasa kola alınır.

Stand-up toplantılarda patron ya da şef bulunmamalıdır. Eğer patron ya da şef toplantıya katıldıysa, toplantıyı yönetmeye yeltenmemelidir. Bu durumda katılımcılar sanki ona rapor verircesine gerekli soruların cevabını vereceklerdir. Toplantı esnasında katılımcının patron ya da şefe değil, gruba karşı sorumluluğu vardır ve raporu gruba verir.

Retrospective Toplantısı

Çevik projelerde düzenli aralıklarla retrospective toplantılar düzenlenir. Bir ile üç gün arası bir zaman dilimi sürebilen bu toplantılarda projeye geri bakış sağlanarak, oluşan sorunlar üzerinde tartışılır.

Geçmişe kritik yaklaşım ekip çalışanları arasında çatışmaya sebep verebilir. Bunu önlemek için herkesin korkusuzca bulunabileceği bir ortamın oluşturulması gerekir. İçinde kritize edilme korkusu olan katılımcılar kendilerini müdafaa edecekleri için mevcut sorunlara çözüm bulmak imkansızlaşır.

Resim 3.2 Retrospective toplantısı

Bu yüzden her retrospektive toplantısı katılımcılar arasında yapılan bir anlaşma ile başlar. Bu anlaşmaya göre:

- Her katılımcı, ekip içinde çalışan herkesin kendi imkanları dahilinde mümkün olanın en iyisini yaptığını kabul eder.
- Herkes bu toplantıda amacın yazılım sürecini iyileştirmek olduğunu kabul eder. Amaç suçlu aramak değildir.
- Her katılımcı korku hissetmeden konuşabileceğini kontrol eder. Her katılımcı toplantıya katılan diğer katılımcıların yanında kendisini iyi hissedip, hissetmediğini kontrol eder.

Toplantıya katılan ekip elemanların proje ve gidişatı hakkında bilgiye sahip olmaları gerekir. Toplantı öncesi katılımcıların hazırlık yaparak, konuşulması gereken sorunlar hakkında fikir alış verişinde bulunurlar.

Çevik Çalışma Ortamı

İletişimin kaliteli olabilmesi için çalışılan ortamın bu aspekt düşünülerek tasarlanmış olması gerekir. XP nin iletişime verdiği önem pair programming

tekniğinde kendini göstermektedir. XP projelerinde iki programcı bir araya gelerek bir eş (pair) oluştururlar ve tek bir bilgisayar ve klavye ile ortak bir şekilde implementasyonu yaparlar. Bu programcılar arasında sıkı bir iletişimin oluşmasını sağlar.

Resim 3.3 Pair programming

Ekip çalışanları aynı odada, birbirlerinden uzak olmayacak şekilde konuşlandırılmalıdırlar. XP projelerinde müşteri de programcılarla aynı odada çalışır. Programcılar oluşan soruları doğrudan müşteriye yönelterek ve dakikalarla ölçülebilecek bir zaman biriminde cevap alarak, implementasyona devam ederler. Müşterinin olmadığı bir ortamı düşünelim. Eğer programcı müşteriye ulaşamıyorsa, e-posta ya da telefon aracılığıyla sorunun cevabını bulmaya çalışacaktır. Bu zaman alıcı bir işlemdir. Çoğu zaman programcılar bu durumda soruyu ileri atarak başka konular üzerinde çalışmaya devam etmektedirler. Bu soru ve sorunların birikmesine ve zor çözülebilir hale gelmesine sebep verebilir.

Resim 3.4 Pair programming ve çalışma odası

Proje planlamasında kullanılan kullanıcı hikaye kartları ve diğer dokümanlar programcıların her an ulaşabilecekleri panolarda sergilenir. Bunun bir örneğini resim 3.5 de görmekteyiz.

Resim 3.5 Pano

Wiki

XP takımlarında bilgi alış verişini ve iletişimi kolaylaştırmak için Wiki araçları kullanılır. Wiki sistemi ekip içinde herkesin okuyup, değiştirebileceği HTML sayfalarından oluşur. Herkes yeni bir sayfa oluşturabilir, silebilir ya da değiştirebilir. Bu takım içinde bilgilerin kısa sürede merkezi bir platform üzerinden yayılmasını kolaylaştırır.

Wiki sistemleri internet üzerinden erişilir hale getirilebilir. Bunun en iyi örneği Wikipedia sayfalarıdır. Wikipedia dünyanın her yerinden birçok insanın katkılarıyla oluşmuş bir online ansiklopedidir.

İlk Wiki 1995 yılında Ward Cunningham tarafından oluşturuldu. Wiki nin bilgi yönetiminde sağladığı avantajları şu şekilde sıralayabiliriz:

- Her sayfa herhangi bir kullanıcı tarafından değiştirilebilir.
- Sayfa değişikliklerini kontrol etmek için versiyon kontrol sistemi kullanılır. Kim ne zaman nasıl bir değişiklik yapmış kontrol edilebilir.
- Sayfalar arası linkler oluşturmak mümkündür. Birbiriyle ilişkili sayfalar bu şekilde gruplanabilir.
- Proje hakkındaki kolektif bilgi merkezi bir platformda tutulur. Herkes istediği zaman bu bilgilere bir web tarayıcısı üzerinden erişebilir.
- Detaylı arama yapma imkanı sunar.
- Ekip içinde e-posta iletişimi aza indirilir, çünkü bir çok soru Wiki sistemi tarafından cevaplanabilir.
- Kolay ve eğlenceli dokümantasyon oluşturma imkanı sağlar.
- Proje yönetim ve kontrolü Wiki tarafından desteklenir.

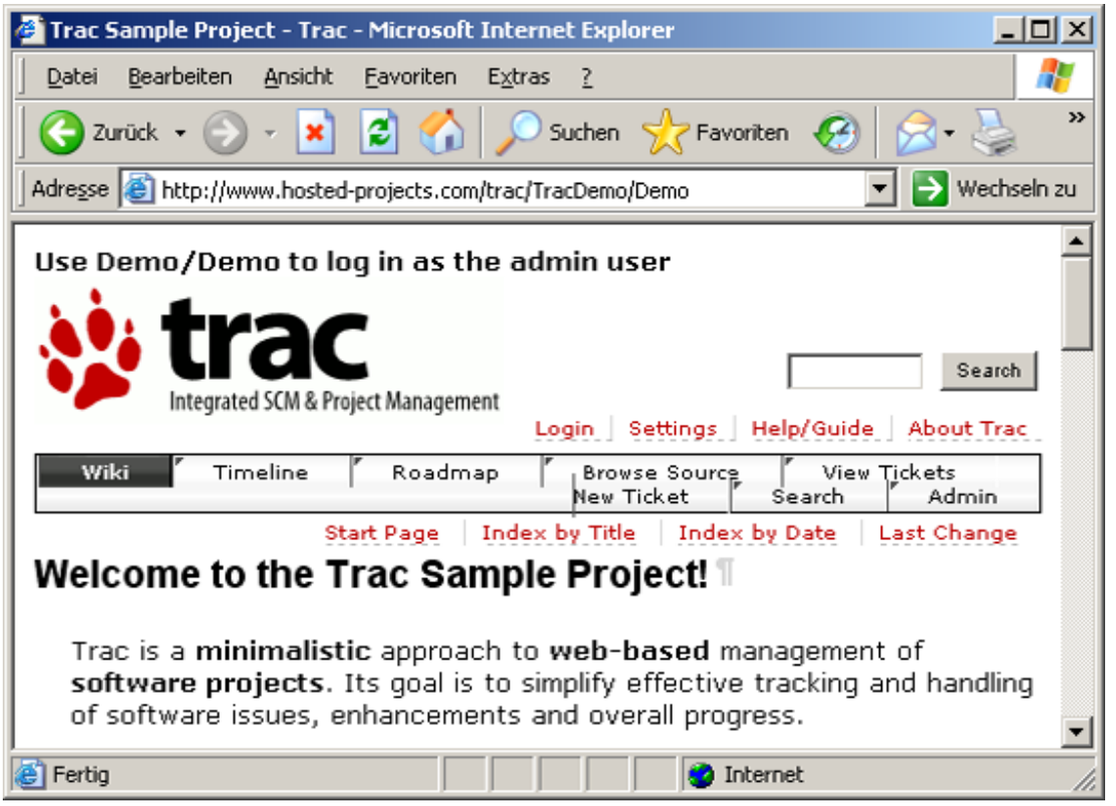
Çevik projelerde Wiki sistemleri yaygın bir şekilde kullanılır. Bilginin oluşmasını ve dağıtılmasını kolaylaştırmak için projelerinizde Wiki sistemlerini kullanmanızı tavsiye ediyorum.

Trac

Açık kaynaklı (open source) olan Trac sunduğu Wiki ve Ticketing sistemi ile son zamanlarda XP projelerinde sıkça kullanılan bir iletişim aracı olmuştur.

Trac Wiki nin ötesinde bir araçtır. Sunduğu Ticketing sistemi ile aktiviteler ya

da görevler (tickets) oluşturularak ekip çalışanlarına atanabilir. Bunlar Wiki sistemine linklenerek kolayca tüm ekip içinde paylaşılır hale getirilir.



Resim 3.6 Trac demo sayfası

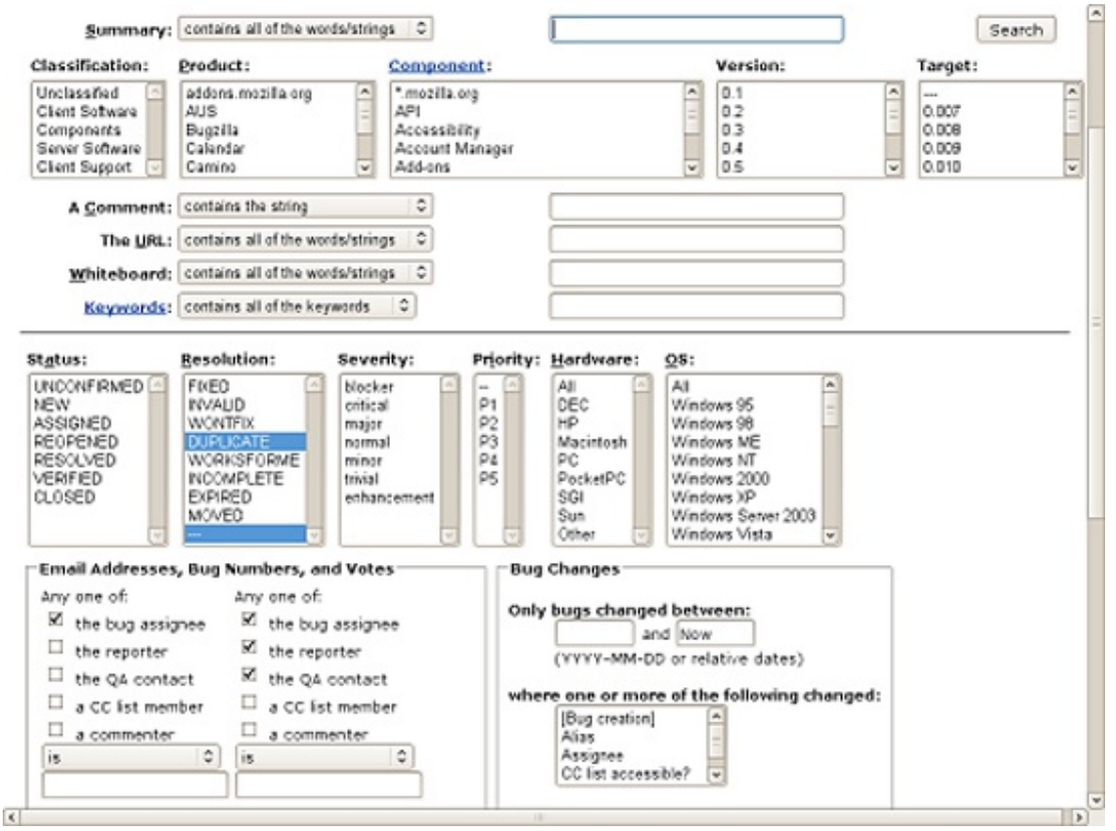
Trac Subversion ile entegre çalışabilir. Bu durumda versiyon kontrol sisteminde olan kodlar Trac aracılığıyla yüklenerek, kolayca gösterilebilir.

Trac kullanıldığı projelerde çeviklik açısından şu avantajları sağlar:

- Beyin fırtınalarında (brain storming) oluşan fikirler daha sonra değerlendirilmek üzere hemen Wiki sayfası olarak kolayca kayda geçirilebilir. Trac bilgi yönetimini kolaylaştırır.
- Kullanıcı hikayeleri Wiki sayfalarında detaylandırılabilirler. Tüm ekip çalışanları merkezi bir platformdan bu bilgilere ulaşabilirler.
- Oluşan her türlü hata (bug), yeni görevler (task) ve değişiklikler Ticketing sistemi kullanılarak takip edilebilir.
- Kodların web tarayıcısı üzerinden kolayca gösterimini sağlar.
- Sürüm ve iterasyon planları kolayca Wiki sayfası olarak tüm ekibin erişebileceği şekilde düzenlenebilir.

Bugzilla

Bugzilla yazılım esnasında oluşan hataları yönetmek için oluşturulmuş açık kaynaklı bir iletişim aracıdır. Tespit edilen hatalar Bugzilla da kayıtlanır ve sorunun çözülmesi için programcılara atanır.



Resim 3.7 Hata kaydetme sayfası

Hata kayıt esnasında birçok detay göz önünde bulundurulabilir. Çoğu zaman oluşan hatalar sistemin içinde barındığı ortamla direk alakalıdır. İşletme sistemi, kullanılan web tarayıcı türü, hangi sistem versiyonunda hatanın oluştuğu önemli bilgilerdir. Bugzilla ile hata kaydı yapılırken çok detaylı olarak bu bilgilerin sağlanması kolaylaştırılır.

Hatalar programcılara atanarak, hangi hatanın hangi programcı tarafından temizleniyor olması takip edilebilir. Programcı hatayı giderdikten sonra, bunu Bugzilla üzerine hata kaydını yapan kullanıcıya bildirir. Hatayı tespit eden kullanıcı yeni versiyonda hatanın olup, olmadığını tekrar kontrol ederek ya hata kaydını bitmiş olarak sonlandırır ya da yeni bir hata oluşması durumunda hata kaydını tekrar aktif hale getirerek, programcının konu hakkında uyarılmasını sağlar. Kullanıcılar arasındaki interaksiyon, Bugzilla tarafından gönderilen e-posta notifikasyonları aracılığıyla sağlanır.

Bugzilla ile sistemin hangi versiyonlarında hangi hataların oluştuğuna dair istatistiki raporlar hazırlanabilir. Bu raporlar sistemin stabilitesi hakkında fikir edinme sürecinde önemli bir rol alırlar.

4. Bölüm

XP Çalışma Ortamı

Giriş

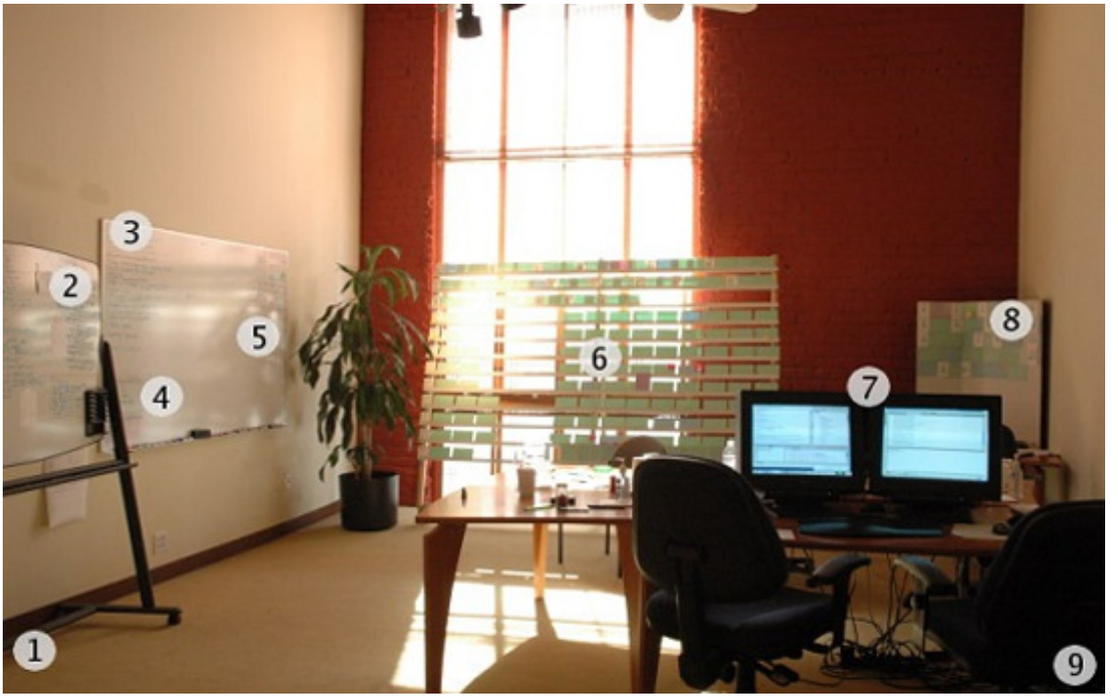
Extreme Programming bünyesinde birçok prensip ve tekniği barındırmaktadır. Bu kitabın ana konusu bu prensip ve tekniklerin neler olduklarını sizlere göstermektir. Sadece teorik bilgilerin XP yi öğrenmek ve uygulamakta yeterli olmadığını düşünüyorum. Bu sebepten dolayı bazı tekniklerin nasıl uygulandığını pratik örneklerle göstermeye çalıştım. Özellikle test güdümlü yazılım (Test Driven Development – TDD) teorik olarak anlatılacak ve anlaşılacak bir metot değildir. Kitapta sadece bu konuyu tematize eden pratik uygulamalı en az iki bölüm yer almaktadır. Kitabın diğer bir misyonu teorik bilgilerin verilmesi yanı sıra, pratikte bazı tekniklerin nasıl uygulandığını da göstermektir.

Bazı tekniklerin XP takımlarınca nasıl uygulandığını resimler aracılığıyla sizlere aktarmaya çalışacağım. Önemli olduğunu düşündüğüm ve pratikte çoğu zaman problem yaratan bir konuda XP çalışma ortamlarının nasıl yapılandırılmaları gereğidir. XP proje tecrübesinin eksikliği durumunda, böyle bir çalışma ortamının nasıl olması gerektiği hakkında kafalarda soru işaretleri oluşabilir.

Bu bölümde bir XP çalışma ortamının nasıl olması gerektiği sorusuna cevap arayacağız. XP iletişime dayanan ve onu arayan bir metodolojidir. Bu yüzden çalışma ortamının şekillendirilmesinde iletişim akpektinin göz önünde bulundurulması gerekmektedir.

XP Çalışma Odası

XP nin sahip olduğu değerler ve prensipler bir XP projesinde çalışma odasının nasıl şekillendirilmesi gerektiği konusunda bize ip uçları verir. XP nin birçok prensibi ve tekniği yüz yüze iletişim gerektirir. Resim 4.1 de tipik bir XP çalışma odası yer almaktadır. Gelin odayı ve yapısını yakından inceleyelim.



Resim 4.1 Tipik bir XP odası

1. Müşteri

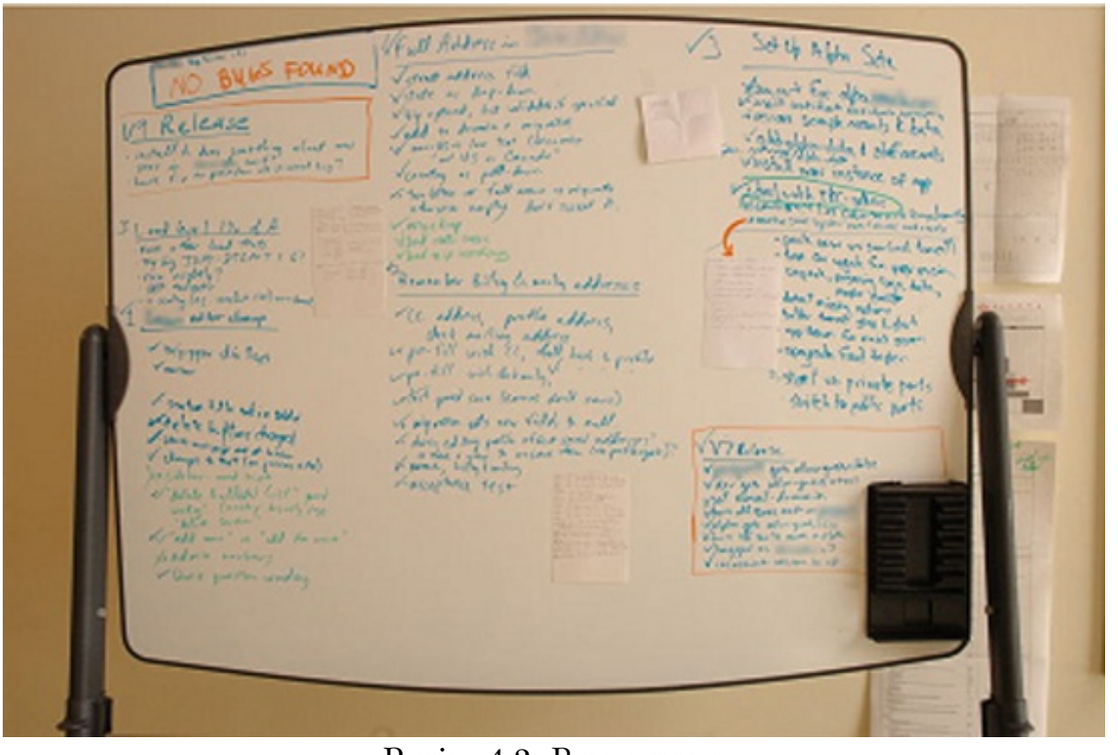
Resim 4.1 de müşterinin kullandığı çalışma masası resim dışında kalmakla beraber, programcıların çalıştığı masaya 3-4 metre uzaklıktadır. Bu müşteri ile programcılar arasında yüz yüze iletişim kurulmasını kolaylaştırır. Programcılar implementasyon esnasında oluşan soruları müşteriye yönelterek kısa sürede cevaplanmalarını sağlarlar.

Çoğu zaman müşterinin bir projeye %100 katılamayacağı düşünülür. Eğer bir müşteriniz talep üzerine projeye bu şekilde iştirak edemeyeceğini söyleyecek olursa, o müşterinin projeyi ne oranda ciddiye aldığı tartışılır. Büyük bir ihtimalle müşteri söylediklerinde ve yapmak istediklerinde samimi değildir.

Seminerlerde bana sorulan en fazla sorulardan birisi de müşterinin nasıl kendi işini bırakıp da, %100 bir projede aktif olabileceğidir. Müşterinin %100 projeye dahil olması demek, çalışma zamanının %100 ünde programcılarla aynı odada oturuyor demektir, %100 programcılarla beraber çalışıyor demek değildir! Çoğu zaman bir bilgisayar aracılığıyla müşteri kendi günlük işlerini yapabilir. Burada önemli olan nokta müşterinin programcılara yakın olmasıdır. Gün boyunca belki programcılar birkaç soruyu müşteriye soracaklardır, belkide hiçbir soru sorulayacaktır. Müşteri kendi işlerine devam eder, ara sıra programcılara sorularında yardımcı olur.

2. Beyaz Pano

Beyaz panolar iletişim aracı olarak idealdir. Özellike bir iterasyon esnasında bilgi akışını güçlendirmek için beyaz panolar kullanılır.



Resim 4.2 Beyaz pano

Beyaz panoya bir iterasyon esnasında üzerinde çalışılan kullanıcı hikayeleri hakkında detaylar yazılabilir. Resim 4.2 de yer alan panoda programcı ekip güncel iterasyonda yer alan kullanıcı hikayelerinin ihtiva ettiği görevleri panoya yazmıştır. Tamamlanan görevler üstleri çizilerek, görevin tamamlandığı belirtilmiştir. Ayrıca panonun sol üst köşesine oluşan hatalar yazılmaktadır. Sağ alt bölümde bir sonraki sürümün kurulumu için gerekli spesifik işlemleri anlatan notlar yer almaktadır.

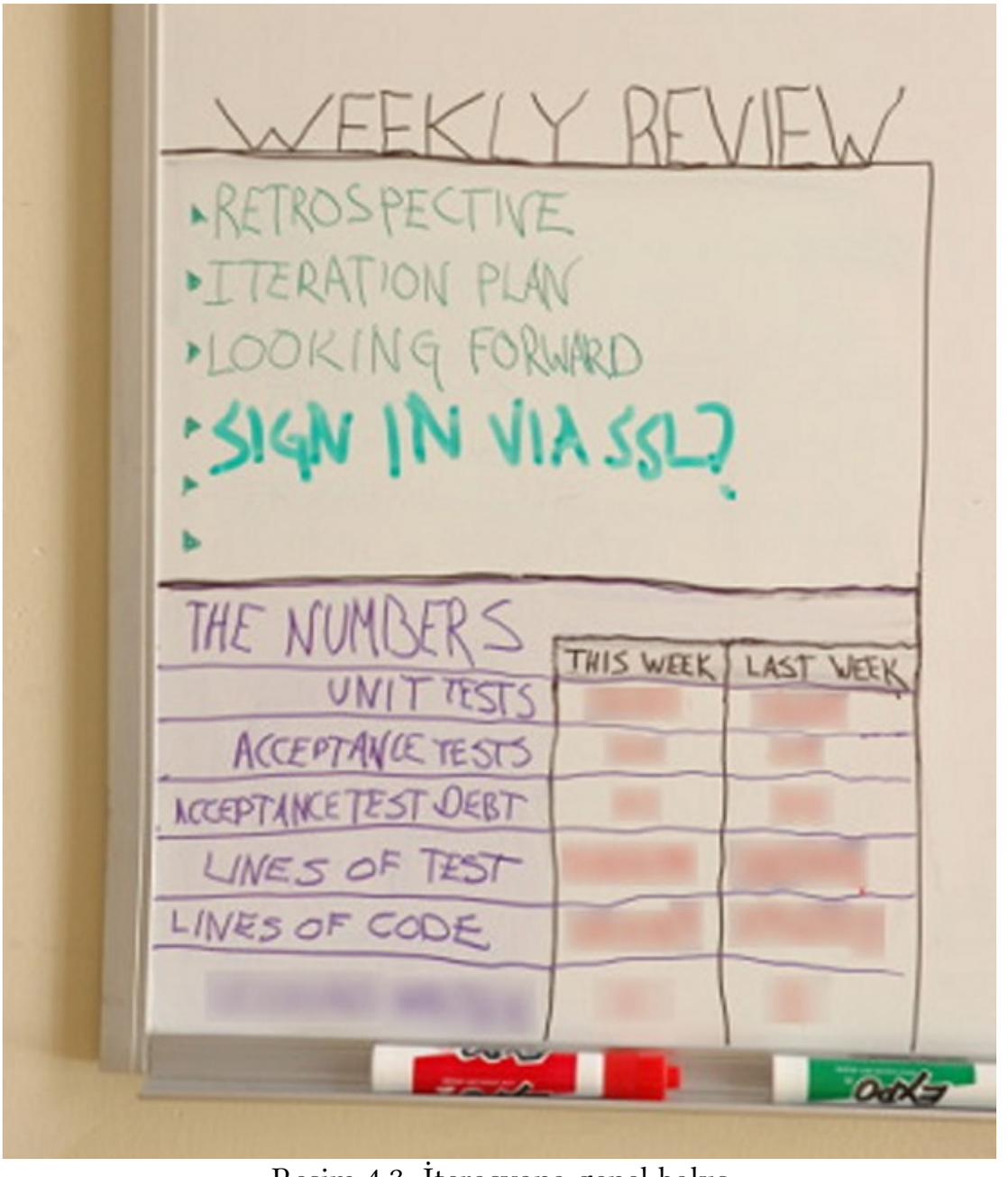
3. Çözülmesi Gereken Teknik Sorunlar

İmplementasyon esnasında programcılar zaman zaman teknik sorunlarla karşılaşabilirler. Bunların çözümü hemen mümkün olmayabilir. Bu problemlerin to-do listesi olarak beyaz panoda kaydedilmesinde fayda vardır. Programcılar zaman buldukça bu listede yer alan sorunları çözmeye çalışırlar.

4. İterasyon Bitirme Toplantısı ve Metrikler

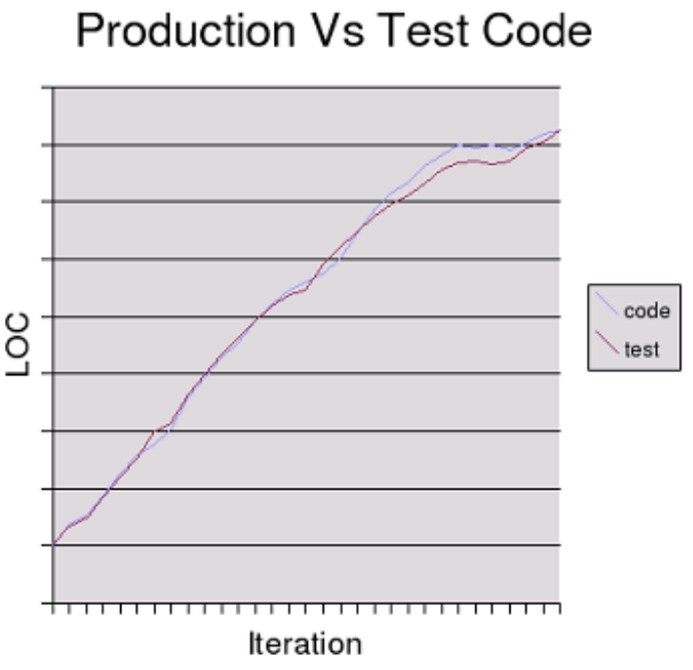
İterasyon bitiminde tüm ekip bir araya gelerek iterasyon bitirme toplantısı düzenlenir. Örneğin iterasyon süresi bir hafta olarak seçilmişse, cuma günü

öğleden sonra bu toplantı yapılır.



Resim 4.3 İterasyona genel bakış

Toplantının içeriğinde iterasyona geri bakış (retrospective) ve bir sonraki iterasyon planlaması yapılması yer alır. Resim 4.3 de toplantının içeriği yer almaktadır. Alt bölümde bazı metrikleri görüyoruz, örneğin hazırlanan birim test adedi, onay/kabul test adedi, testlerin toplam satır adedi vs. Bu metrikler bir önceki haftanın metrikleri ile kıyaslanmaktadır.



Resim 4.4 Kod satırlarıyla test satırlarının karşılaştırıldığı metrik

5. Genel Kullanım Pano Alanı

Resim 4.1 de yer alan 5 numaralı alan ekip tarafından tasarım ve analiz için kullanılan pano alanıdır. Programcılar arasında fikir alış verişini sağlamak, beyin fırtınalarında oluşan fikirleri kaydetmek ve tasarım için gerekli UML diyagramlarını çizmek için kullanılan alandır.

6. Hikaye Kart Panosu

Kullanıcı hikayeleri (user story) hikaye kartlarına (story card) yazılır. Resim 4.5 de bu kartların yer aldığı bir pano görülmektedir.

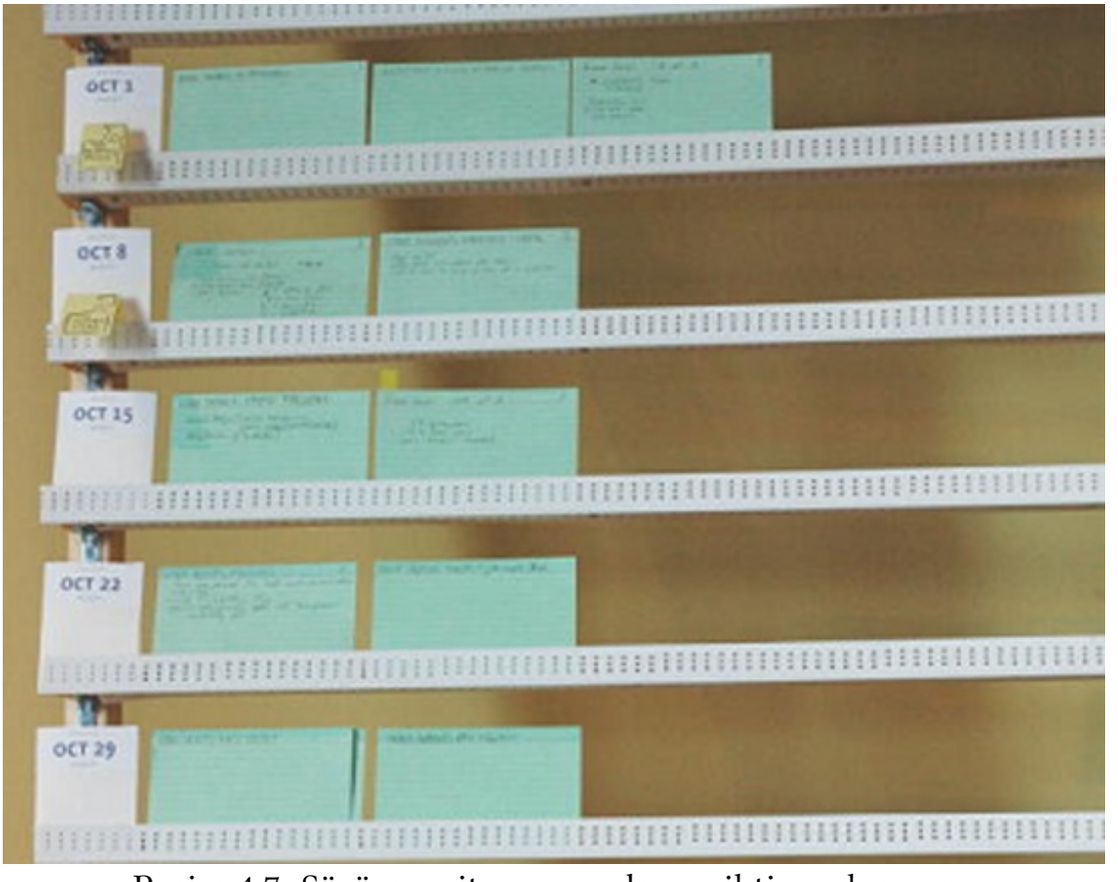
Resim 4.5 Sürüm ve iterasyon planını ihtiva eden pano

Panonun ilk üç katında tamamlanmış iterasyonlar ve ihtiva ettikleri kullanıcı hikayeleri yer almaktadır. Tamamlanan iterasyonlar, iterasyonun başlangıç tarihini ve implementasyon için kullanılan zamanı (5 = 5 gün) ihtiva eden beyaz kartlarla birbirini takip etmektedir.

Resim 4.6 Sürüm ve iterasyon planını ihtiva eden pano

Panonun 4-12 katları planlanan 9 iterasyonu ihtiva eder. Her iterasyon 1 hafta olduğu için panonun bu katlarında 9 haftalık çalışma süresi planlanmıştır. 4. kat güncel iterasyondur. Her katın sol bölümünde o iterasyonda implemente

edilecek olan kullanıcı hikayeleri yer alır. Her katın sağ bölümünde önemlilik derecesi düşük olan ve henüz iterasyon planlarında yer almamış kullanıcı hikayeleri yer alır.



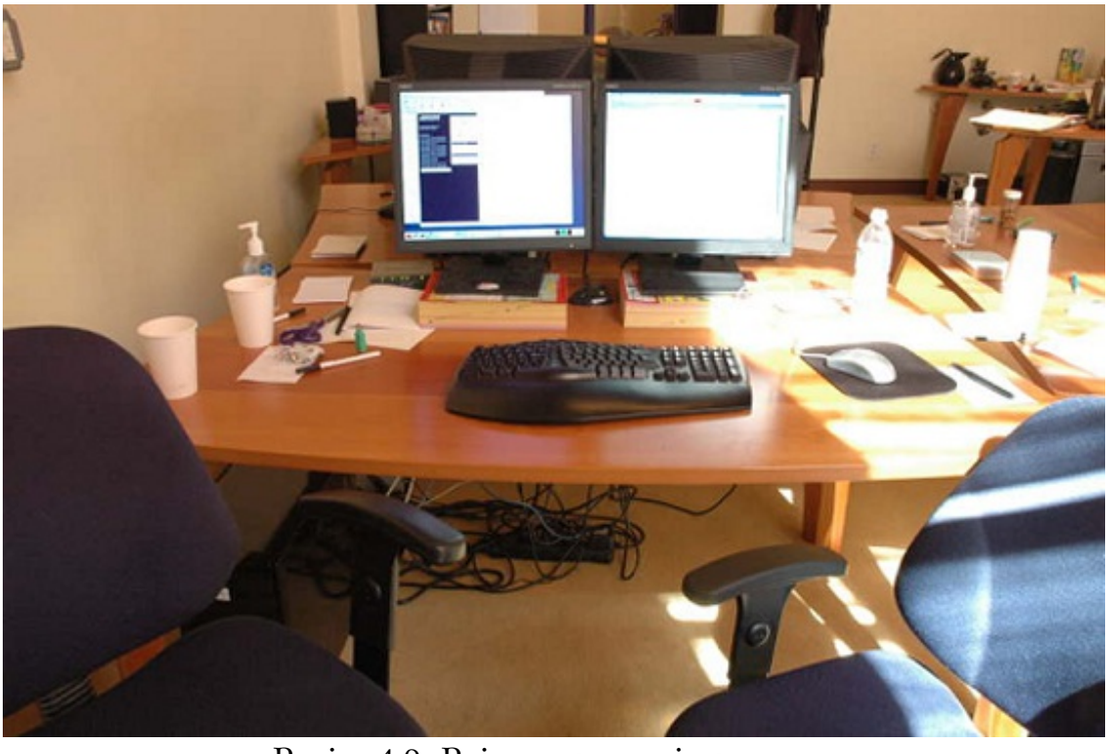
Resim 4.7 Sürüm ve iterasyon planını ihtiva eden pano

Resimlerde görüldüğü gibi hikaye kartları çok basit araçlar kullanılarak oluşturulabilir. Hikaye kartlarının ana amacı programcı ekip ve müşteri arasındaki diyaloğu kuvvetlendirmek ve projede kaydedilen ilerlemeyi görsel olarak sağlamaktır. Bir bilgisayar programı kullanılarak ta hikaye kartları oluşturulabilir. Lakin bu yöntem hikaye kartları kadar çevik olmayacaktır, çünkü bir bilgisayarı açıp, dijital hikaye kartlarına ulaşılana kadar az denilmeyecek bir zaman geçebilir. Oysa hikaye kartları bir pano üzerinde herkesin görebileceği şekilde dizilebilir ve kullanılabilir.

Resim 4.8 Sürüm ve iterasyon planını ihtiva eden pano

7. Programcı Çalışma Masaları

XP projelerinde iki programcı bir araya gelerek, implementasyonu ortak gerçekleştirirler. Bu yönteme pair programming ismi verilmektedir. Pair programming seanslarında iki programcı bir bilgisayarı ortak kullanır.



Resim 4.9 Pair programming masası

8. Tamamlanan Hikaye Kartları

İmplementasyonu tamamlanmış olan hikaye kartları, hikaye kartlarının yer aldığı panoda yer işgal etmemeleri adına başka bir panoda toplanır. Bunun bir örneği resim 4.10 da yer almaktadır. Tamamlanan hikaye kartlarının göz önünde bir yerlerde olmaları, retrospective ve diğer toplantılarda hemen erişilebilir olmalarını kolaylaştırır.

Resim 4.10 Tamamlanan hikaye kartları

9. Sürekli Entegrasyon

Resim 4.1 de resim dışında kalmış olan 9 numaralı alanda sürekli entegrasyonu gerçekleştiren bir Jenkins sunucusu yer almaktadır. Jenkins programcılar tarafından kod üzerinden yapılan her değişikliğin ardından uygulamayı yapılandırarak (build) tüm testleri otomatik olarak koşturur ve sistemin ne durumda olduğunu test eder. Kırılmalar olması durumunda, programcılar e-posta aracılığıyla uyarılarak, hatanın biran önce giderilmesi sağlanır.

5. Bölüm

XP Projesi

Giriş

Extreme Programming konusunu bir XP projesini en basit haliyle yakından inceleyerek daha iyi anlayabileceğimizi düşünüyorum. Bu amaçla birlikte bir Shop sisteminin XP tarzı nasıl oluşturulabileceğine yakından göz atacağız. Kitabın bu bölümünde temelini atacağımız Shop sisteminin ilerleyen bölümlerde işlediğimiz temaya paralel olarak implemente edeceğiz. Amacım sizlere sadece XP tarzı yazılımın nasıl yapıldığını göstermek değil. Çevik süreçte kullanabileceğimiz araçları da yeri geldiğinde size tanıtacağım.

Pratik yapılmadığı taktirde, kuru kuru kaynakları okuyarak, XP yi uygulamak mümkün değildir. Aslında birinci bölümde XP hakkında söylenebilecek birçok şeyden bahsettik. XP nin ne olduğunu şimdi biliyorsunuz, ama nasıl uygulanacağı hakkında bir fikre sahip olmayabilirsiniz. Bu yüzden bir Shop sistemi örneğinde XP yi nasıl uygulayabileceğimizi yakından göreceğiz.

Yazılım yapabilmek için öncelikle ne yapmamız gerektiğini bilmemiz gerekiyor. Burada müşteri, yani programının yapılmasını isteyen, para ödeyerek programı satın alan şahıs sahneye çıkıyor. Ne istediğini sadece müşteri en iyi bilebilir. Zaman zaman ne istediğini tam olarak bilmeyen ya da ifade edemeyen müşterilerle karşılaşsak ta, genel olarak müşteri gereksinimlerinden yola çıkarak bir programa şekil verilir. İsterseniz öncelikle müşterinin ne istediğine ve bu isteklerden yola çıkarak nasıl bir yazılım sürecini başlatabileceğimize bir göz atalım.

Müşteri Ne İster?

Bir programın oluşumundaki yön verici en önemli etken, programı kullanacak olan müşterinin iş piyasasındaki gereksinimleridir. Program müşterinin iş hayatını kolaylaştırmak, firma bünyesindeki aktiviteleri organize etmek ve kazanç sağlamak için kullanılır. Yazılım esnasında müşteri tarafından oluşturulan kriterlerin dikkate alınması gerekmektedir, aksi taktirde müşterinin isteklerini karşılayamayan ve günlük iş hayatında kullanılamaz bir program ortaya çıkar.

Yazılım süreci müşteri isteklerinin analizi ile başlar. Analiz safhasında müşterinin gereksinimleri tespit edilir ve yazılım için gerekli taban oluşturulur. Analiz ve bunu takip eden yazılım, müşteri isteklerinin transformasyona uğradığı ve netice olarak bir programın oluşturulduğu karmaşık bir süreçtir.

Çevik süreç bu transformasyonun bir parçasıdır ve karmaşıklığı azaltmak ve kontrol etmek için kullanılır.

Resim 5.1 Program müşteri gereksinimlerinin transformasyonudur

Transformasyon müşteri gereksinimlerinin tespiti ile başlar. Çevik süreçlerde müşteri gereksinimleri merkezi bir rol oynamaktadır. Sadece müşteri ne istediğini bilebilir ve onun istekleri doğrultusunda süreç ilerler. Burada amaç müşterinin isteklerini en son detayına kadar tespit edip, akabinde programlamaya başlamak değildir. Bu olumsuz bir sonuç doğurabilir. Daha ziyade müşteri için en önemli gereksinimler tespit edilmeli ve öncelik sırasına göre belirli bir zaman diliminde, bu gereksinimler doğrultusunda yazılım yapılmalıdır. Çevik süreç aşağıdaki şekilde işleyecektir:

- Müşteri gereksinimlerini dile getirir. Bunlar sistemden olan beklentileridir. Programcı ekip bu gereksinimlerden yola çıkarak müşteriyle beraber kullanıcı hikayeleri (User Story) oluşturur. Bunlar kullanıcı tarafından sistem üzerinde yapılması gereken bir veya iki cümle ile anlatılabilecek işlem tanımlarıdır. Örneğin "Müşteri shop içinde bir kitap seçerek sepetine ekler" bir kullanıcı hikayesidir. Bir cümleden oluşan bu kullanıcı hikayesi programcı için sistemin ne yapması gerektiğini tanımlar.
- Müşteri oluşan kullanıcı hikayelerini gözden geçirerek, en önemli olanları tespit eder. Her kullanıcı hikayesine bir öncelik notu verilir. En yüksek notu olan hikayeler öncelikli olarak implemente edilir.
- Çevik süreçte kısa aralıklarla yeni program sürümleri oluşturularak, müşteri ile diyaloğa girilir. Yeni sürümler bir ya da üç aylık zaman birimlerinde oluşturulur. Geri bildirimi kuvvetlendirmek için her sürüm bir ya da iki haftadan oluşan iterasyonlara bölünür. Her iterasyonda müşteri tarafından seçilmiş ve önemli olan kullanıcı hikayeleri implemente edilir. İterasyon sonunda oluşan program sürümü müşteriye gösterilerek, görüşleri alınır. Bu şekilde müşteriden geri dönüm sağlanarak, hazırlanan programın gereksinimler doğrultusunda geliştiriliyor olması sağlanır.
- Her iterasyon başlangıcında müşteri ile programcı ekip bir araya gelerek,

iterasyon planı oluşturulur. Müşteri implemente edilmesini istediği kullanıcı hikayelerini seçer. Programcı ekip her kullanıcı hikayesi için görev listesi (task list) oluşturur. Her programcı bir kullanıcı hikayesini oluşturan görevler için gerekli gördüğü zamanı tahmin eder. Bunların toplamı bir kullanıcı hikayesinin implementasyonu için gerekli zamanı verir. Buradan yola çıkarak bir iterasyonun ne kadar süreceği hesaplanabilir. Eğer iterasyon zamanı belli ise, örneğin iki hafta, bu zaman diliminde implemente edilebilecek kullanıcı hikayeleri seçilir. İterasyon için ayrılan zaman diliminde implemente edilemeyen kullanıcı hikayeleri bir sonraki iterasyona bırakılır. İterasyon sonunda mini sürüm oluşturularak, müşteriye sunulur.

- Her iterasyon sürecinde implemente edilmesi gereken kullanıcı hikayeleri için testler tasarlanır. Müşteri onay/kabul testlerini tanımlar. Daha sonra detaylı olarak inceleyeceğimiz onay/kabul testleri ile kara kutu olarak düşünebileceğimiz program kullanıcı gözüyle test edilir. Onay/kabul testleri ile sistemin kullanıcı açısından çalışırlığı kontrol edilir. Paralel olarak programcı ekip entegrasyon ve birim (unit) testlerini hazırlar.

Gereksinimlerin Tespiti

Yazılıma başlayabilmek için müşterinin ne istediğinin anlaşılması gerekmektedir. Gelin müşterinin ağzından ne istediğini beraberce görelim: Ahmet bey firması Kitap Ltd. için internet ortamında kitap satışı gerçekleştirmek üzere bir shop sistemi istemektedir. Sistemden beklentileri şu şekildedir:

- Kitaplar değişik kategorilerde müşteriye gösterilir.
- Müşteri kitap seçerek, sepetine atar. Bu işlemin ardından alış verişe devam edebilir ya da kasaya geçerek ödeme yapabilir.
- Satın alma işleminden önce müşterinin üyelik işlemi tamamlanır. Müşteri isim, adres, telefon ve e-posta adresi bilgilerini girerek sisteme üye olur. Daha sonra kendisine e-posta ile gönderilen şifre ve e-posta adresi ile login yaparak siparişlerini takip edebilir.
- Sipariş işlemi için bir ödeme yöntemi seçilmesi gerekmektedir. Müşteriye kredi kartı ya da banka havalesi ile ödeme yapma imkanı sunulur. Müşteri istediği ödeme yöntemini seçer ve siparişi tamamlar.
- Sistem tarafından müşteriye siparişin alındığına dair bir e-posta gönderilir.
- Kitaplar paketlenip kargoya verildikten sonra müşteriye sistem tarafından

bir e-posta gönderilir.
- Müşteri e-posta adresini ve şifresini kullanarak sisteme login yapabilir. Müşteri eski ve güncel siparişlerini shop sistemi içinde takip edebilir. Müşteriye istediği bir siparişin detayları gösterilir. Her siparişin bir statüsü vardır. Tamamlanmış olan siparişler için GÖNDERİLDİ, kargoda olan siparişler için KARGODA, henüz tamamlanmamış olan siparişler için HAZIRLANIYOR statüleri gösterilir.

Ahmet beyin isteklerini bu şekilde dile getirmesi neticesinde kendisinin nasıl bir sistem istediğine dair kafamızda bir fikir oluştu. Yukarıda yer alan gereksinimler doğrultusunda shop sisteminin yazılımını gerçekleştirmemiz gerekiyor. Bizden bunun ne fazlası, ne de eksiği istenmektedir. Birçok projede müşterinin yerine düşünüldüğü ve gerçek olmayan gereksinimlerden yola çıkıldığı için müşterinin istemediği özelliklerde bir program oluşmaktadır. Bunu engellemenin en kolay yolu, devamlı müşteri ile diyalogda kalarak, ondan alınan geri dönüm ile projenin kat ettiği mesafenin kontrol edilmesidir.

Keşif Safhası (Exploration Phase)

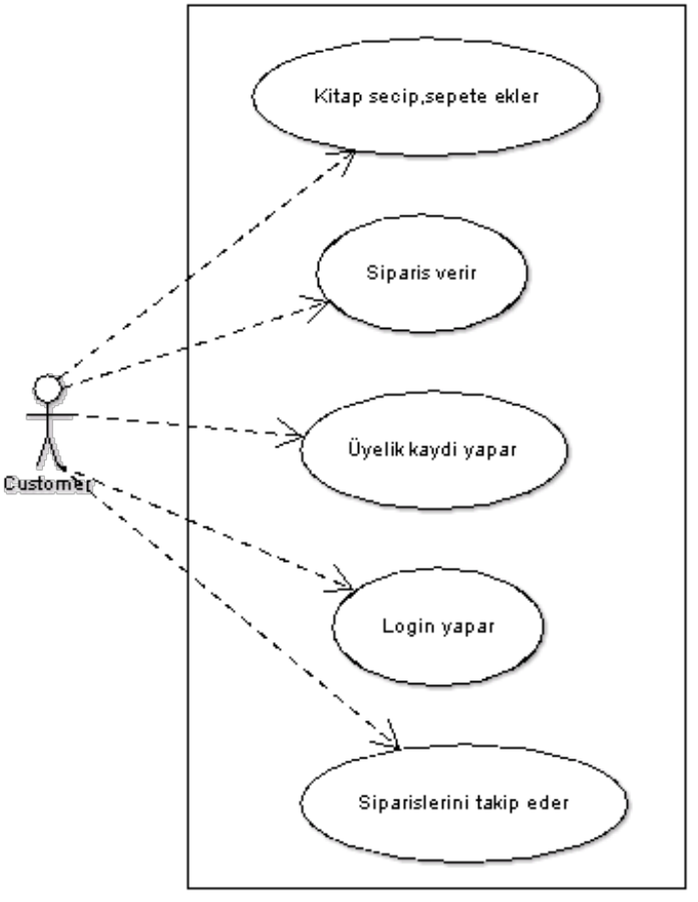
Tipik bir XP projesi keşif safhası ile başlar. Bu safha müşterinin ne istediğini anlamak için çalışmalar ihtiva eder. Programcı ekip tarafından kullanıcı hikayeleri gözden geçirilir ve kullanılacak teknolojiler tespit edilir. AMDD (Agile Modelling Driven Design) kullanılarak proje için gerekli diyagram ve dokümanlar oluşturulur. AMDD yardımıyla oluşan dokümanlar daha sonra implementasyon sürecinde kullanılır. Aşağıdaki listede keşif safhasında yapılabilecek aktiviteler yer almaktadır.

- UML kullanılarak kullanım senaryoları (Use Case) oluşturulur.
- Program içinde kullanılacak nesnelerin yer aldığı domain modeli oluşturulur.
- Kağıt üzerinde kullanıcı arayüz prototipleri oluşturulur ve müşterinin görüşleri alınır.
- Sayfalar arasındaki bağlantıları gösteren sayfa navigasyon modeli oluşturulur.
- Kağıt üzerinde teknolojik mimari oluşturulur.
- Kullanıcı hikayelerinin yeterliliği analiz edilir. İlk sürüm için kullanıcı hikayelerinin yeterli düzeyde tanımlanmış olmaları gerekmektedir.
- Programcı ekip tarafından projenin kapsamı tespit edilir.

Keşif safhasında istenilen sistem hakkında detaylı bilgi oluşur. Buradan yola çıkarak, shop sistemi için yapılması gereken çalışmaları tek tek inceleyelim.

Kullanım Senaryosu

Kullanım senaryoları (Use Cases) ile kullanıcıların sistem ile nasıl çalışacakları belirlenir. Sistemin nasıl kullanıldığını bilmek, kullanıcıların ne istediklerini öğrenmek açısından çok önemlidir. Shop sistemi için aşağıda yer alan UML (Unified Modeling Language) kullanım senaryosu diyagramını oluşturuyoruz.

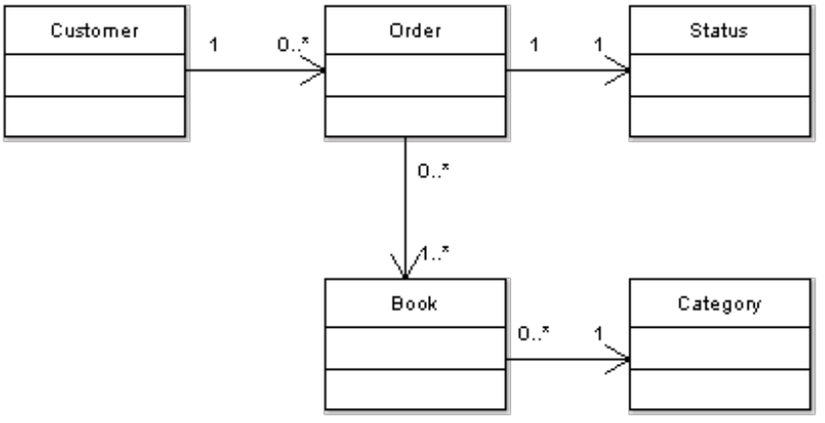


Resim 5.2 Use Case diyagramı

Alan (Domain) Modeli

Her programın faaliyet gösterdiği belli bir alan (domain) vardır. Çoğu zaman faaliyet alanı hakkında konuşmalar esnasında bu alana ait özel terimler kullanılır. Bu terimler programcılara programın yapısı hakkında ip uçları verirler. Buradan yola çıkarak programı oluşturan sınıflar (class) ve bu sınıflar arasındaki ilişkiler tanımlanır. Bunun sonucunda domain modeli oluşur.

Aşağıda shop sistemi için oluşturduğumuz domain modeli yer almaktadır.



Resim 5.3 Domain modeli

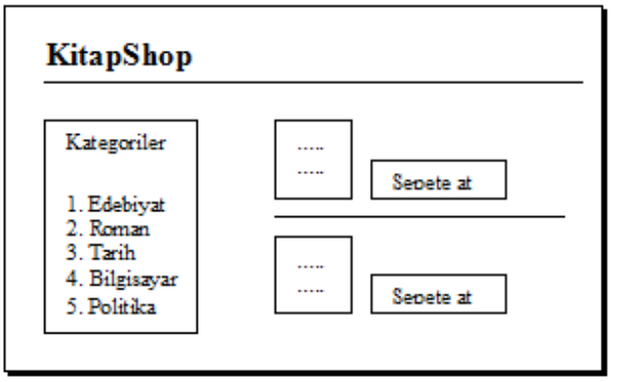
- Customer: Bir müşteriyi modellemek için kullanılır.
- Order: Müşterinin verdiği bir siparişi modellemek için kullanılır. Customer ve Order arasında 1 – 0..* ilişkisi vardır. Buna göre bir müşteri sıfır, bir ya da birden fazla sipariş verebilir. Her sipariş sadece bir müşteriye aittir.
- Status: Her siparişin bir statüsü vardır.
- Book: Shop sisteminde kitaplar satılacağı için Book isminde bir sınıf bulunmaktadır. Book ile Order arasında 1..* - 0..* şeklinde bir ilişki vardır. Buna göre kitap sıfır, bir ya da birden fazla sipariş içinde yer alabilir, bir sipariş içinde en az bir kitap bulunmak zorundadır.
- Category: Shop sisteminde kitaplar değişik kategorilerde yer alır. Book ve Category sınıflar arasında 0..* - 1 ilişkisi vardır. Buna göre bir kitap sadece bir kategori içinde yer alabilir, bir kategori içinde sıfır, bir ya da birden fazla kitap bulunabilir.

Kullanıcı Arayüz Prototipleri

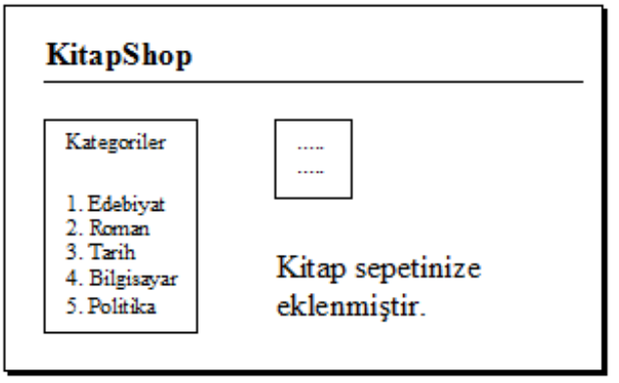
Domain modeli kafamızda oluşturulacak sistem hakkında iyi bir resmin oluşmasını sağlar. Bu süreci desteklemek için sistemde yer alacak kullanıcı arayüzlerinin kağıt üzerinde tasarlanmasında büyük fayda vardır. Bu sayede programcı ekip, müşteri ve sistem kullanıcıları için fikir alış verişini kolaylaştırmak ve daha etkin hale getirmek için bir platform oluşturulmuş olur.

Kullanıcı arayüz prototipleri programcılar tarafından implementasyon esnasında baz olarak kullanılacak olduklarından, mümkün mertebe gerçekleri yansıtacak şekilde tasarlanmaları gerekmektedir. Bu prototipler kalemle çizim usulüyle ya da shop sistemi örneğinde olduğu gibi eğer web tabanlı bir

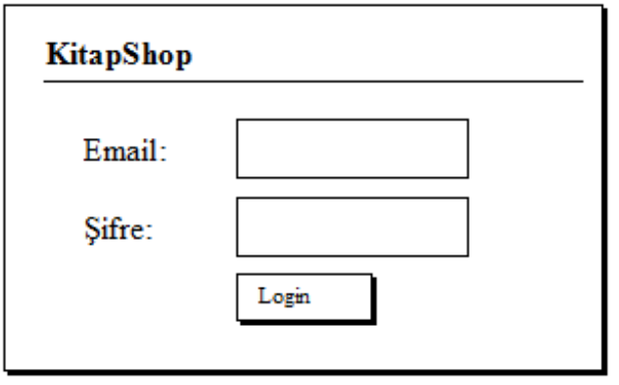
programsa HTML sayfaları olarak hazırlanabilir.



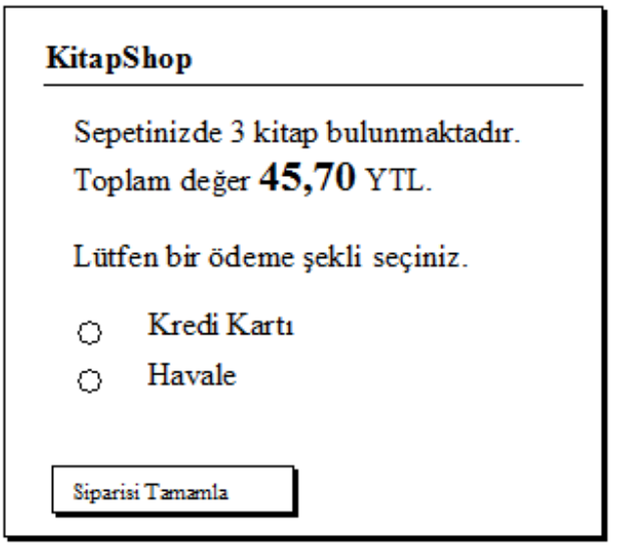
Resim 5.4 Anasayfa



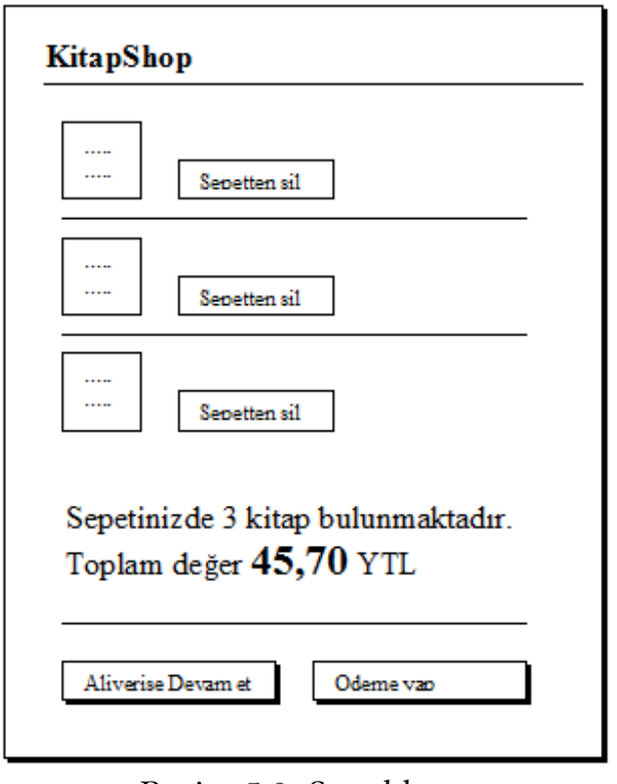
Resim 5.5 Sepete ürün ekleme sayfası



Resim 5.6 Login yapma sayfası



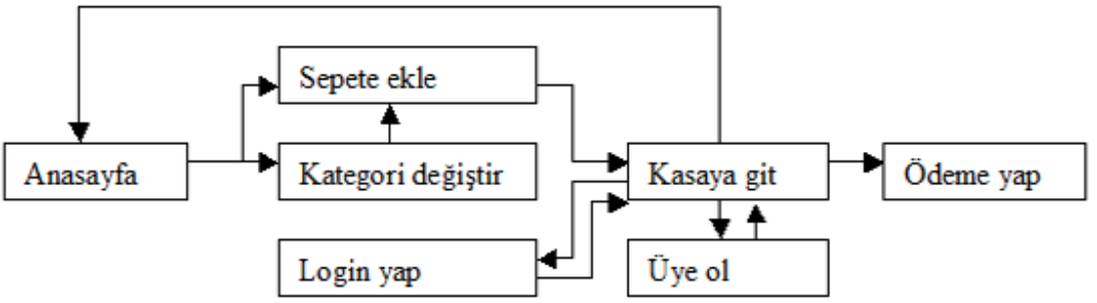
Resim 5.7 Ödeme sayfası



Resim 5.8 Sanal kasa

Sayfa Navigasyon Modeli

Önemli bir diyagram tipide sayfalar arası bağlantıları ve navigasyonu gösteren sayfa navigasyon modelidir.



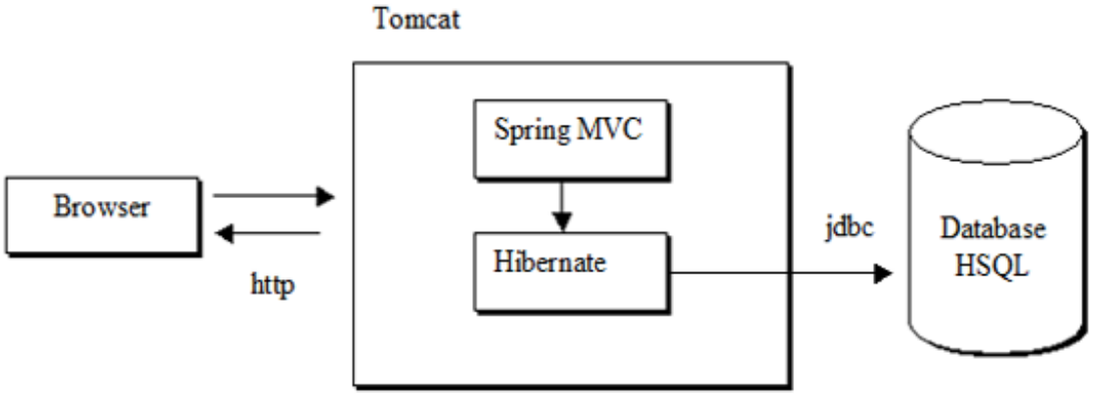
Resim 5.9 Sayfa navigasyon modeli

Shop sisteminin sayfa navigasyon modeli bir önceki resimde yer aldığı şekilde olacaktır. Kullanıcılar ilk önce ana sayfaya girerler. Ana sayfa üzerinde editör tarafından seçilmiş kitaplar ve kampanyalar yer almaktadır. Kullanıcı buradan bir kitap seçerek, sepetine ekleyebilir. Shop sisteminde yer alan sayfaların sol bölümünde kitap kategorileri listesi yer almaktadır. Kullanıcı kitap kategorisini bu liste içinde bir kategori seçerek değiştirebilir. Kategori değiştikten sonra kullanıcı bu kategoride yer alan kitap listesini görür. Buradan tekrar kitap seçerek, sepetine ekleyebilir ya da tekrar kategori değiştirebilir. Kullanıcı istediği kitapları seçtikten sonra ödeme yapmak için kasaya gider. Eğer kullanıcı daha önce üye olmuşsa, e-posta adresi ve şifresi ile login yapması sağlanır. Kullanıcı login sayfasından kasaya geri döner ve satın almak istediği kitap listesini görür. Buradan alış verişi tamamlamak üzere ödeme yapma sayfasına gönderilir. Kullanıcı üyeliğinin olmadığı durumda üye olabileceği sayfaya yönlendirilir. Üyelik işleminin ardından kasaya geri döner ve ödeme yapmak üzere ödeme yap sayfasına yönlendirilir. Kasaya gitmiş olan kullanıcı isteği üzerine alış verişe devam etmek üzere ana sayfaya geri dönebilir.

Sayfa navigasyon modeli programcı ekip ve sistem kullanıcıları arasında köprü vazifesi görür. Etkin bir biçimde oluşturulmayan sayfa navigasyon modeli kullanıcıların istemedikleri bir sistemin ortaya çıkmasına sebep verebilir. Bu sebepten dolayı sistemde bulunan sayfaların ve bu sayfalar arasındaki bağlantıların yer aldığı bir modelin oluşturulması büyük önem taşımaktadır.

Teknik Mimari

Projenin teknik mimarisi proje içinde kullanılan teknolojik ögeleri ihtiva eder. Projenin başlangıcında teknik mimarinin oluşturulması gerekmektedir. Programcı ekip teknik mimarinin öngördüğü teknolojik ögeler doğrultusunda çalışmalarını sürdürür.



Resim 5.10 Teknik mimari

Bir önceki resimde shop sistemi için oluşturduğumuz teknik mimari yer almaktadır. Program Tomcat JSP/Servlet Container içinde çalışacaktır, yani shop sistemi web tabanlı bir program olacaktır. Shop sistemi için kullanacağımız teknolojik ögeler şu şekildedir:

- Java 7
- Tomcat 7
- Spring 3.2
- Hibernate 3.2.6
- HSQLDB 1.8
- JUnit 4.11
- DBUnit 2.4.9
- Contract4J 0.8
- JMock 2.6.0
- Cruise Control 2.8.3
- Ant 1.9.3
- JDepend 2.9
- CheckStyle 5.7
- FindBugs 2.0.3
- PMD 5.1.0
- XPlanner 0.7
- MediaWiki 1.22.5
- Bugzilla 4.5.2

Web tabanlı olan shop sistemini Spring MVC web çatısı ile implemente edeceğiz. Spring MVC ve ismi geçen diğer programların nasıl kullanıldığını tek tek çevik süreç içinde göreceğiz.

Planlama Safhası (Planning Phase)

Keşif safhasını planlama safhası takip eder. Bu safhada hangi kullanıcı hikayelerinin (User Story) hangi zaman diliminde implemente edilmesi gerektiği tespit edilir. Bu safhada sürüm planı ve iterasyon planı hazırlanır.

Shop Sistemi Kullanıcı Hikayeleri

Sıra şimdi shop sistemi için gerekli kullanıcı hikayelerinin tespitine geldi. Aşağıdaki listede shop sistemi için gerekli olduğunu düşündüğüm kullanıcı hikayeleri yer almaktadır. Daha öncede belirttiğim gibi kullanıcı hikayelerini müşteri belirler. Aynı zamanda müşteri kullanıcı hikayeleri için öncelik sırasını tespit eder.

Tablo 5.1: Kullanıcı hikayelerini, öncelik sıralarını ve implementasyon zamanını ihtiva eden liste

#	Hikaye İsmi	Açıklama	Öncelik sırası	Tahmin
1	Kayıt	Kullanıcı kişisel bilgilerini kullanarak sisteme kayıt olur	**5**	1
2	Login	Kullanıcı email ve şifreni kullanarak sisteme login yapar.	**5**	1
3	Logout	Kullanıcı logout butonuna tıklayarak oturumunu sonlandırır.	**4**	0,5
4	Kategori secimi	Kullanıcı bir kategori seçerek, o kategoride yer alan kitapları edinir.	**5**	2
5	Alisveris sepeti	Kullanıcı bir ürün seçerek sepetine ekler.	**3**	1
6	Siparis	Kullanıcı sanal kasaya geçerek sipariş verir.	**5**	1
7	Siparis Emaili	Müşteriye siparişin alındığına dair email gönderilir.	**2**	1
8	Siparis listesi	Shop adminisratörü günlük sipariş listesine bakar	**4**	1
9	Ürün ekleme	Shop adminisratörü kategori seçerek yeni bir ürün ekler	**5**	2
10	Ürün silme	Shop adminisratörü seçilen bir ürünü siler.	**5**	1
			Toplam	11,5 iş günü

Tablo 5.1 de öncelik sırası ve tahmin isminde iki kolon bulunmaktadır. Öncelik sırası için müşteri tarafından verilebilecek en yüksek not 5 dir. Buna göre 5 notunu almış olan bir kullanıcı hikayesi en öncelikli olarak implemente edilmek zorundadır, çünkü müşteri bu özelliğin bir an önce oluşmasını istemektedir. Programcı ekip tahmin kolonuna bu kullanıcı hikayesinin implementasyonu için gerekli zamanı tahmin ederek yazar. İlk örneğimizde bir kullanıcının kayıt yapabileceği fonksiyonun implementasyon zamanı 1 gün olarak tahmin edilmiştir.

Tablo 5.1 de yer alan kullanıcı hikaye listesi tam değildir. Örnek olarak bu kullanıcı hikayelerini seçtim. İterasyon planında yer alacak kullanıcı hikayelerini öncelik sırasına göre seçmemiz gerekiyor. Buna göre ilk iterasyonda 1,2,4 ve 6 nolu kullanıcı hikayeleri yer alacaktır, çünkü bu kullanıcı hikayelerinin öncelik notu 5 dir. İlk iterasyon için seçtiğimiz kullanıcı hikayelerinin implementasyon süresi toplam 5 gün olarak tahmin edilmiştir, yani bir haftalık çalışma sonunda bu kullanıcı hikayelerinin implemente edilmiş olmaları gerekiyor. Buradan da anlaşılacağı gibi iterasyon süresini 1 hafta olarak belirliyoruz. Her iterasyon 1 hafta sürecek ve her sürüm 4 haftalık, yani dört iterasyon ardından oluşturulacaktır. Buradan yola çıkarak iterasyon ve sürüm planını oluşturacağız.

Sürüm ve İterasyon Planı

Sürüm planı yazılım sisteminin bir sonraki sürümü (versiyon) için yapılan plandır. Bu planda sürümün ihtiva edeceği kullanıcı hikayeleri yer alır. Kullanıcı hikayeleri iterasyonlarda gruplanır. Her sürüm birden fazla iterasyondan oluşur. Her iterasyon için bir zaman dilimi tanımlanır. Bu bir ile dört hafta arasında bir zaman dilimi olabilir. Her iterasyon öncesi, o iterasyonda implemente edilecek olan kullanıcı hikayeleri öncelik sırasına göre seçilir ve iterasyon planına eklenir. Kullanıcı hikayelerinin öncelik sırası müşteri tarafından belirlenir. Hangi kullanıcı hikayelerinin hangi iterasyonda implemente edileceğinin kararını da veren müşteridir.

Shop sistemi için her iterasyon bir hafta uzunluğunda olacaktır. Dört iterasyondan oluşan bir sürüm için dört haftalık bir zaman dilimi gerekmektedir, yani sürüm zamanı bir ay olarak seçilmiştir. Her iterasyon ve sürüm sonunda çalışır durumda olan program, yeni eklenen kullanıcı hikayeleri ile müşteriye sunulur ve değerlendirmesi beklenir. Müşteri programı test ederek görüşlerini bildirir. Eğer gerek gördüyse, kullanıcı hikayeleri ve öncelik sıralarını değiştirebilir. Programcı ekip bir sonraki iterasyon planında bu değişiklikleri göz önünde bulundurur.

Tablo 5.2: Sürüm ve iterasyon planı

İterasyon	Yapılacak olanlar	Bitiş tarihi
0	Programcı ekip için ortam kurulumu (Eclipse, ant, Hibernate, JDK setup)	03.10.2014
1	İlk iterasyon. 1,2,4 ve 6 nolu kullanıcı hikayeleri implemente edilecek	10.10.2014
2	9,10,3 ve 8 nolu kullanıcı hikayeleri implemente edilecek	17.10.2014
3	5 ve 7 nolu kullanıcı hikayeleri implemente edilecek	24.10.2014

İlk sürüm tarihi 24.10.2014'dür

Tablo 5.2 de ilk sürüm ve iterasyon planı yer almaktadır. İmplemente etmek istediğimiz shop sistemi çok geniş kapsamlı olmadığı için sürüm ve iterasyon planımızda buna orantılı olarak kapsamlı olmadı. Yapılan tahminler sonucunda programcılar ilk iterasyonun son günü olan üç ekim 2014 tarihinde çalışma ortamlarını kurmuş olacaklar. Beş ekimden itibaren ilk iterasyona başlayarak, 5 öncelik notuna sahip olan kullanıcı hikayelerini implemente edeceklerdir. İlk iterasyon on ekim tarihinde son buluyor. İkinci iterasyon on iki ekimde başlayıp, on yedi ekimde son buluyor. Son iterasyon 19-24 ekim tarihlerinde yapılacak.

Bu plana göre sürüm zamanı dört hafta olarak belirlenmiştir. Her iterasyonun süresi bir hafta olup, ilk sürümde on kullanıcı hikayesi implemente edilecektir.

Program v.1.0 versiyonunda müşteriye 24.10.2014 tarihinde teslim edilecektir.

Bakım Safhası (Maintenance Phase)

Bu programın bakımının ve geliştirilmesinin yapıldığı safhadır. Bu safhada kullanıcılar için eğitim seminerleri hazırlanır ve küçük çapta eklemeler ve sistem hatalarının giderilmesi için işlemler yapılır. Müşterinin istekleri doğrultusunda bir sonraki büyük sürüm için çalışmalara başlanır. Bu durumda tekrar keşif safhasına geri dönülmesi ve oradan işe başlanması gerekmektedir.

6. Bölüm

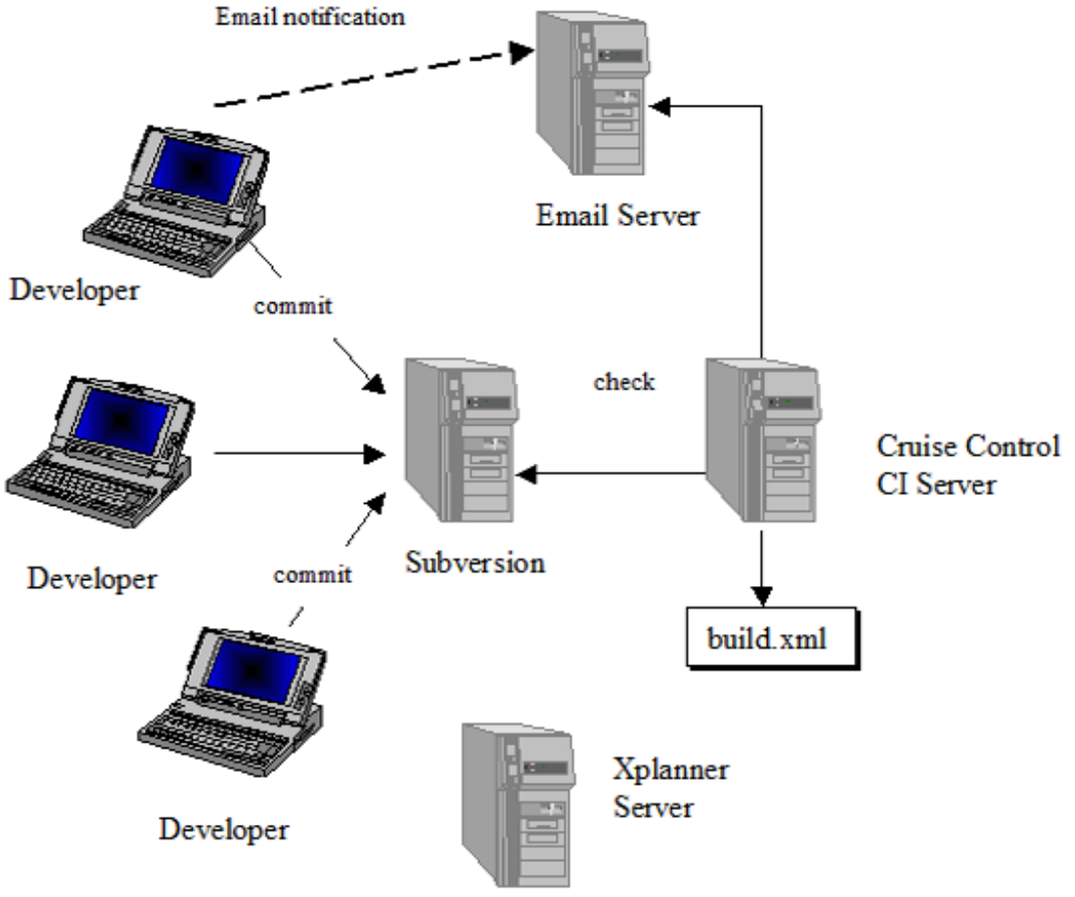
Çalışma Ortamı Kurulumu

Giriş

Çevik süreçlerde en önemli kaynak çalışma ekibinin parçası olan insanlardır. Takım içinde çalışma yeteneğine sahip insanlar verimli çalışarak projenin zamanında bitirilmesini mümkün kılarlar. Bunun yanı sıra verilimli çalışıp iyi sonuç alabilmek için iyi bir teknik çalışma ortamının oluşturulması gerekmektedir.

Çevik süreçlerde doğru araçların doğru yerde kullanılması da önemli bir rol oynamaktadır. Projenin başlangıcında yüksek bir teknolojik donanım gerekmemektedir. Projeye basit araç ve gereçlerle başlayıp, bu araçların yetersiz kaldığı yerlerde daha profesyonel olanları ile değiştirilmeleri doğru olacaktır. Örneğin lisans bedeli olan bir versiyon kontrol sistemi kullanmak yerine, Subversion gibi açık kaynaklı bir versiyon sistemi ile projeye başlanabilir. Ya da UML diyagramları çizmek için bir program satın almak yerine bir beyaz panonun üzerinde ilk çizimler yapılabilir.

Bu kitapta uyguladığımız çevik süreçte kullanacağımız araçlar resim 6.1 deki şekilde olacaktır.

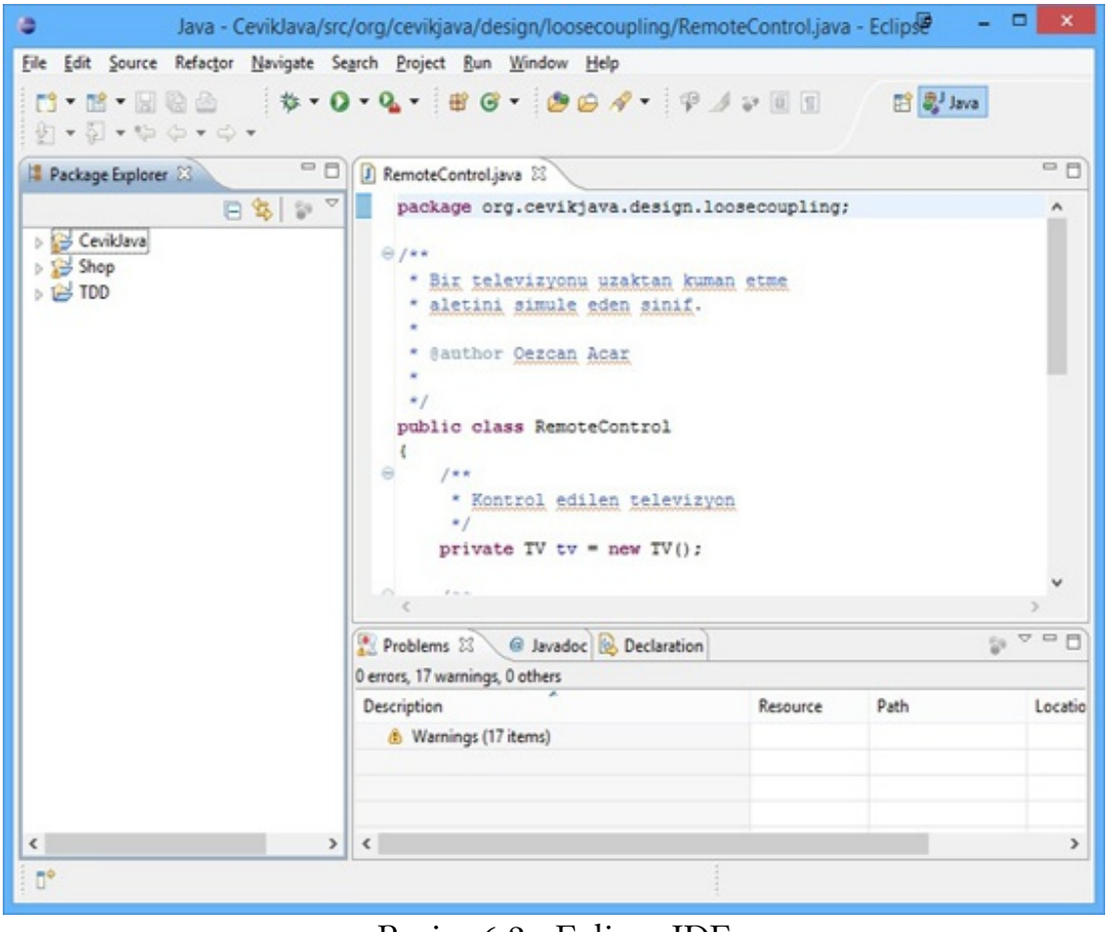


Resim 6.1 Çevik süreçte kullanılan araçlar

Projede versiyon kontrolü yapabilmek için Subversion versiyon kontrol sisteminden yararlanacağız. Her programcı ağ üzerinde Subversion sunucusuna bağlanarak, commit ve update işlemlerini yapabilir. Başka bir sunucu üzerinde Cruise Control çalışmaktadır. Cruise Control ile otomatik olarak sürekli entegrasyon gerçekleşecektir. Oluşan hatalar e-posta sunucusu üzerinden programcı ekibine e-posta olarak gönderilir. Proje yönetimi için XPlanner programından faydalanacağız. Görüldüğü gibi altyapı için birden fazla sunucu gerekmektedir. Küçük projelerde birden fazla program aynı sunucu üzerinde çalıştırılabilir. Örneğin bir sunucu Subversion, e-posta ve Cruise Control sunucusu olarak faaliyet gösterebilir. Orta ve büyük projelerde tek bir sunucu yetersiz kalabilir. Ben ismi geçen tüm programları kitabı yazdığım notebook bilgisayarımda (Windows 7, 8 GB Ram) çalıştırabildim ve tamamen entegre bir sistem oldu

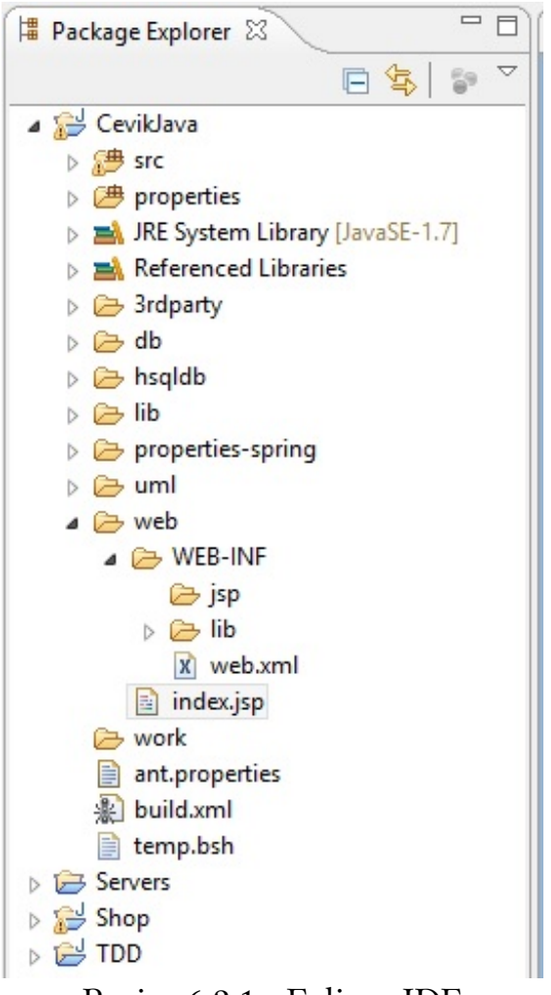
Eclipse

En önemli çalışma aracımız şüphesiz Eclipse IDE (Integrated Development Environment) olacaktır. Eclipse http://www.eclipse.org adresinden temin edilebilir. Pluginler aracılığıyla genişletilebilir yapıda olan Eclipse ile Java ve diğer dillerde program yazılımı mümkündür.



Resim 6.2 Eclipse IDE

Eclipse altında CevikJava isminde bir proje oluşturuyoruz. Dizin yapısı aşağıdaki şekilde olacaktır.



Resim 6.2.1 Eclipse IDE

- 3rdParty: Bu dizin içinde Tomcat, CheckStyle, JDepend gibi proje içinde kullandığımız programlar bulunmaktadır.
- lib: Proje içinde kullanılan .jar dosyalarının bir kısmı bu dizin içinde yer alır.
- Properties-spring: Spring için kullanılan konfigürasyon dosyaları (XML) bu dizinde yer alır.
- src: Shop sistemini oluşturan program kodu src (source) dizininde yer alır.
- src-test: Bu dizinde junit, entegrasyon ve onay/kabul testleri yer alır.
- uml: ArgoUML ile oluşturacağımız UML diyagramların yer aldığı dizindir.
- web: Bu dizinde JSP sayfaları ve Shop sistemi için gerekli diğer web dosyaları yer alır.
- work: Tomcat tarafından derlenip, Servlet haline dönüştürülen JSP sayfaları bu dizinde yer alır.

Ant

Ant bir proje yapılandırma (Build Tool) aracıdır. Ant ile otomatik olarak tüm Java sınıfları derlenir, JUnit testleri çalıştırılır ve gereken diğer işlemler yapılır.

Yapılmak istenen işlemler bir XML dosyasında tanımlanır. Bu dosyaya genelde build.xml ismi verilir. Projemize aşağıda yer alan build.xml dosyası ile başlıyoruz.

Kod 6.1 Ant script

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="CevikJava" default="compile">

    <property file="ant.properties" />

    <path id="compile.classpath">
        <fileset dir="${base.web.lib}">
            <include name="*.jar" />
        </fileset>
        <fileset dir="${base.lib}">
            <include name="*.jar" />
        </fileset>
    </path>


    <target name="compile" depends="clean">

        <javac srcdir="${base.src}"
            destdir="${build.web-inf.classes.dir}"
            debug="on" verbose="off">
            <classpath>
                <path refid="compile.classpath" />
            </classpath>
        </javac>
    </target>



    <target name="clean">
        <delete quiet="true" includeemptydirs="true">
            <fileset dir="${dist.dir}" />
        </delete>

        <delete quiet="true" includeemptydirs="true">
            <fileset dir="${build.dir}" />
        </delete>
```

```
        <mkdir dir="${build.dir}" />
        <mkdir dir="${build.web-inf.dir}" />
        <mkdir dir="${build.web-inf.classes.dir}" />
        <mkdir dir="${build.web-inf.lib.dir}" />
    </target>

</project>
```

build.xml dört bölümden oluşmaktadır. İlk bölümde <property> komutuyla ant.properties dosyasında tanımlanan anahtar ve değerleri build.xml içinden kullanılır hale getirilir. ant.properties dosyasının içeriği bir sonraki tabloda yer almaktadır. build.xml dosyasında kullanılan değişkenler ant.properties gibi dış bir dosyada tanımlanarak, build.xml in diğer bölümlerinde kullanılabilir.

Kod 6.2 - ant.properties dosyası

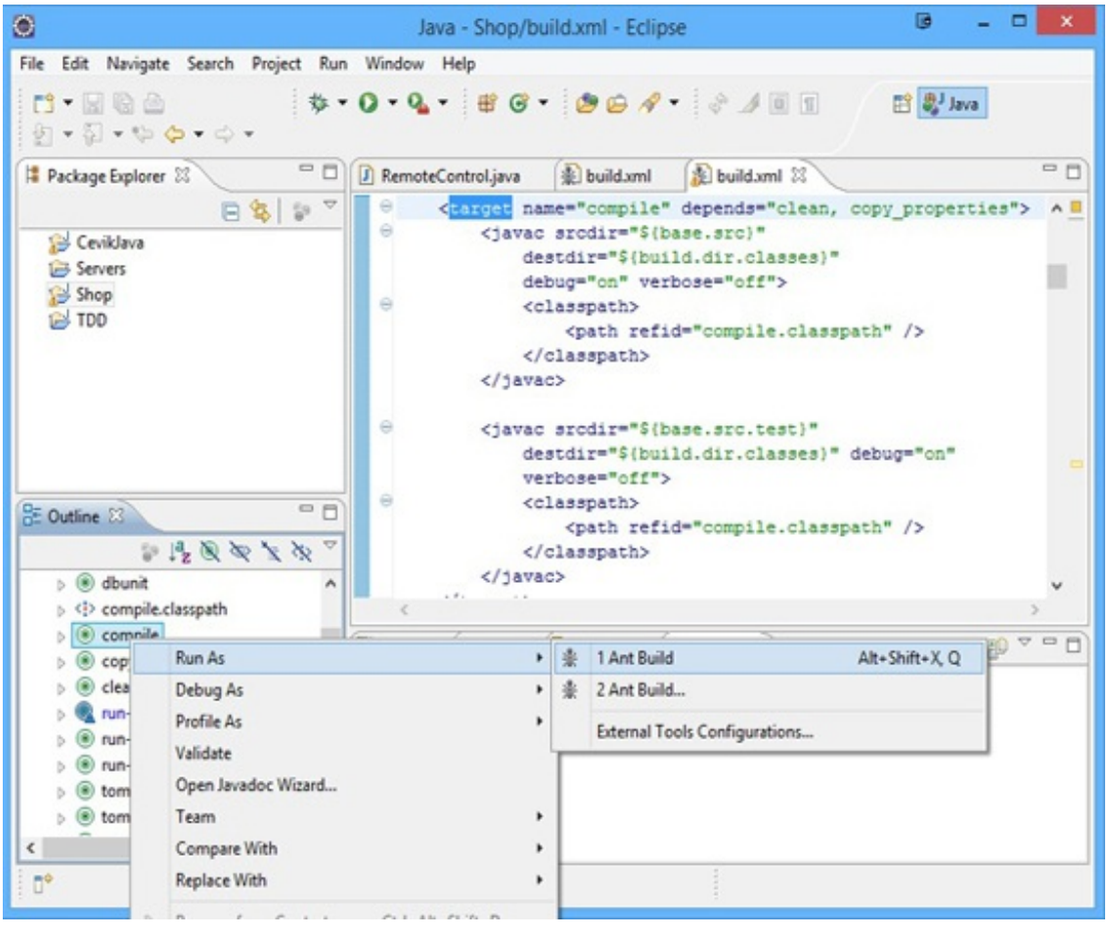
```
#base
base.dir=${basedir}
base.lib=${base.dir}/lib
base.web.lib=${base.dir}/web/WEB-INF/lib
base.src=${base.dir}/src
base.src.test=${base.dir}/src-test

#build
build.dir=${base.dir}/build
build.dir.root=${base.dir}/build
build.web-inf.dir=${build.dir.root}/WEB-INF
build.web-inf.classes.dir=${build.dir.root}/WEB-INF/classes
build.web-inf.lib.dir=${build.web-inf.dir}/lib
```

İkinci bölümde <path> komutuyla gerekli Jar dosyalarının yer aldığı bir classpath oluşturulur. Java sınıflarının derlenebilmesi için böyle bir classpath değerine ihtiyaç duyulmaktadır.

Üçüncü bölümde compile isminde bir hedef (target) tanımlanmıştır. <target> bir Java sınıfının metodu olarak düşünülebilir. Bir hedef içinde birden fazla işlem yapılabilir. compile ismini taşıyan hedef içinde <javac> komutunu kullanarak tüm Java sınıflarını derliyoruz. Oluşan sınıflar ${build.web-inf.classes.dir} yani ${base.dir}/build/WEB-INF/classes dizinine kopyalanacaktır. ${base.dir} projenin içinde yer aldığı ana dizindir (root). Her hedef görevine başlamadan önce başka bir hedef tetikleyebilir (çalıştırabilir). Böylece kendisi için gerekli ön çalışmaları başka bir hedef üzerinden halleder. Bir hedefin başka bir hedefe bağımlılığı depends ile tanımlanır. Compile ismini taşıyan hedef ihtiva ettiği işlemlere başlamadan önce clean ismindeki hedefin koşmasını sağlayacaktır.

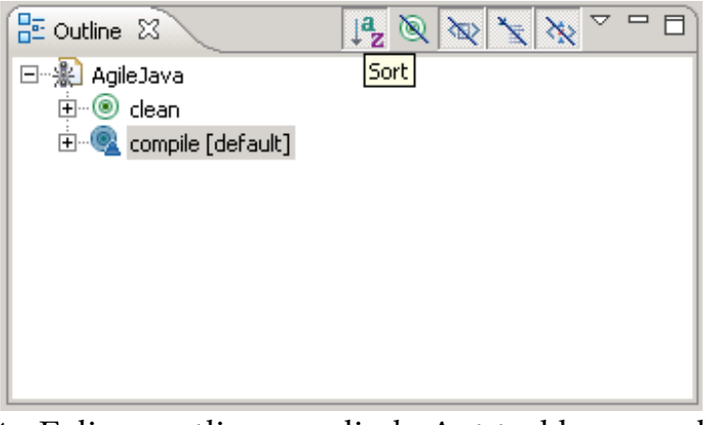
Son bölümde clean ismini taşıyan bir hedef yer almaktadır. Bu hedef içinde build ismini taşıyan ve derlenen sınıfların kopyalandığı bir dizin oluşturulur.

Eclipse beraberinde güncel Ant sürümünü getirmektedir. Bir sonraki resimde görüldüğü gibi Eclipse altında build.xml koşturulabilir.



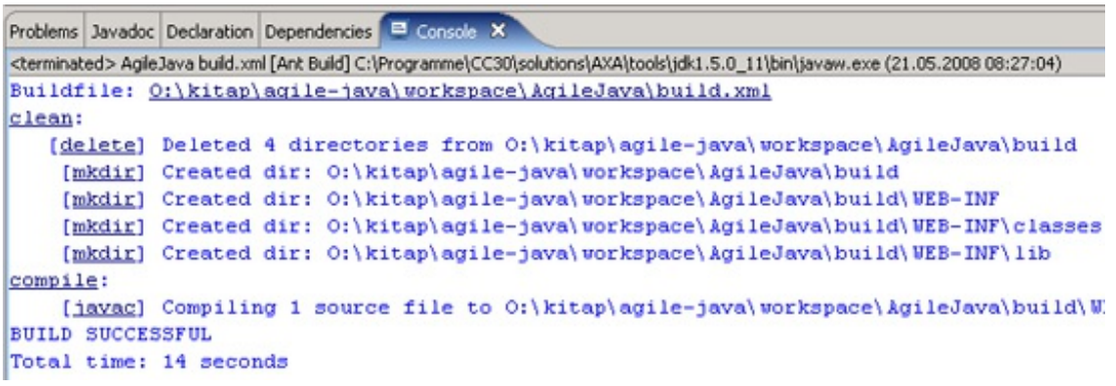
Resim 6.3 Eclipse Ant entegrasyonu

Eclipse Outline panelinde build.xml içinde tanımlanmış olan hedef listesi yer alır. Bu listeden herhangi bir hedef seçilerek koşturulabilir.



Resim 6.4 Eclipse outline panelinde Ant taskları yer almaktadır

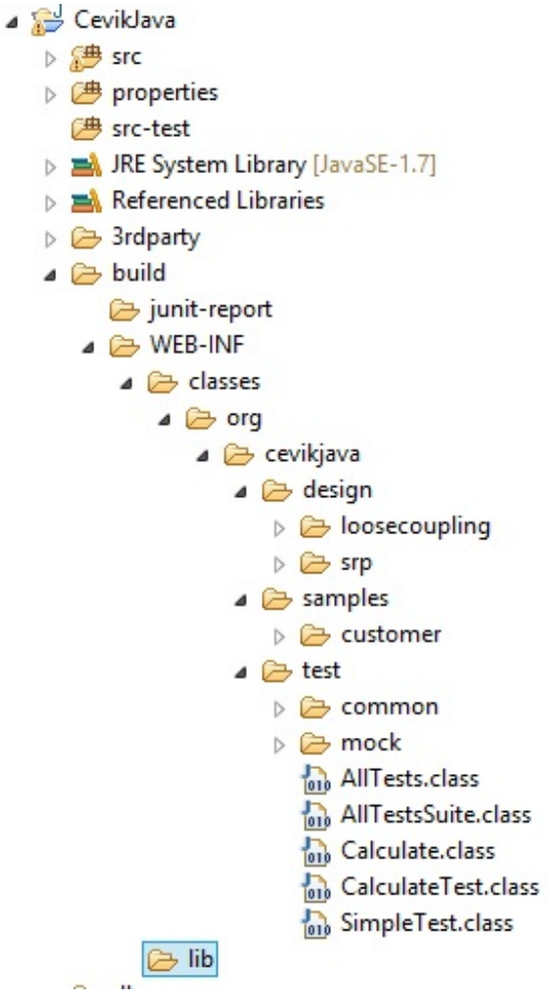
Eclipse console panelinde ekran çıktısı aşağıdaki şekilde olacaktır:



Resim 6.5 Eclipse console

Console panelinde görüldüğü gibi ant ilk önce clean ve akabinde compile hedeflerini çalıştırır. clean hedefi içinde build ve alt dizinler oluşturulur. Compile hedefi içinde mevcut Java sınıfları derlenir ve build\WEB-INF\classes dizinine kopyalanır.

Bu işlemin ardından dizin yapısı aşağıdaki şekilde olacaktır:



Resim 6.6 Proje dizin yapısı

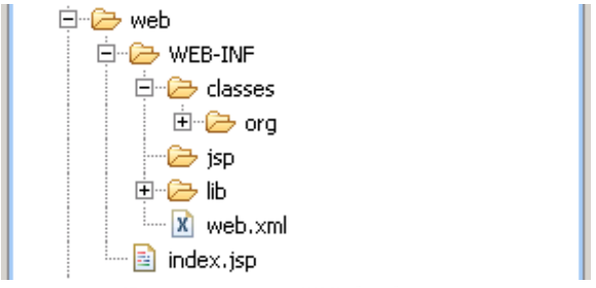
Ant ile proje yapılandırma işlemini otomatize etmiş olduk. Bahsettiğimiz işlemlerin elden yapılması çok zaman alıcı olacaktır. Bu yüzden çevik süreçlerde mutlaka Ant gibi proje yapılandırma aracın kullanımı gerekmektedir.

Projenin ilerleyen bölümlerinde JUnit testlerini ve diğer işlemleri nasıl otomatize edebileceğimizi göreceğiz.

Tomcat

Web tabanlı olan projemiz için Tomcat JSP/Servlet uygulama sunucusunu kullanacağız. Tomcat in güncel sürümünü http://tomcat.apache.org/download-70.cgi web adresinden edinebilirsiniz.

Web tabanlı bir programın Tomcat altında çalıştırılabilmesi için belli bir dizin yapısına sahip olması gerekmektedir. Bu yapı aşağıdaki şekildedir:



Resim 6.7 Web dizini

- ***WEB-INF*** - Her web projesinde bu dizin yer alır. Classes, jsp, lib dizinlerini ve web.xml dosyasını ihtiva eder.
- ***WEB-INF\clasess*** - Proje içinde oluşturulan Java sınıfları derlenerek bu dizine kopyalanır. Tomcat için classes dizini classpath değerinin bir parçasıdır ve Java sınıflarını bu dizin içinde arar.
- ***WEB-INF\jsp*** - Oluşturulan JSP arayüzleri bu dizin içinde yer alır.
- ***WEB-INF\lib*** - Proje içinde kullanılan Jar dosyaları bu dizin içinde yer alır. Java sınıflarında import komutuyla kullanılan diğer sınıflar ya classes ya da lib dizininde bulunan bir Jar dosyasında yer almak zorundadır. Aksi taktirde ClassNotFoundException oluşacaktır.
- ***WEB-INF\web.xml*** - Her web projesinin bir web.xml konfigürasyon dosyası olmak zorundadır. Bu dosyanın yapısını daha sonra detaylı olarak inceleyeceğiz.
- ***index.jsp*** - Web tabanlı programın giriş sayfasıdır.

WEB-INF dizini ve index.jsp dosyası projenin ana dizininde yer alan web dizini içinde bulunmaktadır. Bu dizin web tabanlı programın ihtiyaç duyduğu gerekli tüm dosya ve dizinleri ihtiva eder.

Tomcat ile projemiz arasındaki bağlantıyı 3rdparty\apache-tomcat-7.0.53\conf\server.xml dosyası üzerinden oluşturabiliriz. Bu dosyayı aşağıdaki şekilde adapte ediyoruz:

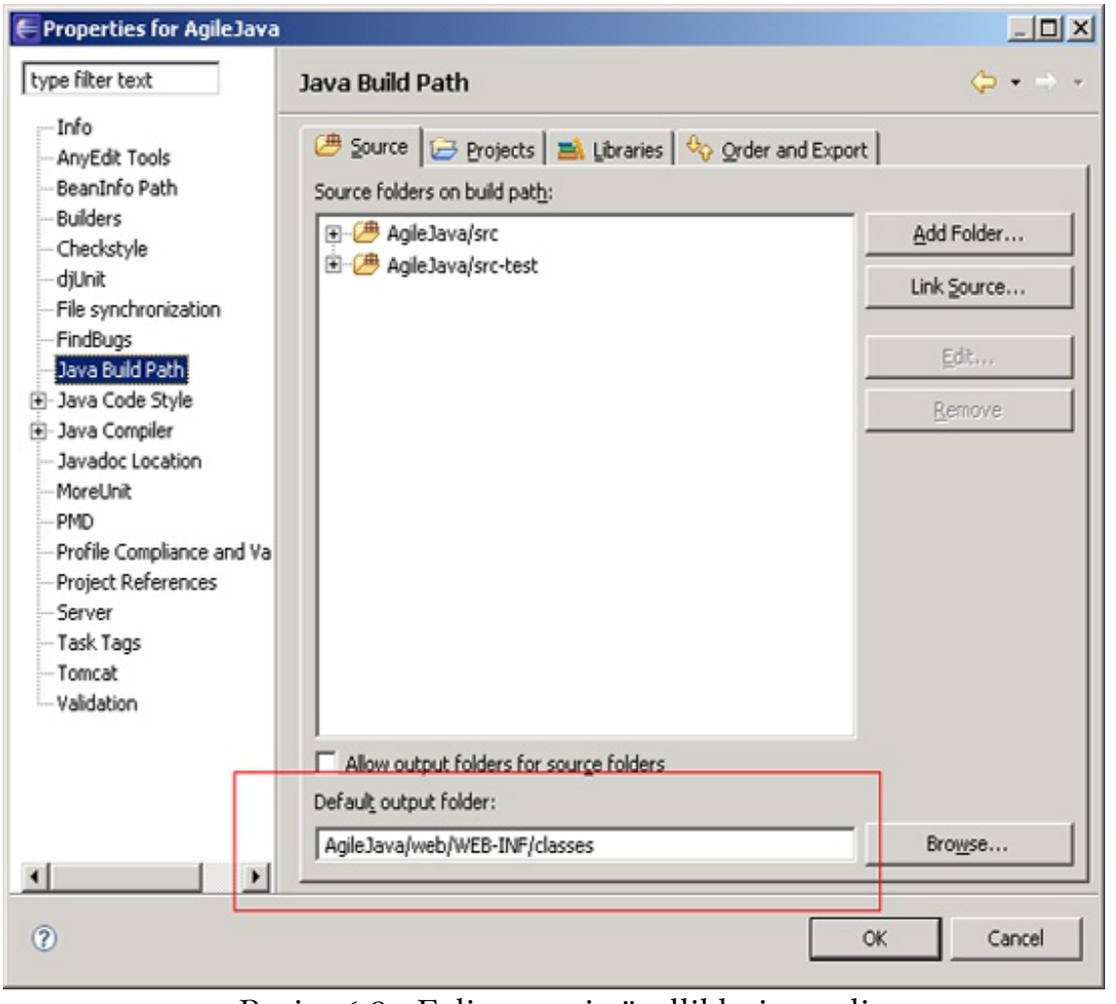
Kod 6.3 Tomcat server.xml konfigürasyon dosyası

```
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">

    <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
        type="org.apache.catalina.UserDatabase"
        description=""
        factory="org.apache.catalina.users.
                            MemoryUserDatabaseFactory"
        pathname="conf/tomcat-users.xml" />
    </GlobalNamingResources>

    <Service name="Catalina">
        <Connector port="80" />

        <Connector port="8009" protocol="AJP/1.3" />

        <Engine name="Catalina" defaultHost="localhost">
            <Realm
                className="org.apache.catalina.realm.
                            UserDatabaseRealm"
                resourceName="UserDatabase" />
            <Host name="localhost" appBase="webapps">

                <Context path="/" reloadable="true"
                    docBase="../../../web"
                    workDir="../../../work">
                </Context>

            </Host>
        </Engine>
    </Service>
</Server>
```

<context> elementi üzerinden Tomcat ile projemiz arasındaki bağı oluşturuyoruz. Tomcat web dizininde yer alan JSP sayfalarını ve diğer dosyaları

lokalize ederek çalıştırır.

Eclipse arka planda otomatik olarak üzerinde değişiklik yaptığımız Java sınıflarını derler (Project menüsünde Build Automatically opsiyonu). Tomcat tarafından bu değişikliklerin fark edilebilmesi için bu sınıfların web/WEB-INF/classes dizinine kopyalanması gerekmektedir. Eclipse tarafından Java sınıflarının web/WEB-INF/classes dizinine derlenebilmesi için projenin default output parametresinin değiştirilmesi gerekmektedir.



Resim 6.8 Eclipse proje özellikleri paneli

JSP sayfaları ve derlenen Java sınıfları doğru dizin içinde olduğu için Tomcat i çalıştırabiliriz. Bunu iki şekilde yapabiliriz:

- Ant üzerinden Tomcat çalıştırabilir / durdurabiliriz - Eclipse sunucu pluginini kullanarak, Tomcat i Eclipse içinde çalıştırabilir ve durdurabiliniriz.

Ant ile Tomcat sunucusunun nasıl kontrol edebileceğimizi gördükten sonra, bir sonraki bölümde Tomcat Eclipse sunucu pluginini yakından inceleyeceğiz.

Proje içinde rutin işleri otomatize ederek, zaman kazanabiliriz. Otomasyon için Ant birçok imkan sunmaktadır. Şimdi Ant yardımı ile Tomcat i nasıl kontrol edebileceğimizi görelim. Bu amaçla build.xml içinde tomcat-start ve tomcat-stop isminde iki yeni hedef tanımlıyoruz.

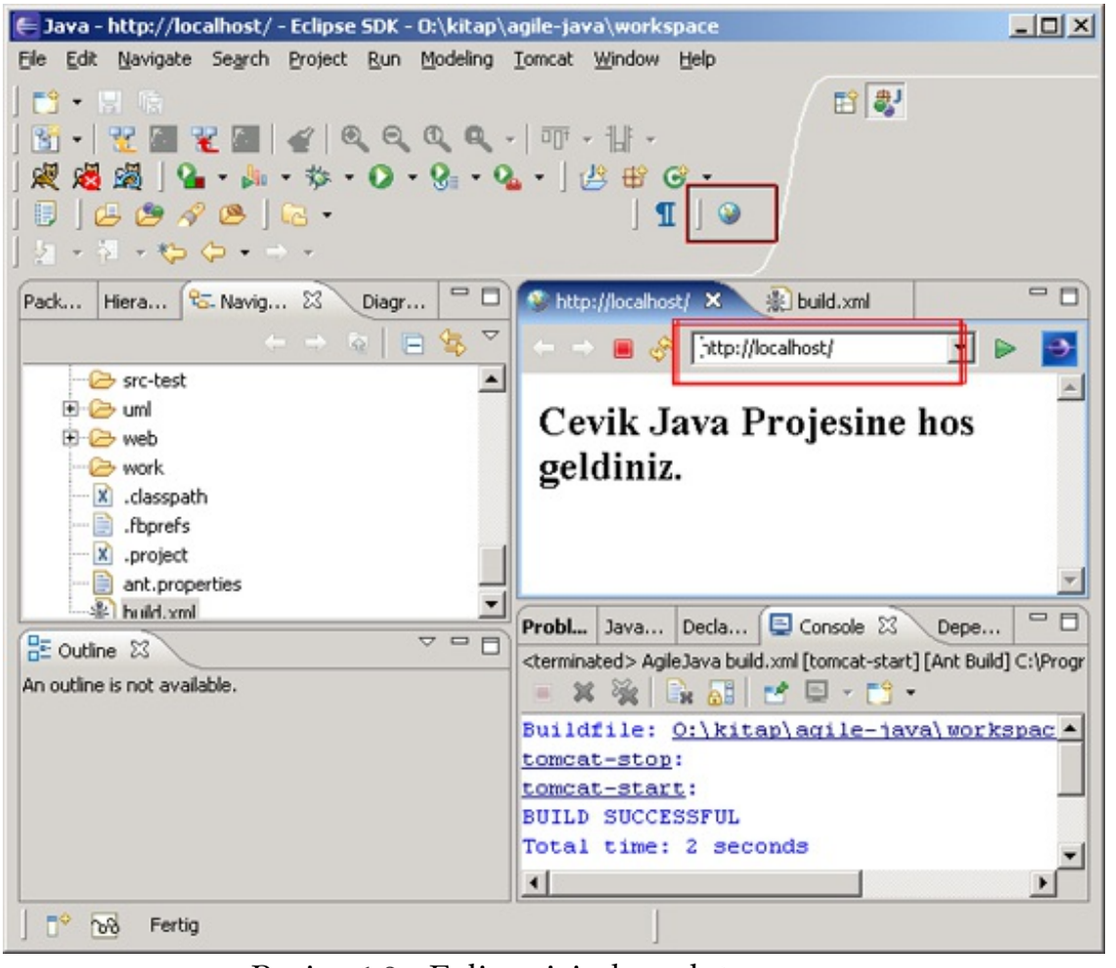
Kod 6.4 Tomcat Ant hedefleri

```
<target name="tomcat-start" depends="tomcat-stop">
    <java classname="org.apache.catalina.startup.Bootstrap"
                  failonerror="true" fork="true">
        <classpath path="${tomcat.home}/bin/bootstrap.jar:
                  ${tomcat.home}/bin/tomcat-juli.jar" />
        <jvmarg value="-Djava.util.logging.manager=org.apache.
                  juli.ClassLoaderLogManager" />
        <jvmarg value="-Djava.util.logging.config.file=${tomcat.home}
                  /conf/logging.properties" />
        <jvmarg value="-Dcatalina.home=${tomcat.home}" />
        <jvmarg value="-Dcatalina.base=${tomcat.home}" />
        <jvmarg value="-Djava.io.tmpdir=${tomcat.home}/temp" />
        <arg line="start" />
    </java>
</target>

<target name="tomcat-stop">
    <java classname="org.apache.catalina.startup.Bootstrap"
                  fork="true">
        <classpath path="${tomcat.home}/bin/bootstrap.jar:${tomcat.home}
                  /bin/tomcat-juli.jar" />
        <jvmarg value="-Dcatalina.home=${tomcat.home}" />
        <arg line="stop" />
    </java>
</target>
```

İki hedef içinde ${tomcat.home} değişkeni kullanılmaktadır. Bu değişkeni ant.properties içinde tanımlıyoruz. Bu değişken bir sonraki tabloda yer aldığı gibi ant.properties dosyasına eklenir.

Kod 6.5 ant.properties dosyası

```
#tomcat
tomcat.home=${basedir}/3rdparty/apache-tomcat-7.0.53
```



Resim 6.9 Eclipse içinde web tarayıcısı

Console panelinde görüldüğü gibi ilk önce tomcat-stop akabinde tomcat-start hedefleri (target) çalıştırılmaktadır. Tomcat i ant aracılığıyla çalıştırdıktan sonra Eclipse bünyesinde bulunan web tarayıcısıyla http://localhost adresine bağlanarak, projemizin ilk sayfasını görebiliriz.

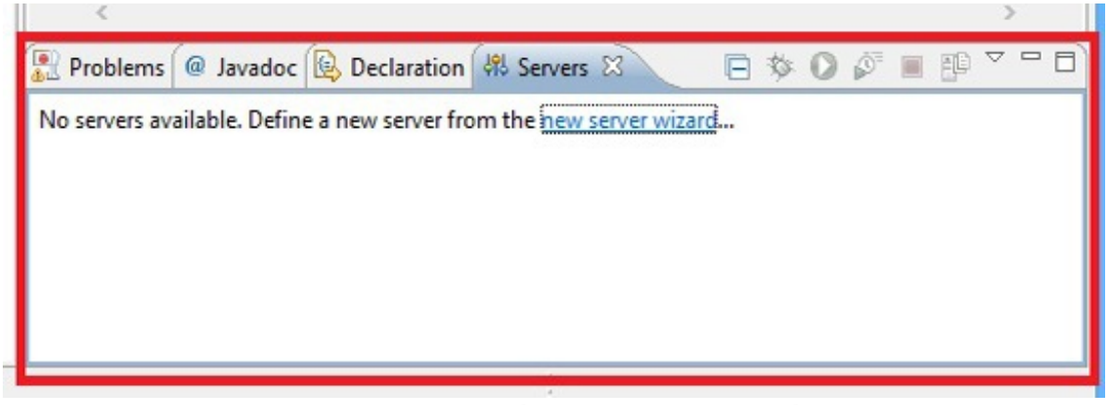
Şimdiye kadar yaptığımız işlemleri özetleyecek olursak:

- Eclipse altında CevikJava isminde bir proje oluşturduk.
- Ant ile projeyi yapılandırmak üzere compile ve clean hedeflerini (target) tanımladık. Bu andan itibaren Ant ile her zaman otomatik yapılandırma işlemini kullanabiliriz. Ant derleme, kopyalama ve dizin oluşturma işlemlerini düzenlediğimiz şekilde yerine getirecektir. Bu otomasyon bize büyük ölçüde zaman kazandıracaktır. Daha sonra yakından inceleyeceğimiz JUnit testlerinin de Ant ile nasıl entegre edilebileceğini göreceğiz.
- Tomcat kurulumunu gerçekleştirdik. Ant üzerinden Tomcat uygulama sunucusunu kontrol edebilecek hedefler (tomcat-start, tomcat-stop) oluşturduk. Artık Ant ile Tomcat uygulama sunucusnu çalıştırabilir / durdurabiliriz.

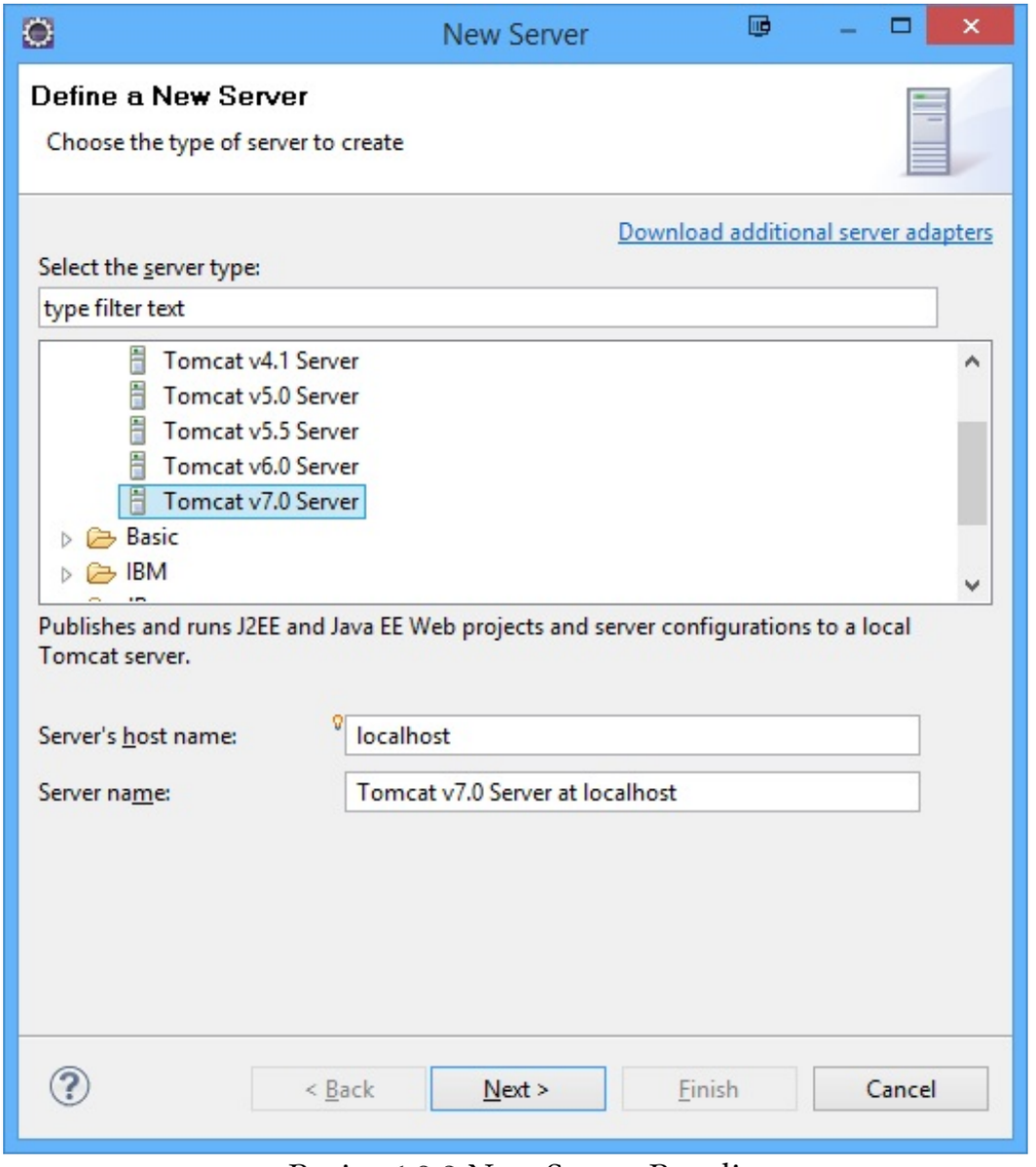
Yukarıda sıraladığımız işlemleri, Eclipse dışına çıkmak zorunda kalmadan gerçekleştirebiliriz. Bu bize tek bir program (Eclipse) aracılığıyla birçok işlemi yapabilme avantajını sunmaktadır. Bu işlemlerin ardından Eclipse kelimenin tam anlamıyla bir IDE (Integrated Development Environment) yani entegre yazılım platformu haline gelmiştir.

Tomcat Eclipse Entegrasyonu

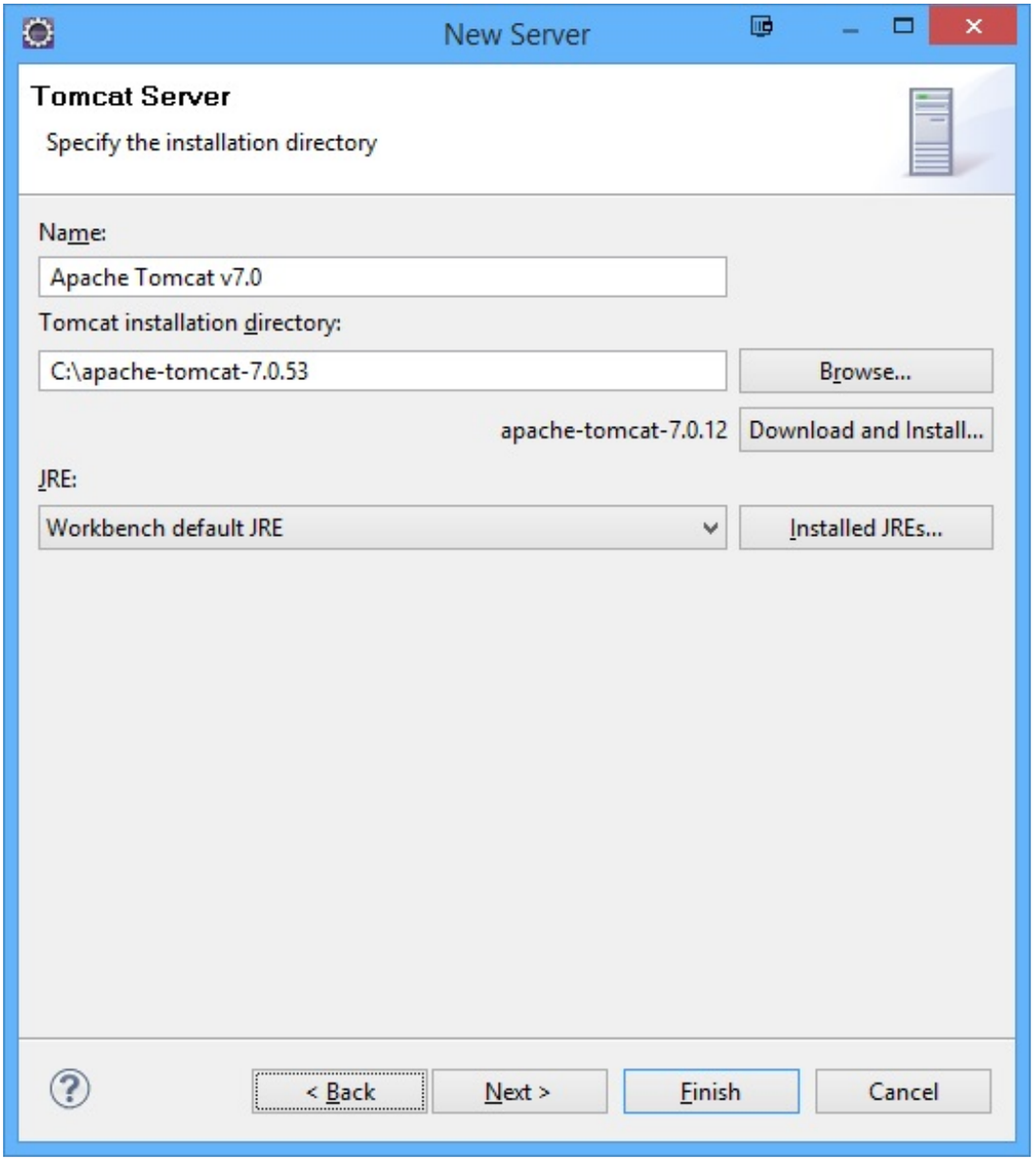
Tomcat 7 sürümünü Eclipse altında çalıştırılabilir hale getirebiliriz. Bu amaçla Window > Show View menüsünden Server kaydını seçmemiz gerekiyor. Bu aşağıda görülen Server panelinin oluşmasını sağlayacaktır. Buradan "new server wizzard" linkine tıklayarak, Tomcat sunucumuzu oluşturabiliriz.



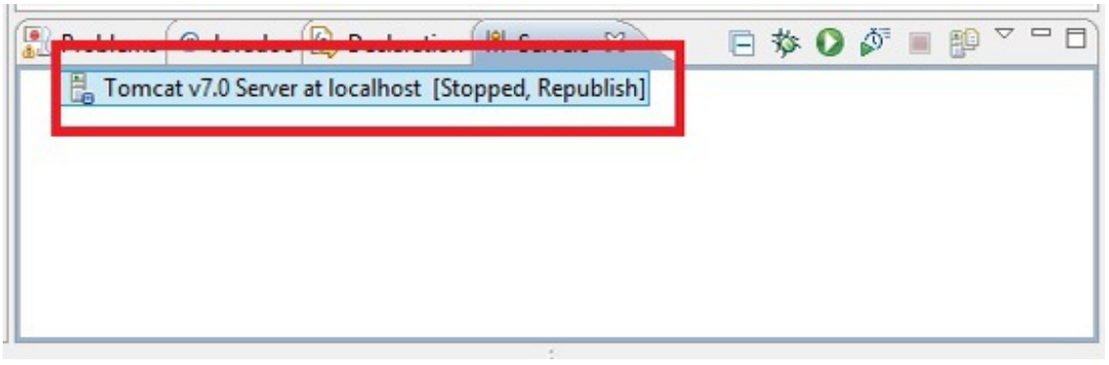
Resim 6.9.1 Eclipse Servers Paneli



Resim 6.9.2 New Server Paneli



Resim 6.9.3 New Server Paneli



Resim 6.9.4 New Server Paneli

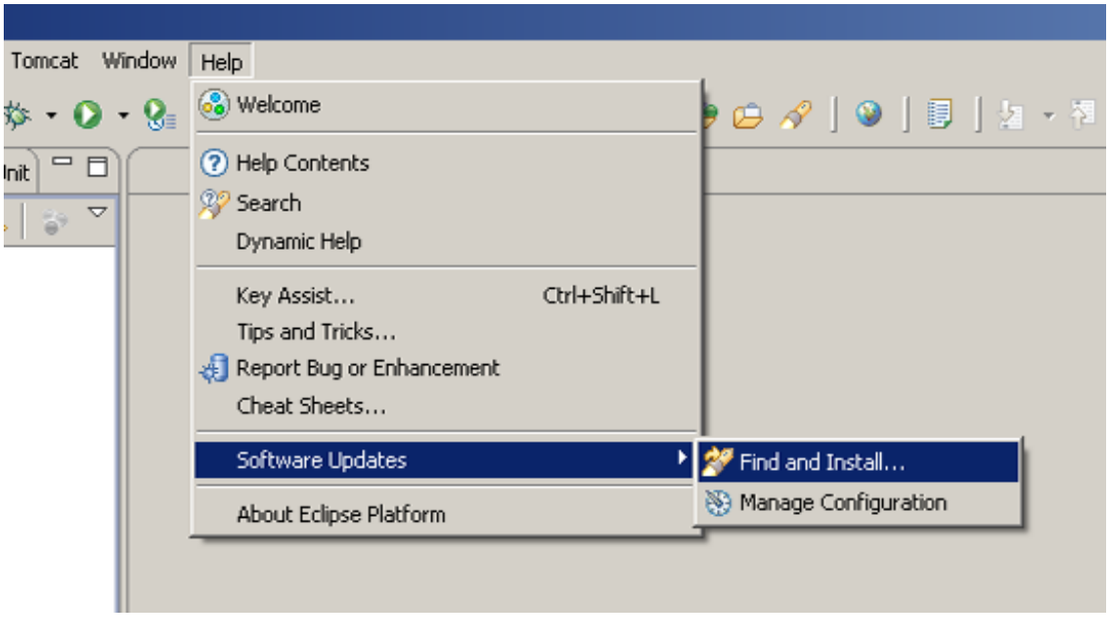
Bu işlemlerin ardından oluşturduğumuz bu yeni sunucu kaydının üstüne tıklayarak sunucuyu çalıştırabiliriz.

Subversion / Subclipse

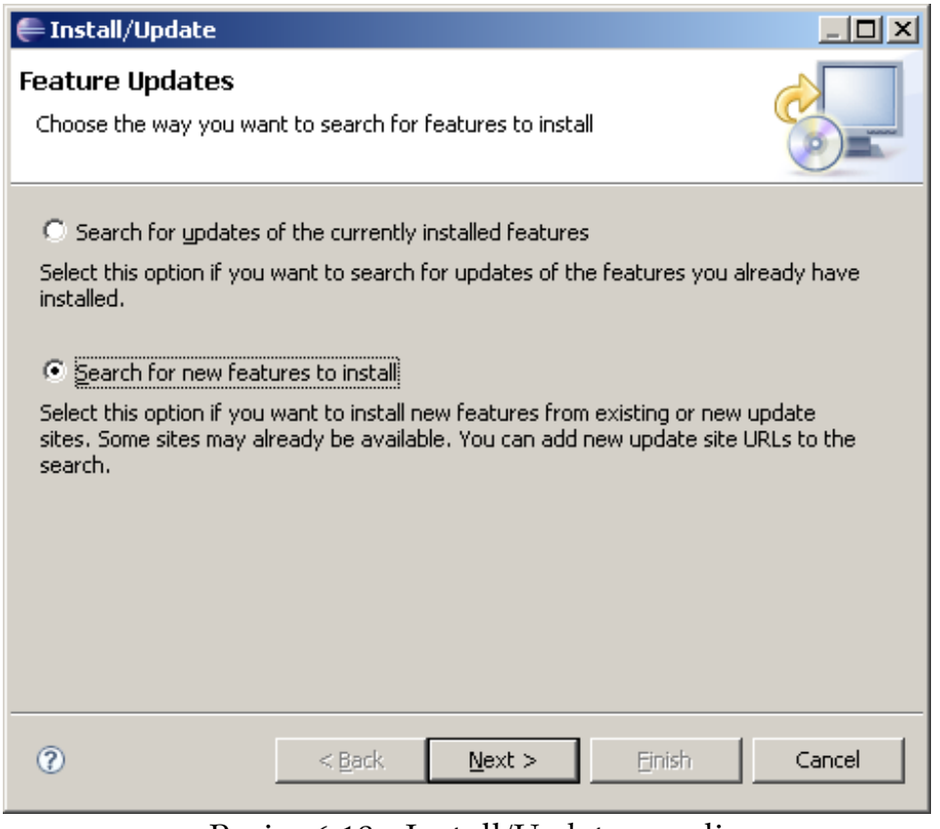
Subversion açık kaynaklı bir versiyon kontrol programıdır. Bu konudaki detaylı bilgiyi bu kitabın Subversion ile versiyon kontrolü bölümünde bulabilirsiniz.

Bu bölümde Eclipse içinde plugin olarak kullanabileceğimiz bir Subversion client programından bahsetmek istiyorum: Subclipse.

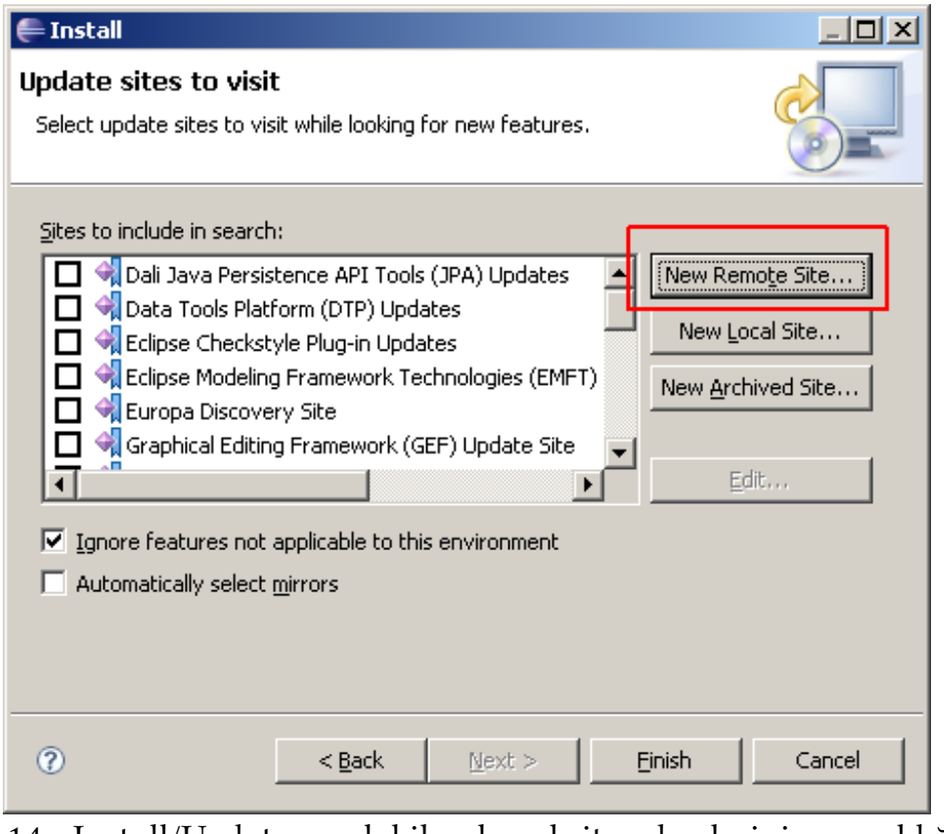
Bu programı http://subclipse.tigris.org adresinden temin edebilirsiniz. Eclipse altında kurulum Help > Software Updates menüsünden gerçekleştirilebilir.



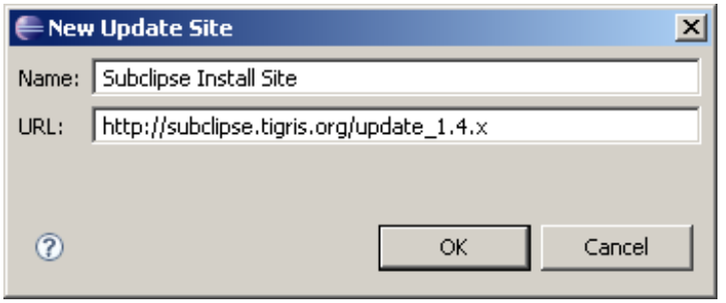
Resim 6.12 Eclipse Software Updates menüsü



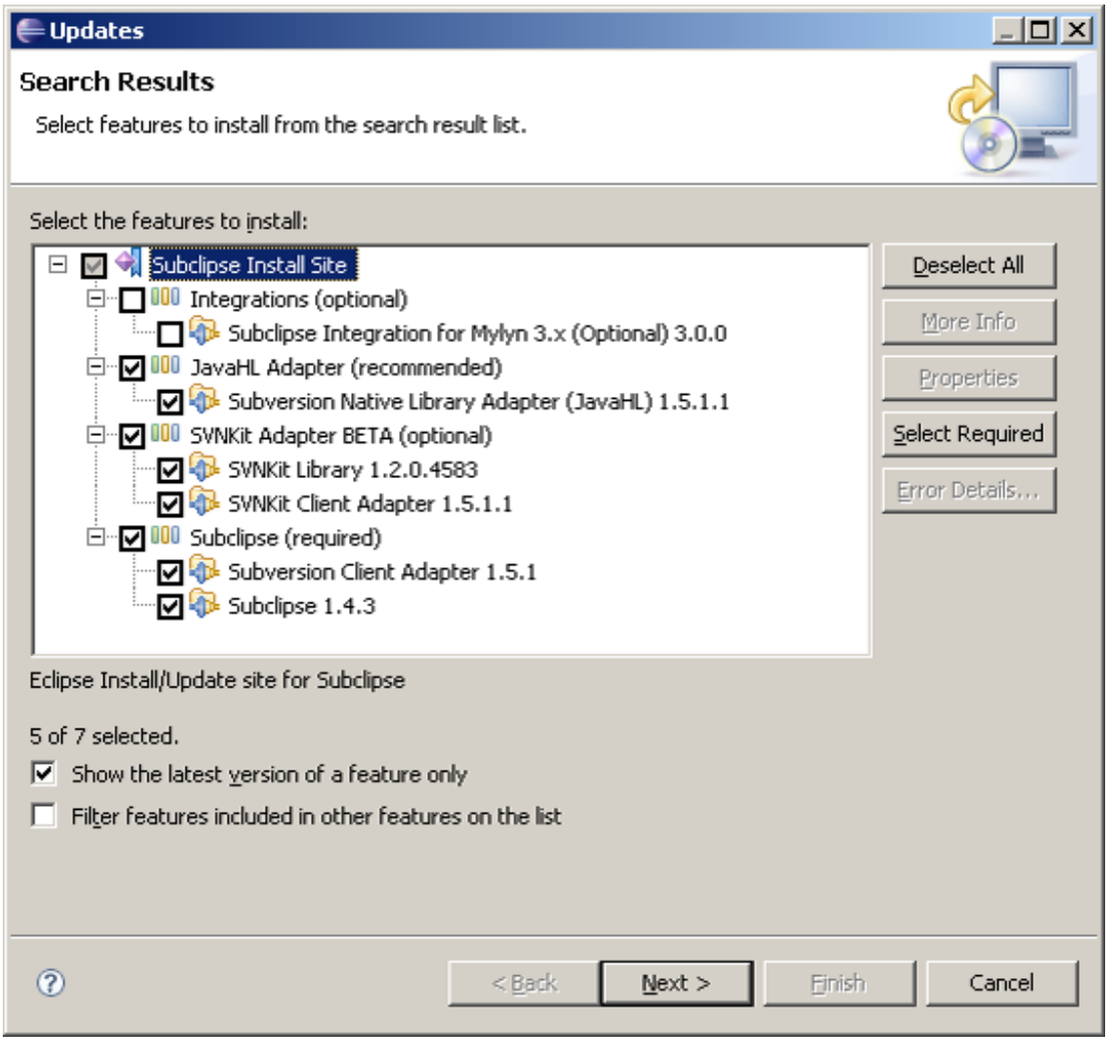
Resim 6.13 Install/Update paneli



Resim 6.14 Install/Update yapılabilecek website adreslerinin yer aldığı panel

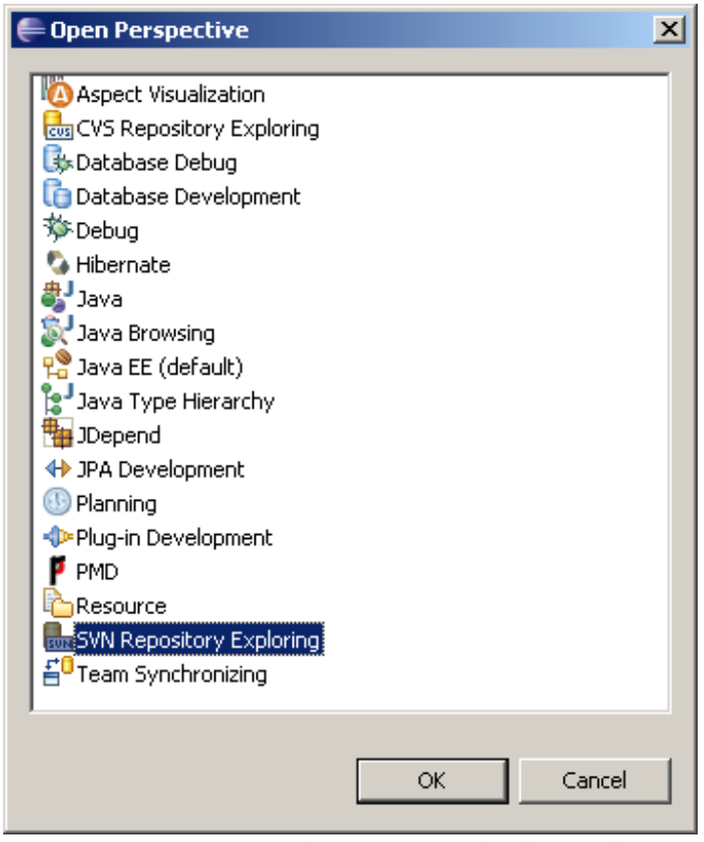


Resim 6.15 Subclipse install adresi



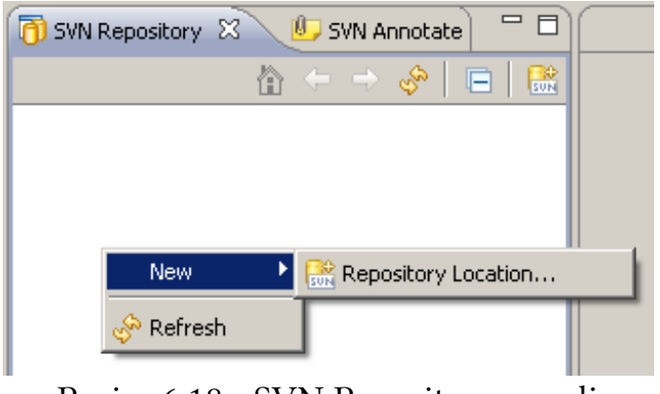
Resim 6.16 Subclipse kurulum opsiyonları

Bu işlemlerin ardından Subclipse en güncel versiyonunda Eclipse altında plugin olarak kurulacaktır. Subclipse aracılığıyla bir Subversion sunucusuna bağlanmak için Eclipse altında SVN perspektifine geçmemiz gerekiyor.

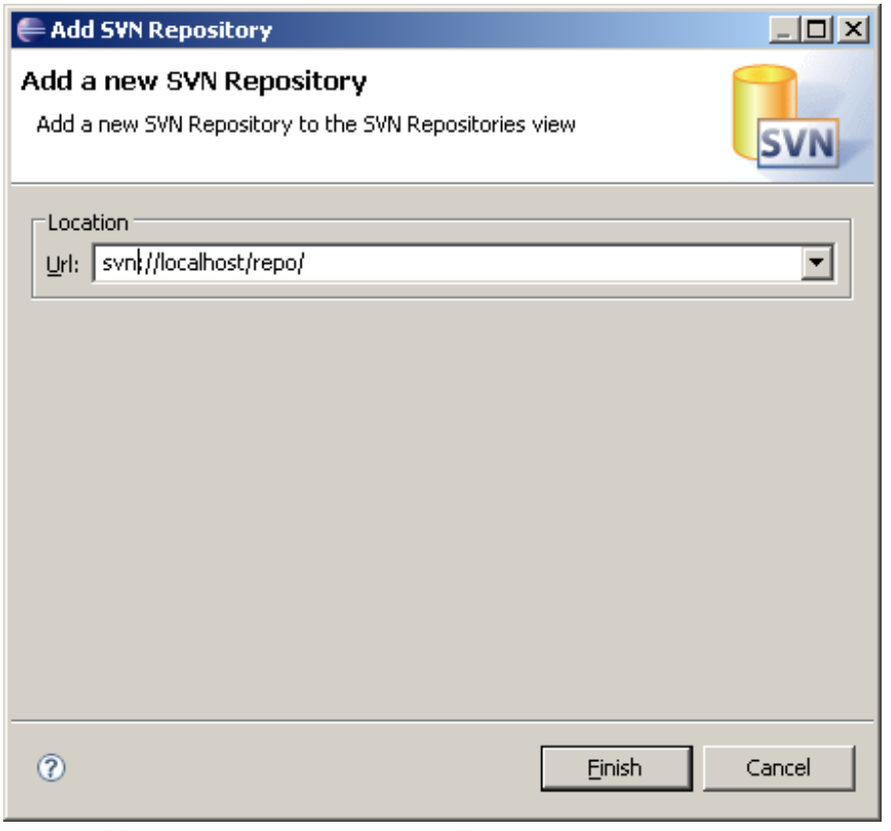


Resim 6.17 Eclipse perspektif paneli

Bu işlemin ardından sol üst bölümde SVN Repository paneli yer alacaktır. Bu panel üzerinden bağlanmak ve kod edinmek istediğimiz Subversion sunucusunu tanımlayabiliriz.



Resim 6.18 SVN Repository paneli



Resim 6.19 Kullanmak istediğimiz Subversion sunucusnun lokasyonu

Resim 6.19 da yer aldığı gibi kullanmak istediğimiz Subversion sunucusunun adresini (svn://localhost/repo) giriyoruz. Bu işlemin ardından bağlantı kurmuş olduğumuz Subversion sunucusunda yer alan projelerin listesini edinebilir ve istediğimiz projeyi kendi çalışma alanımıza alabiliriz (checkout işlemi).

JUnit

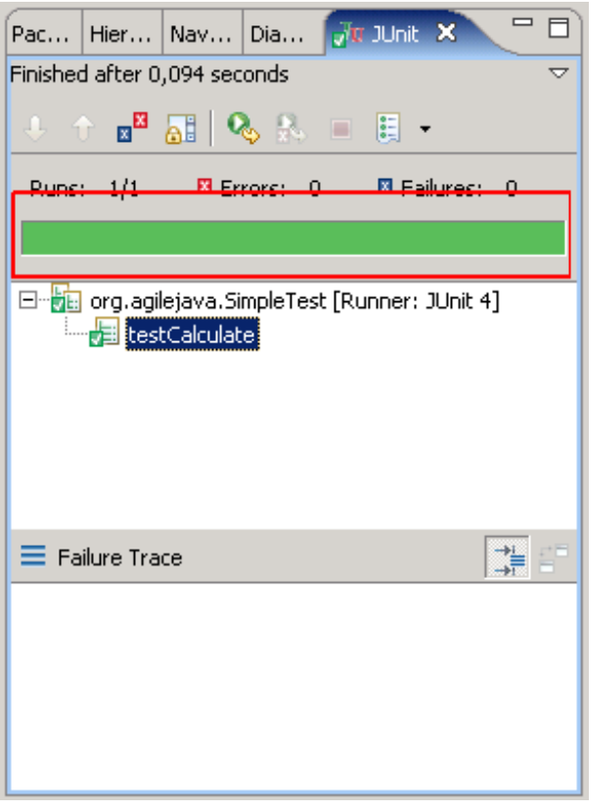
Karmaşık yapıdaki bir sistemi anlamak, geliştirebilmek ve yönetebilmek için böl ve yönet (divide and conquer) prensibi kullanılır. Birçok mühendislik disiplininde kullanılan bu prensip ile bir bütün parçalarına ayrılarak, izole edilen parçalar üzerinde gerekli işlemler gerçekleştirilir. Bir yap boz oyununda olduğu gibi resmin küçük parçaları tek başlarına anlam taşımasalar da, bir araya getirildiklerinde resmin oluşmasında üstlerine düşen görevi yerine getirirler ve böylece anlam kazanırlar.

Bilgisayar programları yapı itibariyle karmaşık olmaya eğilim gösterirler. Yazılım esnasında böl ve yönet prensibi uygulanarak, karmaşaya hakim olunabilir. Programın bütününü oluşturan modüller üstlerine düşen görevleri yerine getirdikleri sürece program hatasız çalışacaktır.

Bir modülün (bu tek bir Java sınıfı bile olabilir) doğru çalıştığını nasıl anlayabiliriz? Modülü test ederek... Peki bir modül nasıl test edilir? Bunun birçok alternatif yolu var. Örneğin program çalışır hale getirilir ve kullanıcı arayüzü aracılığıyla bir kullanıcı programı test eder. Kullanıcı oluşan hataları programcı ekibe bildirir. Onlarca kullanıcı arayüzü ve modülden oluşan bir programı bu şekilde test etmek çok zaman alıcı bir uğraş olacaktır. Diğer bir yöntem ise birim testleri ismini taşıyan modül testleri oluşturmaktır. Birim testleri her modül için ihtiva ettiği fonksiyonları test eden Java sınıflarıdır. Programcı bir modülü oluştururken beraberinde birim testlerini de hazırlar. Kitabın diğer bir bölümünde detaylı olarak inceleyeceğiz test güdümlü programcılıkta (TDD = Test Driven Development) programcı önce birim testlerini oluşturur ve buradan yola çıkarak gerekli modülü programlar. Bu ilk etapta tuhaf bir yazılım metodu gibi görünse de, TDD tarzı çalışmanın ne kadar sağlıklı sonuçlar doğurduğunu yakından inceleyeceğiz.

Birim testleri oluşturabilmek için Junit test çatısından faydalanabiliriz. Junit çatısını http://www.junit.org adresinden temin edebilirsiniz.

Eclipse bünyesinde birim testlerini çalıştırmak için bir plugin bulunmaktadır. Test sınıfı Run As / JUnit Test menüsü üzerinden çalıştırılabilir.



Resim 6.20 Eclipse JUnit Plugin

Birim testlerinin oluşturulması ve test güdümlü programlama hakkında detaylı bilgiyi kitabın dokuzuncu ve onuncu bölümlerde bulabilirsiniz.

Ant JUnit Entegrasyonu

Birim testlerini Ant ile entegre ederek otomatik olarak çalıştırılmalarını sağlayabiliriz. Ant ile proje yapılandırma sürecinde şöyle bir senaryo düşünülebilir: Tüm Java sınıflarını derledikten sonra birim testleri çalıştırılarak tüm sınıflar test edilir. Testlerde hata oluşması durumunda yapılandırma işlemi Ant tarafından durdurulacaktır. Bu durumda hatanın oluştuğu sınıf lokalize edilerek hata giderilir ve tekrar proje yapılandırılır. Tamamen otomatize edilmiş bu süreçte Ant ve birim test entegrasyonunun sağladığı avantaj görülmektedir. Programcı yaptığı her değişiklikten sonra birim testlerini Ant ile çalıştırarak, diğer modüller ile etkileşimi kontrol edebilir.

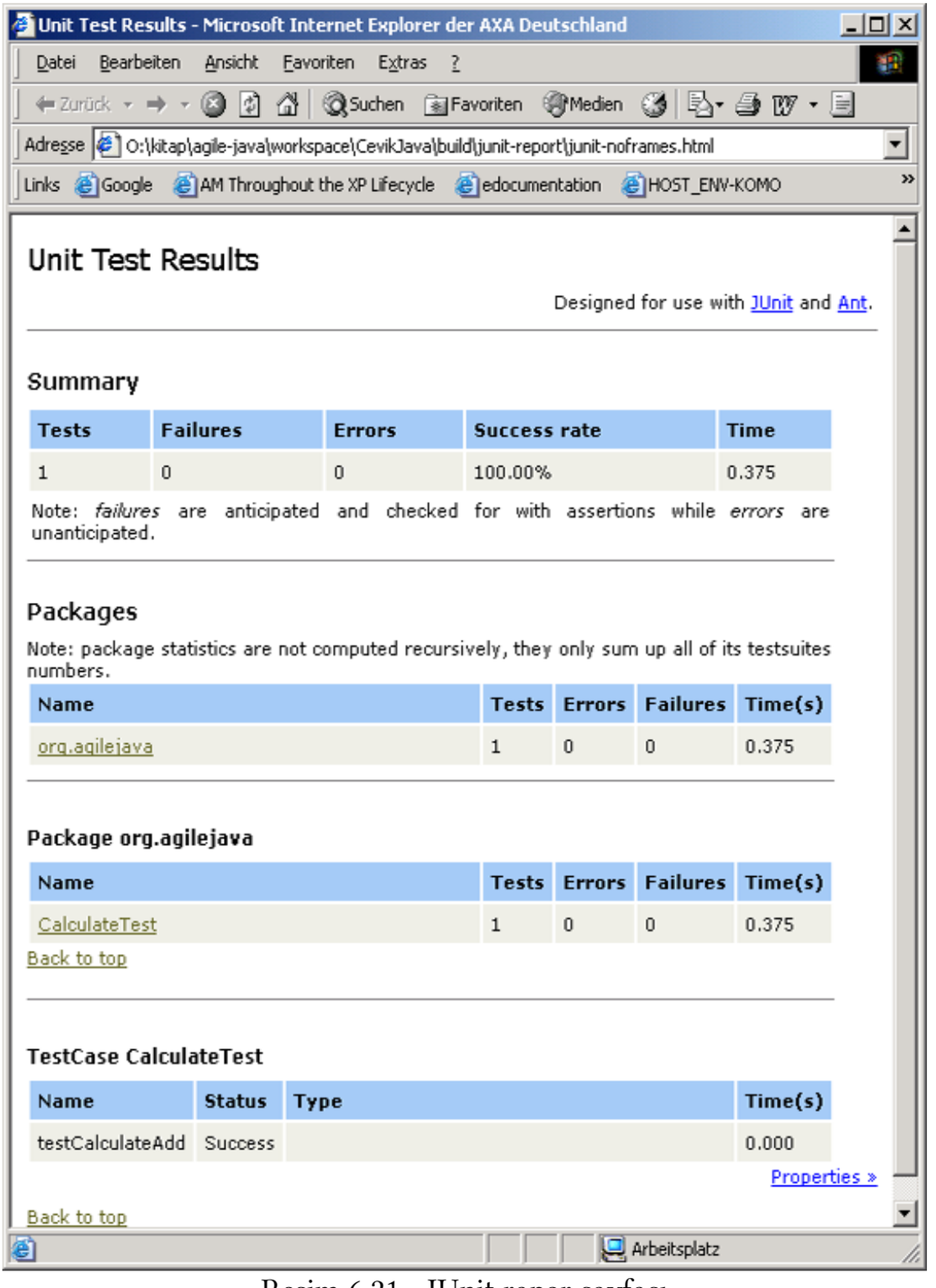
Ant JUnit entegrasyonun sağlamak için run-junit isminde bir hedef (target) tanımlıyoruz.

Kod 6.6 Ant JUnit entegrasyonu

```
<target name="run-junit" depends="compile">

    <junit fork="false" failureproperty="tests.failed"
        showoutput="true" printsummary="yes"
        haltonfailure="yes">

        <classpath refid="compile.classpath" />
        <classpath path="${build.web-inf.classes.dir}"/>

        <test name="org.cevikjava.CalculateTest"
            haltonfailure="yes"

            outfile="build/junit-result">
            <formatter type="xml" />
        </test>
    </junit>

    <fail if="tests.failed">
        **************************
        JUnit testlerinde hata olustu.
        Lütfen kontrol ediniz.
        **************************
    </fail>
```

```
    <junitreport todir="${junit.report.dir}">
        <fileset dir="${build.dir}">
            <include name="junit-result.xml" />
        </fileset>
        <report format="noframes"
            todir="${junit.report.dir}" />
    </junitreport>
</target>
```

<junit> komutu ile JUnit testleri için çalışma ortamı oluşturulur. <test> komutu ile JUnit test sınıfı tanımlanır. Örneğimizde org.cevikjava.CalculateTest sınıfı kullanılmaktadır. Hata oluşması durumunda <junit> komutu tarafından tests.failed isminde bir değişken oluşturulur. <fail> komutu ile bu değerin true olup olmadığı kontrol edilir. Eğer bu değişken true değerine sahip ise Ant tarafından yapılandırma işlemi durdurulur. Örneğimizde <test> komutu test sonuçlarını (outfile) bir xml dosyasına (build/junit-result.xml) aktarmaktadır. Son bölümde yer alan <junitreport> komutu ile junit-result.xml dosyasında bulunan değerler kullanılarak bir HTML rapor dosyası oluşturulur. Bu dosya aşağıdaki yapıya sahiptir.

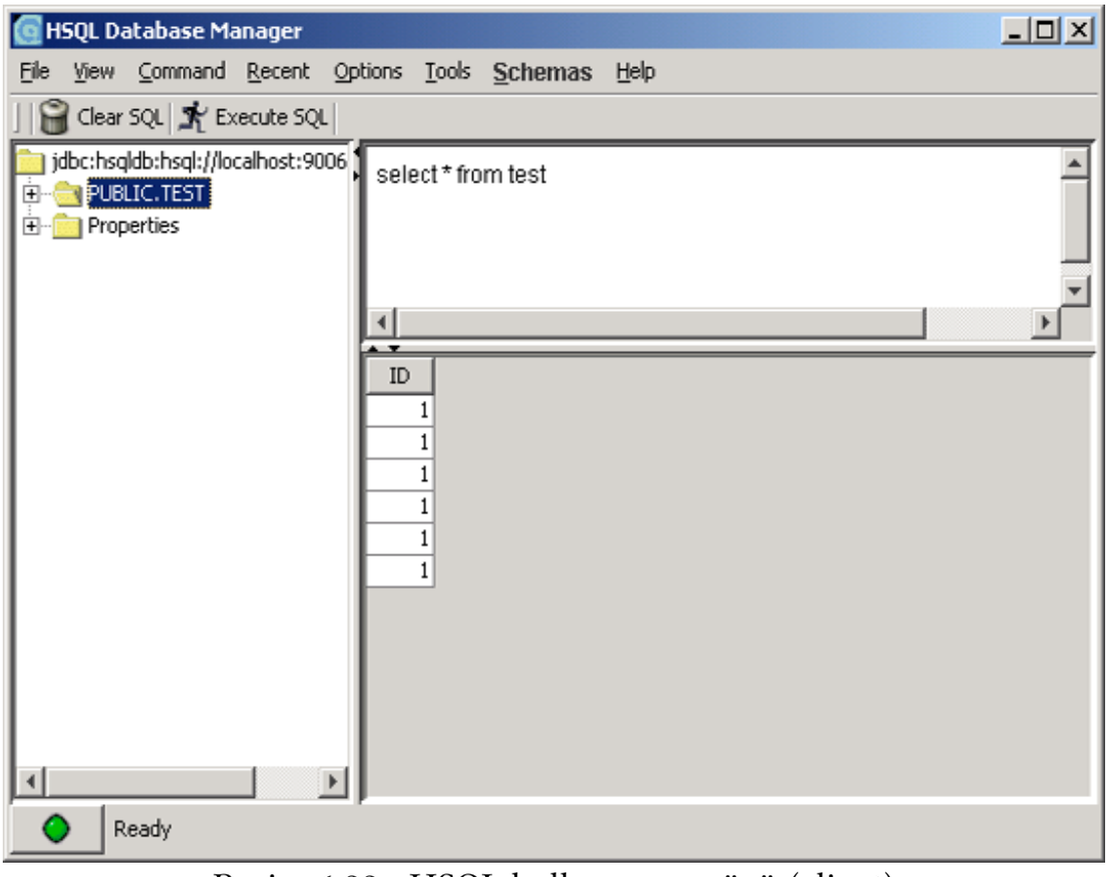


Resim 6.21 JUnit rapor sayfası

JUnit testlerinin Ant tarafından çalıştırılabilmeleri için junit.jar dosyasının ant/lib dizinine eklenmesi gerekmektedir.

HSQL Veri Tabanı

Yazılım esnasında her programcının kullanabileceği bir veri tabanına ihtiyaç duyulmaktadır. Bu MySQL ya da Postgres gibi açık kaynaklı bir veri tabanı sistemi olabilir. Programcı veri tabanını kendi bilgisayarı üzerine kurabilir ya da başka bir sunucu üzerinde bulunan veri tabanını kullanabilir. Veri tabanı sistemleri bilgisayar üzerinde az denilemeyecek oranda sistem kaynaklarını kullanırlar. Ayrıca bu sistemlerin durdurma ve çalıştırma işlemi zaman alıcı olabilir. Çevik çalışma tarzını desteklemek için hafif bir veri tabanı kullanılması gerekmektedir. HSQL Java dilinde yazılmış, tek bir Jar dosyasında yer bulan SQL uyumlu bir veri tabanıdır. Güncel sürümünü http://hsqldb.org/ adresinden temin edebilirsiniz.



Resim 6.22 HSQL kullanıcı arayüzü (client)

Bir önceki resimde görüldüğü gibi HSQL bir kullanıcı arayüzüne sahiptir. Standart SQL komutlar kullanılarak istenilen işlemler yapılabilir.

Ant HSQL Entegrasyonu

Amacımız yazılım esnasında yüksek derecede otomasyon sağlamak olmalıdır. Rutin işler otomatize edilmediği taktirde, yazılım sürecinde fazla zaman kaybedilmesine yol açabilir. Ant bu konuda bize birçok imkan sunmaktadır.

HSQL veri tabanını Ant buil.xml bünyesinde hsql-start, hsql-stop hedefleri (target) tanımlayarak kontrol edebiliriz. Böyle bir entegrasyonun bize ne gibi bir faydası olabilir? İnceleyelim:

- Sadece bir noktadan (build.xml) veri tabanını kontrol edebilir ve kullanabiliriz.
- Proje yapılandırma işleminde otomatik olarak önce veri tabanı çalıştırılır, gerekli veriler veri tabanına eklenir, Java sınıflar derlenir, JUnit testleri çalıştırılır, veri tabanında veriler silinir ve veri tabanı durdurulabilir. Bu şekilde bir senaryo entegrasyon testi olarak düşünülebilir. Tüm sistem sadece bir Ant hedefi (run-integration, daha sonra bu hedefi oluşturacağız) ile çalıştırılarak test edilebilir.

Ant HSQL entegrasyonu kod 6.7 de yer almaktadır.

Kod 6.7 Ant HSQL entegrasyonu

```
<target name="hsql-start">
<java fork="true"
    classname="${hclass}"
    classpath="${hjar}"
    args="${hfile} -dbname.0 ${halias} -port
                            ${hport}"
    spawn="true"
    newenvironment="true"
    failonerror="false" />
</target>

<!-- HSQL serveri stop etmeden tekrar
    calistirmamiz gerekiyor. Sorun: Windows
    altında sorun yok, ama build.xml linux
    altında calistigi zaman, calismayan bir
    HSQL serveri stop etmeye kalkinca,server
     calismadigi için SocketException geliyor
     ve build failed oluyor. Bunu önlemek için
     önce serveri calistiriyoruz ve stop
     ediyoruz. Eger server calisiyorsa,
     tekrar calistirmak istememiz sorun degil! -->
<target name="hsql-stop">

    <antcall target="hsql-start">
    </antcall>

    <concat destfile="temp.bsh">
<![CDATA[
```

```
    import java.sql.*;

    Class.forName("org.hsqldb.jdbcDriver");
    String url = "jdbc:hsqldb:hsql://127.0.0.1:9006/" + bsh.args[0];
    Connection con = DriverManager.getConnection(url, "sa", "");
    String sql = "SHUTDOWN";
    Statement stmt = con.createStatement();
    stmt.executeUpdate(sql);
    stmt.close();
]]>
    </concat>

    <java classname="bsh.Interpreter"
        fork="true"
        failonerror="false"
        resultproperty="hsql-stop-result">
        <classpath refid="compile.classpath" />
        <arg line="temp.bsh ${halias}" />
    </java>

    <echo>${hsql-stop-result}</echo>
</target>


<target name="hsql-start-client">
    <java fork="true"
        classpath="${hjar}"
        classname="org.hsqldb.util.DatabaseManagerSwing" />
</target>


<target name="hsql-start-sqltool">
    <java fork="true"
        classpath="${hjar}"
        classname="org.hsqldb.util.SqlTool"
        args="localhost-sa" />
</target>
```

build.xml içinde HSQL entegrasyonu için gerekli hedefleri tanımladık. Bu hedefler içinde kullanılan değişkenleri ant.properties dosyasına ekliyoruz.

Kod 6.8 ant.properties dosyası

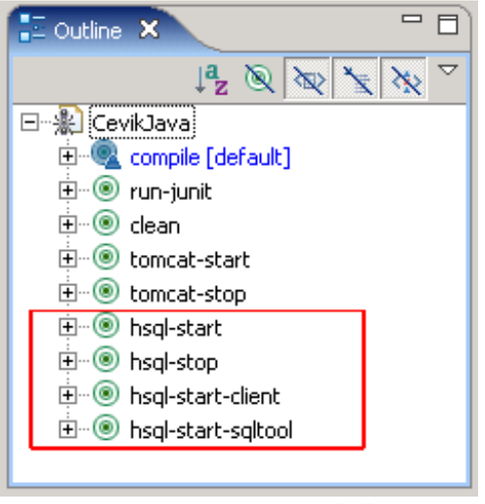
```
#ant.properties - hsql degiskenleri
hjar=web/WEB-INF/lib/hsqldb.jar
hclass=org.hsqldb.Server
hfile=-database.0 hsql/data/
halias=cevikjava
```

```
hport=9006
```

Ant bu tanımlamalar yardımıyla classpath içinde yer alan hsqldb.jar dosyasını lokalize ederek, org.hsqldb.Server sınıfını çalıştıracaktır. Bu ayarlar ile HSQL sunucusu localhost 9006 nolu portta aktif hale gelecektir.

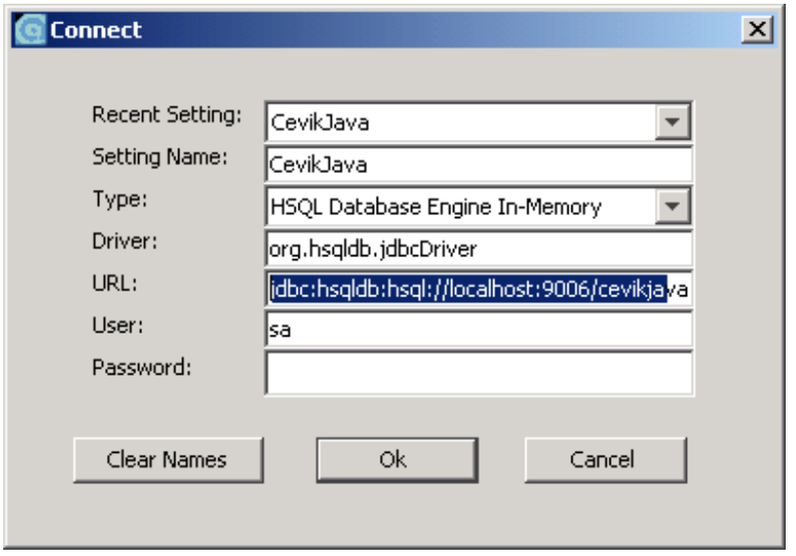
build.xml bünyesinde dört yeni hedef tanımladık. Bunlar:

- ***hsql-start*** - Çalıştığımız bilgisayar üzerinde HSQL veri tabanı sunucusunu çalıştırır.
- ***hsql-stop*** - Çalışır durumda olan bir HSQL sunucusunu durdurur.
- ***hsql-start-client*** - Daha önce gördüğümüz HSQL Database Manager programını çalıştırır.
- ***hsql-start-sqltool*** - Console tabanlı bir arayüz programını çalıştırır.



Resim 6.23 Ant task listesi

hsql-start hedefiyle HSQL sunucusunu çalıştırdıktan sonra, hsql-start-client hedefiyle Database Manager programını çalıştırabiliriz. HSQL sunucusuna bağlantı kurabilmek için bir sonraki resimde yer alan değerleri kullanabiliriz.



Resim 6.24 HSQL server baglantı ayarları

DBUnit

Birim (unit) testleri ile Java sınıfları ve ihtiva ettikleri metotlar test edilir. Test esnasında bu sınıflar sistemin diğer bölümlerinden tamamen izole edilmiştir. Sınıfın bağımlı olduğu diğer sınıflar Mock (sekizinci bölümde inceleyeceğiz) nesneler kullanılarak simüle edilirler. Her birim testi sınıfın belli bir davranış biçimini (metodunu) test eder.

Bazı Java sınıfları veri tabanı ile olan bağlantıyı gerçekleştirmek ve veri tabanından veri edinmek için vardır. Bu sınıflar için de JUnit testleri hazırlanır. Bu testlere entegrasyon testleri adı verilir, çünkü test esnasında Java sınıfı yanı sıra veri tabanı da test içinde kullanılır. Böylece iki sistem arasındaki entegrasyon test edilmiş olur.

Veri tabanı ve program arasındaki işlevi birim testleri ile test edebilmek için veri tabanı sisteminin belirli verilerle ile yüklenmiş olması gerekmektedir. Her test başlangıcında veri tabanına bu veriler yüklenir ve veri tabanı tanımlanmış bir seviyeye getirilir. Bu testler için önemlidir, çünkü testler için çıkış noktasının aynı olması gerekmektedir. Aynı şartlar altında çalışmayan testler değişik sonuçlar doğurabilir. Bu yüzden veri tabanının her test başlangıcında tanımlanmış veri seviyesine getirilmesi gerekmektedir. Bu işlem için DBUnit kullanılır. JUnit gibi DBUnit bir test çatısıdır. DBUnit in güncel sürümünü http://www.dbunit.org adresinden edinebilirsiniz.

Test için gerekli verileri bir XML dosyasında tanımlıyoruz.

Kod 6.9 dbunit-dataset.xml dosyası

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

    <customer id="1"
        email="test@localhost.com"
        password="testtest"
        birthday="1974-07-18"
        firstname="Oezcan"
        name="Acar"
        sex="1"
        created="2007-12-12"
        website="1"
        activated="0"
        approved="0"
        banned="0"
        activationcode="123456"
        language="de" />

    <customer id="2"
        email="test@localhost.com"
        password="testtest"
        birthday="1974-07-18"
        firstname="Ahmet"
        name="Yildirim"
        sex="1"
        created="2007-12-12"
        website="1"
        activated="0"
        approved="0"
        banned="0"
        activationcode="123456"
        language="de" />
</dataset>
```

Bir önceki tabloda dbunit-dataset.xml ismini taşıyan dosya yer almaktadır. DBUnit bu dosyada tanımlanmış olan verileri test öncesi veri tabanına yükler. Akabinde testler çalıştırılarak bu veriler yardımı ile modüller test edilir.

dbunit-dataset.xml dosyasında <customer> isminde bir etiket (tag) tanımlanmıştır. Bu etiket kullanılan veri tabanındaki bir tablonun ismidir. Customer tablosunda email, password ve birthday gibi kolonlar yer almaktadır. DBUnit <customer/> etiketi içinde tanımlanmış olan verileri tabloda bir kayıt (record) oluşturacak şekilde customer isimli veri tabanı tablosuna ekler. Yukarıdaki örnekte customer tablosu için iki record oluşturulur. Bu şema takip

edilerek veri tabanında bulunan tablolar için istenilen adette kayıt oluşturulabilir.

Aşağıdaki şekilde bir CustomerDao sınıfı için nasıl bir test konsepti uygulanabilir?

Kod 6.10 CustomerDao sınıfı

```
package org.cevikjava.samples;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * Customer tablosundan müsteri
 * bilgilerini edinmek için kullanilan
 * sinif.
 *
 * @author Oezcan Acar
 *
 */
public class CustomerDao
{
    public CustomerVo getCustomer(long id)
                    throws Exception
    {
        Connection con = DBManager.instance()
                .getConnection();
        PreparedStatement pstmt =  null;
        ResultSet rs = null;

        CustomerVo vo = new CustomerVo();
        try
        {
            pstmt = con.prepareStatement("" +
                    "select name, firstname " +
                    "firstname from customer " +
                    "where id=?");
            pstmt.setLong(1, id);
            rs = pstmt.executeQuery();

            if(rs.next())
            {
                vo.setName(rs.getString(1));
                vo.setFirstname(rs.getString(2));
            }
            rs.close();
```

```
            pstmt.close();
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.close();
        }
        return vo;
    }
}
```

Bu sınıf direk JDBC üzerinden veri tabanına bağlanarak, customer tablosunda, verilen id nin değerini sahip bir kayıt aramaktadır. Böyle bir sınıfı test edebilmek için önce veri tabanında belirli bir id ye sahip kayıt oluşturmamız gerekmektedir. Bu görevi bizim için DBUnit üstlenir.

CustomerDao.getCustomer() metodunu test etmek için CustomerDaoTest isminde bir test sınıfı oluşturuyoruz.

Kod 6.11 CustomerDaoTest sınıfı

```
package org.cevikjava.sample;

import org.cevikjava.common.DBUnitTestCase;
import org.cevikjava.samples.CustomerDao;
import org.cevikjava.samples.CustomerVo;

/**
 * CustomerDao sinifini test etmek
 * için kullanilir.
 *
 * @author Oezcan Acar
 *
 */
public class CustomerDaoTest    extends DBUnitTestCase
{

    private CustomerDao dao = null;
    private final long id = 1;

    public void testGetCustomer()
    {
        CustomerVo vo = null;
```

```
        try
        {
            vo = dao.getCustomer(id);
            assertNotNull(vo);
            assertEquals(vo.getName(), "Acar");
            assertEquals(vo.getFirstname(), "Oezcan");
        }
        catch (Exception e)
        {
            fail();
        }
    }


    public void setUp() throws Exception
    {
        super.setUp();
        dao = new CustomerDao();

    }

    public void tearDown() throws Exception
    {
        super.tearDown();
        dao = null;
    }
}
```

CustomerDaoTest sınıfı DBUnitTestCase sınıfını genişletmektedir. DBUniTestCase aşağıdaki yapıya sahiptir:

Kod 6.12 Soyut DBUnitTestCase sınıfı

```
package org.cevikjava.common;

import java.io.File;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import org.dbunit.DBTestCase;
import org.dbunit.DatabaseTestCase;
import org.dbunit.database.DataabaseConfig;
import org.dbunit.database.DatabaseConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.ext.hsqldb.HsqldbDataTypeFactory;
import org.dbunit.operation.DatabaseOperation;
```

```
public abstract class DBUnitTestCase
    extends DBTestCase
{

    private static final String DRIVERCLASS =
            "org.hsqldb.jdbcDriver";

    private static final String CONNECTIONURL =
            "jdbc:hsqldb:hsql://localhost:9006/cevikjava";

    private static final String USERNAME =
            "sa";

    private static final String PASSWORD =
            "";


    protected void setUp() throws Exception
    {
        try
        {
            IDatabaseConnection connection =
                        getConnection();
            IDataSet dataSet = getDataSet();
            try
            {
                DatabaseOperation.CLEAN_INSERT.
                        execute(connection, dataSet);
            }
            finally
            {
                connection.close();
            }

        }
        catch (Exception e)
        {
            throw e;
        }

    }

    protected final IDataSet getDataSet()
        throws Exception
    {
        final URL url = DatabaseTestCase.class
                .getResource("/dbunit-dataset.xml");
```

```
        final File file = new File(url.getPath());
        return new FlatXmlDataSet(file);
    }


    protected void tearDown() throws Exception
    {
        super.tearDown();
    }


    protected IDatabaseConnection getConnection()
        throws Exception
    {

        Class driverClass = Class.forName(DRIVERCLASS);

        Connection jdbcConnection =
            DriverManager.getConnection(
                    CONNECTIONURL,USERNAME, PASSWORD);

        IDatabaseConnection con =
                new DatabaseConnection(jdbcConnection);

        DatabaseConfig config = con.getConfig();
        config.setProperty(
                DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
                new HsqldbDataTypeFactory());
        return con;
    }
}
```

DBUnitTestCase DBUnit ile entegrasyonu sağlayan bir ara sınıftır. Bu sınıfın setUp() metodunda veri tabanı (HSQL) ile bağlantı kurularak, dbunit-dataset.xml dosyasında tanımlanmış olan kayıtlar veri tabanına aktarılır. DBUnitTestCase.setUp() CustomerDaoTest.testGetCustomer() metodundan önce devreye girdiği için test başlamadan önce veri tabanında veriler yerini alır.

CustomerDaoTest.testGetCustomer() test metodunda CustomerDao sınıfının getCustomer() metodunu test ediyoruz. Bu sınıf eğer geçerli bir id verildiği taktirde veri tabanındaki müşteri bilgilerini edinerek, CustomerVo isminde bir nesne içinde geriye verecektir. CustomerDaoTest.testGetCustomer() metodunda assert komutlarıyla CustomerDao.getCustomer() metodundan geriye verilen CustomerVo nesnesi değerlendirilir.

Görüldüğü gibi entegrasyon testleri için gerekli verileri otomatik olarak her test

öncesi veri tabanına eklememiz mümkündür.

Ant DBUnit Entegrasyonu

Şimdi bir adım daha ileri gidelim ve Ant ile DBUnit in nasıl entegre edilebileceklerine bir göz atalım. Burada amacımız tanımlanmış bir Ant hedefi (target) üzerinden veri tabanı tablolarını oluşturmak (create) ve verileri yüklemektir.

Bu amaçla build.xml içinde dbunit-hsql ismini taşıyan bir hedef oluşturuyoruz.

Kod 6.13 Ant DBUnit entegrasyonu

```
<taskdef
    name="dbunit"
    classname="org.dbunit.ant.DbUnitTask"
    classpath="${base.web.lib}/dbunit-2.2.jar"/>

<target name="dbunit-hsql" depends="hsql-execddl">
    <dbunit
        datatypefactory
        ="org.dbunit.ext.hsqldb.HsqldbDataTypeFactory"
        driver="org.hsqldb.jdbcDriver"
        url="jdbc:hsqldb:hsql://localhost:9006/cevikjava"
        userid="sa"
        password="">

        <classpath>
            <pathelement
            location="${base.web.lib}/hsqldb.jar" />
        </classpath>
        <operation
            type="INSERT"
            src="${base.dir}/properties/dbunit-dataset.xml"/>
    </dbunit>
</target>


<target name="hsql-execddl">

    <antcall target="hsql-stop" />
    <antcall target="hsql-start" />
    <sleep seconds="5" />

    <sql
        classpath="${hjar}"
```

```
        driver="org.hsqldb.jdbcDriver"
        url="jdbc:hsqldb:hsql://localhost:${hport}/${halias}"
        userid="sa"
        password=""
        print="yes">

        DROP TABLE customer IF EXISTS;

        CREATE TABLE customer
        (
           id IDENTITY NOT NULL,
           email VARCHAR(100) NOT NULL,
           password VARCHAR(100) NOT NULL,
           birthday DATE NOT NULL,
           firstname VARCHAR(255) NOT NULL,
           name VARCHAR(255) NOT NULL,
           sex int,
           approved boolean,
           banned boolean,
           activated boolean,
           created DATE NOT NULL,
           website int,
           activationcode varchar(10),
           language varchar(2)
        );
      </sql>
</target>
```

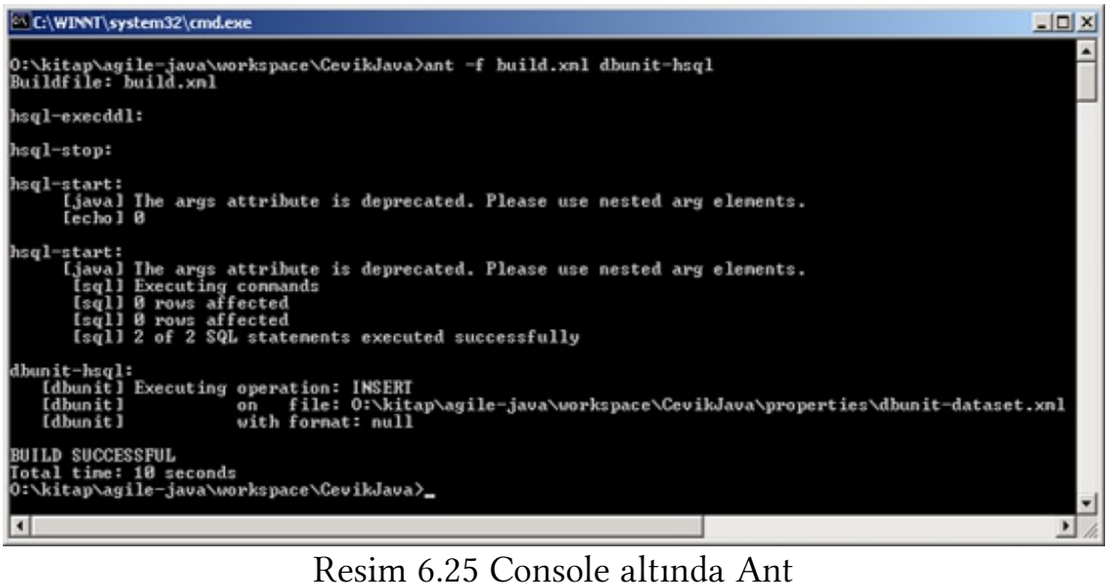
<dbunit> komutunun (Task) Ant tarafından anlaşılabilmesi için bu komutun <taskdef> ile tanımlanması gerekmektedir. build.xml içinde dbunit-hsql ve hsql-execddl isimlerinde iki yeni hedef tanımlıyoruz. hsql-execddl ile önce HSQL sunucusunu çalışır duruma getirilir, daha sonra <sql> komutuyla customer tablosu oluşturulur. dbunit-hsql hedefi dbunit-dataset.xml içinde tanımlanmış verileri HSQL veri tabanının customer tablosuna yükler.

Ant ile DBUnit in entegre edilmesi, programcı tarafından program ve veri tabanı arasındaki entegrasyon testlerinin çabukça yapılmasını kolaylaştırır. Programcı istediği zaman tek bir hedef (target) üzerinden veri tabanını silerek, yeniden oluşturabilir.

Eclipse altında bazen build.xml ile sorunlar yaşanabiliyor. Benim kullandığım sürümde örneğin dbunit-hsql hedefi çalışmamakta kararlıydı. Bu gibi durumlarda build.xml console altında çalıştırılabilir.

```
O:\kitap\agile-java\workspace\CevikJava>ant -f build.xml dbunit-hsql
```

```
Buildfile: build.xml

hsql-execddl:

hsql-stop:

hsql-start:
     [java] The args attribute is deprecated. Please use nested arg
                    elements.
     [echo] 0

hsql-start:
     [java] The args attribute is deprecated. Please use nested arg
                    elements.
      [sql] Executing commands
      [sql] 0 rows affected
      [sql] 0 rows affected
      [sql] 2 of 2 SQL statements executed successfully

dbunit-hsql:
   [dbunit] Executing operation: INSERT
   [dbunit]           on   file: O:\kitap\agile-java\workspace\
                          CevikJava\properties\dbunit-dataset.xml
   [dbunit]           with format: null

BUILD SUCCESSFUL
Total time: 10 seconds
```



Resim 6.25 Console altında Ant

7. Bölüm

Çevik Tasarım

Giriş

Tasarım, bir program için tasarım şablonları ve prensipleri kullanılarak oluşturulan yapıdır. Bu yapı zaman içinde implementasyon esnasında oluşur. Çevik süreçlerde implementasyon öncesi detaylı olarak (big design upfront) tasarım oluşturulmaz. Her iterasyonda program müşteri istekleri doğrultusunda değişikliğe uğrar. Çevik programcı takımlar bu değişiklikler doğrultusunda program yapısını da bir adım ileriye götürerek, bakımı ve geliştirilmesi kolay kod oluştururlar.

İyi bir tasarım oluşturulan program için hayat sigortasıdır. Zaman içinde program üzerinde yapılan değişiklikler program yapısının bozulmasına sebep verebilir. Bunun sebebi zamanla değişen müşteri gereksinimleridir. Değişen müşteri gereksinimlerini adapte edebilmek için kod üzerinde değişiklik yapılır. Eğer bu çalışmalar mevcut yapıyı gereksinimler paralelinde ileriye taşımazsa, tasarım bozulmaya başlar. Bunun sonucu olarak programın değiştirilmesi ve geliştirilmesi zorlaşır. Belli bir aşamadan sonra bozulan tasarım, programın geliştirilmesini bloke eder ve yeni bir tasarım sürecini gerekli kılar.

Bu bölümde iyi bir tasarım için kullanabileceğimiz tasarım prensiplerini yakından inceleyeceğiz.

Test Edilebilir Tasarım

XP projeleri test güdümlü (test driven development =TDD) ilerler. Programcı testlerin gerektirdiği sınıfları oluştururken tasarım kararları alır ve bu bunları uygular. Bu tasarım kararları kodun gelecekte ne oranda yeniliklere açık olduğunu belirler. Seçilen tasarımın test edilebilir yapıda olması da büyük önem taşımaktadır. Aksi taktirde testlerin koddan önce oluşturulması bir sorun haline gelir. TDD tarzı implementasyonda tasarım sorunlarıyla karşılaşmamak ve test edilebilir bir tasarım oluşturmak için takip edilmesi gereken bazı tasarım prensipleri şöyledir:

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

Kalıtım Yerine Kompozisyon Kullanılmalıdır

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

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

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

Statik Metot ve Tekil Yapılar Kullanılmamalıdır

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

Kod 7.1 Static metod ve test edilebilirlik sorunu

```
package shop;

public class PdfCreator
```

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

package shop;

import junit.framework.TestCase;

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

Kod 7.1 de yer alan PdfCreator.create() metodunu PdfCreatorTest.testPdfCreator() metodunda test edebilmemiz için PdCreator sınıf ismini test içinde kullanmamız gerekiyor, çünkü create() statik bir metottur. Bu test metodu ile PdfCreator sınıfı arasında bir bağımlılık oluşturuyor ve bizim create() metodunu örneğin bir mock nesne ile simüle etmemizi engelliyor. Burada PdfCreator sınıfını kalıtım ile genişletip, create() metodunu yeniden implemente etsek bile, PdfCreator ve test metodu arasındaki bağımlılıktan dolayı yeni sınıfı kullanmamız mümkün değildir.

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

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

Bağımlılıkların İzole Edilmesi Gerekir

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

Kod 7.2 Metot bağımlı olduğu nesneyi bir singleton üzerinden alıyor

```
package shop;

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

Kod 7.2 de yer alan create() metodu bir Pdf dosya oluşturabilmek için PdfCreatorService singleton sınıfını kullanmaktadır. create() metodunu bu hali ile test etmek çok güçtür, çünkü PdfCreatorService bir singleton sınıf olduğu için bünyesinde olup bitenleri kontrol etme şansımız yoktur. Örneğin bu singleton sınıf private olan konstrüktöründe bir veri tabanına bağlanarak gerekli konfigürasyonları ediniyor olabilir. Bu durumda create() metodunu test etmek için oluşturduğumuz testlerde veri tabanı bağımlılığı oluşacaktır. Bu istenmeyen bir durumdur.

create() metodundaki bağımlılığı kod 7.3 deki gibi izole ederek, değiştirilebilir hale getirebiliriz.

Kod 7.3 Bağımlılık başka bir metot içinde isole edilir

```
package shop;

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


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

getService() isminde yeni bir metot oluşturarak gerekli servis sınıfının edinilme

işlemini izole etmiş oluyoruz. Test esnasında bu izole edilen bölümü değiştirerek, create() metodunu test edilebilir hale getirebiliriz.

Kod 7.4 İsole ettiğimiz metodu başka implementasyon ile değiştiriyoruz

```
package shop;

import junit.framework.TestCase;

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

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

PdfCreator sınıfından yeni bir nesne oluştururken getService() metodunu yeniden yapılandırabiliriz. Bu bize istediğimiz service implementasyonunu kullanma fırsatı verir. Bunun bir örneği kod 7.4 de yer almaktadır. DummyPdfCreatorService test için oluşturduğumuz bir mock sınıftır. Bu sınıfın kullanımı create() metodunun test edilmesini kolaylaştırmaktadır.

Bağımlılıkların Enjekte Edilmesi Testleri Kolaylaştırır

Kod 7.3 de PdfCreatorService sınıfına direk bağımlı olan PdfCreator sınıfı yer almaktadır. Bu bağı test esnasında kod 7.4 de yer aldığı gibi yeni bir implementasyon ile çözebiliriz. Bu işlemi daha kolay hale getirmek için başka bir yöntem daha mevcuttur.

İki sınıf arasındaki ilişkinin daha esnek bir hale gelmesini sağlamak ve testleri kolaylaştırmak için bağımlılıkların enjekte edilmesi (dependency injection) metodunu kullanabiliriz. Bunun bir örneğini kod 7.5 de görmekteyiz.

Kod 7.5 Dependency injection

```
package shop;

public class PdfCreator
{

    private PdfCreatorService service;

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

    public PdfCreatorService getService()
      {
        return service;
    }

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

PdfCreatorService sınıfından olan bir değişkeni sınıf değişkeni (service) olarak tanımlıyoruz. setService() metodu ile bu değişkenin değerini değiştirebiliriz. Bu bize test esnasında istediğimiz bir implementasyonu kullanma imkanı tanıyacaktır. Bunun nasıl yapıldığı kod 7.6 da yer almaktadır.

Kod 7.6 Dependency injection

```
package shop;

import junit.framework.TestCase;

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

```
}
```

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

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

Tasarım Prensipleri

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

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

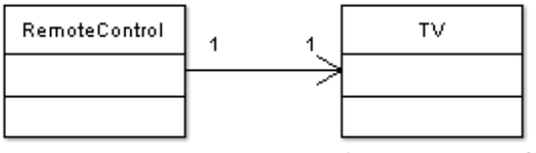
- Loose Coupling (LC)
- Open Closed Principle (OCP)
- Single Responsibility Principle (SRP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)
- Reuse-Release Equivalence Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure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

Loose Coupling (LC) - Esnek Bağ

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

Buradan "Eğer bağımlılık varsa, sorun var, bu yüzden bağımlılıkların ortadan kaldırılması gerekmektedir" sonucunu çıkartabiliriz. Nesneye yönelik tarzda tasarlanmış bir program içinde bağımlılıkları ortadan kaldırmak imkansızdır, çünkü nesnelerin olduğu yerde interaksiyon ve bağımlılık olmak zorundadır. Bağımlılıkları ortadan kaldıramıyorsak, o zaman onları kontrol altına almamız işimizi kolaylaştıracaktır. Eğer bana soracak olursanız, yazılım disiplininin özü de burada saklıdır: Yazılımcı olarak kullandığımız tüm metotların temelinde bağımlılıkların kontrolü ve yönetilmesi yatmaktadır. İyi bir tasarım oluşturmak için sarf ettiğimiz efor bağımlılıklara hükmetmek isteyişimizden kaynaklanmaktadır, yani iyi bir tasarım kontrol edilebilir bağımlılıkları beraberinde getirdiği için iyidir, kendimizi programcı olarak iyi hissetmemizi sağladığı için değil! Biliyoruz ki kötü bir tasarım nesneler arası yüksek derecede bağımlılığa sebep vereceği için programcı olarak hayatımızı zorlaştıracaktır. Bu yüzden sahip olduğumuz tüm teknik yeteneklerimizle bağımlılığa karşı bir savaş veririz. Onu yenmemiz mümkün olmasa da, bize zarar vermeyecek şekilde kontrol altına almamız mümkündür. Bunun yolu da çevik tasarımdan geçmektedir.

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



Resim 7.1 RemoteControl ve TV sınıfları

RemoteControl (tv uzaktan kumanda aleti) ve TV (televizyon) sınıflarının arasında tek yönlü (uni directional) bir bağ oluşmuştur, çünkü bir RemoteControl nesnesi görevini yerine getirebilmek için bir TV nesnesine ihtiyaç duymaktadır. Nesneler arası bağımlılıkların yönü vardır. Resim 7.1 de bu bağımlılığın yönünün RemoteControl sınıfından TV sınıfına doğru olduğunu görmekteyiz. Bu tek yönlü bir bağdır ve RemoteControl sınıfı bünyesinde TV tipinde bir sınıf değişkeni barındırarak, bağı oluşturur. Sınıflar arası bağlar karşılıklı da (bidirectional) olabilir. İki taraflı bağlarda sınıflar bir sınıf değişkeni aracılığıyla karşılıklı olarak birbirlerine işaret ederler. RemoteControl nesnesi bir TV nesnesi olmadığı sürece işe yaramaz. Bu yüzden bu iki sınıf arasında direk bağlantı oluşturulmuştur. Böyle bir bağımlılık aşağıdaki sorunların oluşmasına sebep vermektedir:

- RemoteControl nesnesi bir TV nesnesi olmadığı sürece kendi görevini yerine getiremez, bu yüzden tek başına bir işe yaramaz. Mutlaka ve mutlaka var olabilmesi için bir TV nesnesine ihtiyaç duymaktadır. Bu durumda RemoteControl sınıfını başka bir alanda kullanmak istediğimizde TV sınıfını da yanına koymak zorundayız. Böyle bir bağımlılık RemoteControl sınıfının tek başına başka bir alanda tekrar kullanılmasını engellemektedir. Buradan söyle bir sonucu çıkartıyoruz: Program parçalarının (sınıflar) tekrar kullanılabilir olabilmeleri için diğer sınıflara olan bağımlılıklarının düşük seviyede olması gerekmektedir. Esnek bağımlılık olmadan bir kere yazılan kodun tekrar kullanımı çok güçtür.
- TV sınıfı bünyesinde meydana gelen yapısal değişiklikler RemoteControl sınıfını doğrudan etkileyecektir. Böyle bir bağımlılık RemoteControl sınıfının yapısal değişikliğe uğrama rizikosunu artırır.
- RemoteControl nesnesi sadece bir TV nesnesini (yani bir televizyonu) kontrol edebilir. Oysaki bir evde uzaktan kumanda edilebilecek başka aletler de olabilir. Böyle bir bağımlılık RemoteControl nesnesini sadece bir TV nesnesiyle beraber çalışmaya mahkum eder. Uzaktan kumanda edilebilen aletler için başka RemoteControl sınıflarının oluşturulması gerekmektedir, örneğin bir CD çalıcıyı kontrol etmek için CDPlayerRemoteControl isminde bir sınıfı oluşturmak zorundayız.

RemoteControl sınıfı kod 7.7 de yer almaktadır.

Kod 7.7 RemoteControl.java

```
package org.cevikjava.design.loosecoupling;
```

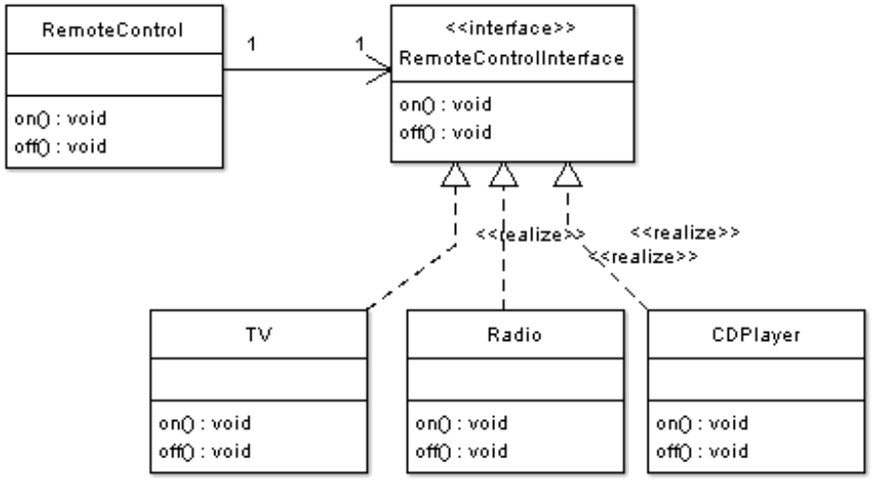
```
/**
 * Bir televizyonu uzaktan kuman etme
 * aletini simule eden sınıf.
 *
 * @author Oezcan Acar
 *
 */
public class RemoteControl
{
    /**
     * Kontrol edilen televizyon
     */
    private TV tv = new TV();

    /**
     * Televizyonu acmak
     * için kullanilan metot.
     *
     */
    public void tvOn()
    {
        tv.on();
    }


    /**
     * Televizyonu kapatmak
     * için kullanilan metot.
     */
    public void tvOff()
    {
        tv.off();
    }
}
```

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

Kod 7.8 TV.java

```
package org.cevikjava.design.loosecoupling;

/**
 * Bir televizyonu simule eden sınıf.
 *
 * @author Oezcan Acar
 *
 */
```

```
public class TV
{
    /**
     * Televizyonu acmak için
     * kullanilan metot.
     *
     */
    public void on()
    {
        System.out.println("TV acildi.");
    }

    /**
     * Televizyonu kapatmak için
     * kullanilan metot.
     *
     */
    public void off()
    {
        System.out.println("TV kapandi");
    }
}
```



Resim 7.2 Esnek bağ için gerekli sınıf yapısı

Nesneler arası kuvvetli bağların oluşmasının programın bakımı, geliştirilmesi ve kodun tekrar kullanımını negatif yönde etkilediğini gördük. Bu sorunu ortadan kaldırabilmek için sınıfların ilişkileri üzerinde yapısal değişikliğe gitmemiz gerekiyor. Esnek bağ oluşturabilmek için soyut (abstract) ya da interface sınıflardan faydalanabiliriz. Resim 7.2 de görüldüğü gibi RemoteControlInterface ismini taşıyan bir interface sınıf ile RemoteControl ve bu sınıfın kontrol etmek istediği aletler arasında esnek bir bağ oluşturuyoruz. Böyle bir yapının esnekliği nereden gelmektedir, bunu yakından inceleyelim.

Kod 7.9 RemoteControlInterface.java

```
package org.cevikjava.design.loosecoupling.design;

/**
 * RemoteControlInterface sınıfı
 *
 * @author Oezcan Acar
 *
 */
public interface RemoteControlInterface
{
    /**
     * Bu sınıfı implement eden
     * bir aleti acmak için
     * kullanilan metot.
     *
     */
    void on();


    /**
     * Bu sınıfı implement eden
     * bir aleti kapatmak için
     * kullanilan metot.
     *
     */
    void off();
}
```

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

Resim 7.2 de görüldüğü gibi RemoteControl sınıfı direk TV sınıfı kullanmak yerine, RemoteControlInterface ismini taşıyan interface sınıfı ile beraber çalışmaktadır. Böyle bir yapılanma ile RemoteControl ve bu sınıfın kullanmak istediği somut sınıflar (örneğin TV) arasına bir set çekmiş olduk. RemoteControl sınıfı bu setin arkasında yer alan sınıfları, bunlar RemoteControlInterface sınıfını implemente eden sınıflardır, tanımamaktadır ve tanımak zorunda değildir. RemoteControl sınıfının tanıması gereken tek sınıf RemoteControlInterface sınıfı ve bu interface sınıfının dış dünyaya sunduğu

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

Kod 7.10 RemoteControl.java

```
package org.cevikjava.design.loosecoupling.design;

/**
 * RemotControlInterface sınıfını
 * implemente eden sınıfları
 * kontrol edebilen sınıf.
 *
 * @author Oezcan Acar
 *
 */
public class RemoteControl
{
    /**
     * Delegasyon islemi için RemoteControlInterface
     * tipinde bir sınıf degiskeni tanimliyoruz.
     * Tüm islemler bu nesnenin metodlarina
     * delege edilir.
     */
    private RemoteControlInterface remote;

    /**
     * Sinif konstuktörü. Bir nesne oluşturma islemi
     * esnasında kullanilacak RemoteControlInterface
     * implementasyonu parametre olarak verilir.
     *
     * @param _remote RemoteControlInterface
     */
    public RemoteControl(RemoteControlInterface _remote)
    {
        this.remote = _remote;
    }

    /**
     * Aleti acmak
     * için kullanilan metot.
     *
     */
    public void on()
    {
        remote.on();
```

```
    }


    /**
     * Aleti kapatmak
     * için kullanilan metot.
     */
    public void off()
    {
        remote.off();
    }

}
```

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

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

Kod 7.11 TV.java

```
package org.cevikjava.design.loosecoupling.design;

/**
 * Bir televizyonu simule eden sınıf.
 * RemoteControlInterface sınıfını
 * implemente ederek bir RemoteControlInterface
 * haline gelir.
 *
 * @author Oezcan Acar
 *
 */
public class TV implements RemoteControlInterface
{
```

```
    /**
     * Televizyonu acmak için
     * kullanilan metot.
     *
     */
    public void on()
    {
        System.out.println("TV acildi.");
    }

    /**
     * Televizyonu kapatmak için
     * kullanilan metot.
     *
     */
    public void off()
    {
        System.out.println("TV kapandi");
    }
}
```

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

Kod 7.12 Test.java

```
package org.cevikjava.design.loosecoupling.design;

/**
 * Test sınıfı
 *
 * @author Oezcan Acar
 *
 */
public class Test
{

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

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

```
TV acildi.
TV kapandi
```

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

Oluşturduğumuz yeni tasarımın bize sağladığı avantajlar şöyledir:

- Bir interface sınıf (RemoteControlInterface) kullanarak RemoteControl ve kontrol etmek istediği aletler (TV, CDPlayer) arasında bir bariyer oluşturduk. RemoteControl sınıfı kontrol etmek istediği aleti tanımak zorunda olmadığı için bu sınıf ve diğerleri arasındaki sıkı bağı çözmüş ve interface sınıf kullanarak daha esnek bir hale getirmiş oluyoruz.
- Bu tarz bir tasarım ile programı gelecekte oluşacak değişiklikleri taşıyabilecek hale getirdik. RemoteControl sınıfı RemoteControlInterface sınıfını implemente eden her sınıfı kontrol edebilir. RemoteControlInterface sınıfını implemente ederek sisteme uzaktan kumanda edilebilen yeni aletler ekleyebiliriz.
- RemoteControl sınıfı, RemoteControlInterface sınıfının implemente edildiği başka bir ekosistemde tekrar kullanılabilir hale geldi. Esnek bağımlılık oluşturmak kodun tekrar kullanımını kolaylaştırmaktadır.

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

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

sürecine olan bakış açınızı tamamen değiştirecek niteliktedir.

Open Closed Principle (OCP) - Açık Kapalı Prensibi

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

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

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

Şu şekilde tercüme edilebilir:

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

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

Çevik süreç büyük çapta bir tasarım hazırlığı ile start almaz. Daha ziyade mümkün olan en basit şekilde yazılıma başlanır. Her iterasyon başlangıcında implemente edilmesi gereken kullanıcı hikayeleri seçildikten sonra mevcut yapının yeni gereksinimlere cevap verip, veremeyeceği incelenir. Büyük bir ihtimalle mevcut yapı yeni kullanıcı hikayelerinin implementasyonu için yeterli

olmayacaktır. Bu durumda programcı ekip refactoring yöntemleriyle programın yapısını yeni gereksinimleri kabul edecek şekilde modifike eder. Bu esnada tasarım yapısal değişikliğe uğrar. Bu gerekli bir işlemdir ve yapılmak zorundadır, aksi taktirde bir sonraki iterasyon beraberinde getirdiği değişikliklerle programcı ekibini yazılım esnasında zorlayacaktır.

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

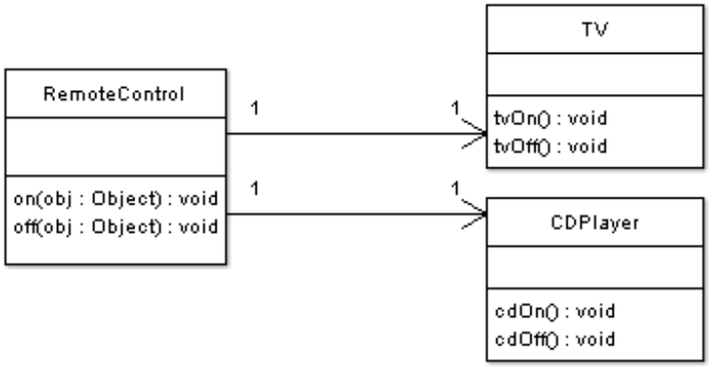
Tekrar bölüm başında sorduğum soruya dönelim ve sorunun cevabını bulmaya çalışalım. Şu şekilde bir soru sormuştum: "Değişiklik kaçınılmaz olduğuna göre, bir programı gelecekte yapılması gereken tüm değişiklikleri göz önünde bulundurarak, şimdiden buna hazır bir şekilde geliştirmek mümkün müdür?" Bu mümkün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

Programlar geliştirilmeye açık ama değiştirilmeye kapalı olmalıdır.

Programı geliştirmek programa yeni bir davranış biçimi eklemek anlamına gelmektedir. OCP ye göre programlar geliştirmeye açık olmalıdır, yani programı oluşturan modüller yeni davranış biçimlerini sergileyecek şekilde genişletilebilmelidirler. Bir modüle yeni bir davranış biçimi kazandırılarak düşünülen değişiklik sağlanır. Bu yeni kod yazılarak gerçekleştirilir (bu yüzden bu işleme değiştirme değil, genişletme denir), mevcut kodu değiştirerek değil! Eğer kendinizi bir müşteri gereksinimini mevcut kod üzerinde değişiklik yaparken bulursanız, biliniz ki OCP prensibine ters düşüyorsunuz. Kod üzerinde yapılan değişiklik, bir sonraki gereksinimlerinde ayni şekilde implemente edilmesini zorunlu kılacaktır. Bu durum kodun zaman içinde içinden çıkılmaz ve çok karmaşık bir yapıya dönüşmesini çabuklaştırır.

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

Resim 7.1 de görüldüğü gibi RemoteControl sınıfı TV sınıfını kullanarak işlevini yerine getirmektedir. Eğer RemoteControl sınıfını TV haricinde başka bir aleti kontrol etmek için kullanmak istersek, örneğin CDPlayer Resim 7.3 deki gibi değişiklik yapmamız gerekebilir. Bu noktada esnek bağ prensibini unutarak, resim 7.3 de yer alan çözümün bizim için yeterli olduğunu düşünelim.



Resim 7.3 RemoteControl sınıfı TV ve CDPlayer sınıflarından olan nesneleri kontrol etmektedir.

Kod 7.13 RemoteControl.java

```
package org.cevikjava.design.ocp;

/**
 * TV ve CDPlayer sınıflarından
 * nesneleri kontrol eder.
 *
 * @author Oezcan Acar
 *
 */
public class RemoteControl
{

    /**
     * Aleti acmak
     * için kullanilan metot.
     *
     */
    public void on(Object obj)
    {
        if(obj instanceof TV)
        {
            ((TV)obj).tvOn();
        }
```

```
        else if(obj instanceof CDPlayer)
        {
            ((CDPlayer)obj).cdOn();
        }
    }


    /**
     * Aleti kapatmak
     * için kullanilan metot.
     */
    public void off(Object obj)
    {
        if(obj instanceof TV)
        {
            ((TV)obj).tvOff();
        }
        else if(obj instanceof CDPlayer)
        {
            ((CDPlayer)obj).cdOff();
        }

    }
}
```

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

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

Resim 7.2 de yer alan çözüm OCP ye uygun yapıdadır, çünkü kod üzerinde değişiklik yapmadan programa yeni davranışlar eklemek mümkündür. OCP prensibi, esnek bağ prensibi kullanılarak uygulanabilir. OCP ye uygun RemoteControl sınıfının yapısı şu şekilde olmalıdır:

```
Kod 7.14   RemoteControl.java

package org.cevikjava.design.loosecoupling.design;
```

```
/**
 * RemotControlInterface sınıfını
 * implemente eden sınıfları
 * kontrol edebilen sınıf.
 *
 * @author Oezcan Acar
 *
 */
public class RemoteControl
{
    /**
     * Delegasyon islemi için RemoteControlInterface
     * tipinde bir sınıf degiskeni tanimliyoruz.
     * Tüm islemler bu nesnenin metodlarina
     * delege edilir.
     */
    private RemoteControlInterface remote;

    /**
     * Sinif konstuktörü. Bir nesne oluşturma islemi
     * esnasında kullanilacak RemoteControlInterface
     * implementasyonu parametre olarak verilir.
     *
     * @param _remote RemoteControlInterface
     */
    public RemoteControl(RemoteControlInterface _remote)
    {
        this.remote = _remote;
    }

    /**
     * Aleti acmak
     * için kullanilan metot.
     *
     */
    public void on()
    {
        remote.on();
    }


    /**
     * Aleti kapatmak
     * için kullanilan metot.
     */
    public void off()
    {
        remote.off();
```

```
        }

}
```

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

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

Stratejik Kapama (Strategic Closure)

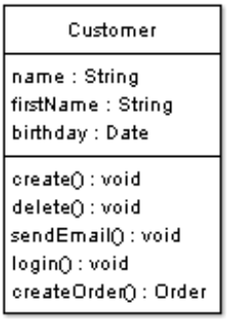
Ne yazık ki bir yazılım sistemini OCP prensibini uygulayarak %100 değişikliklere karşı korumamız imkansızdır. OCP ye uygun olan bir metot müşterinin yeni istekleri doğrultusunda OCP ye uygun olmayan bir hale gelebilir. Programcı metodu ilk implemente ettiği zaman gelecekte olabilecek değişiklikler hakkında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

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

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

Single Responsibility Principle (SRP) – Tek Sorumluk Prensibi

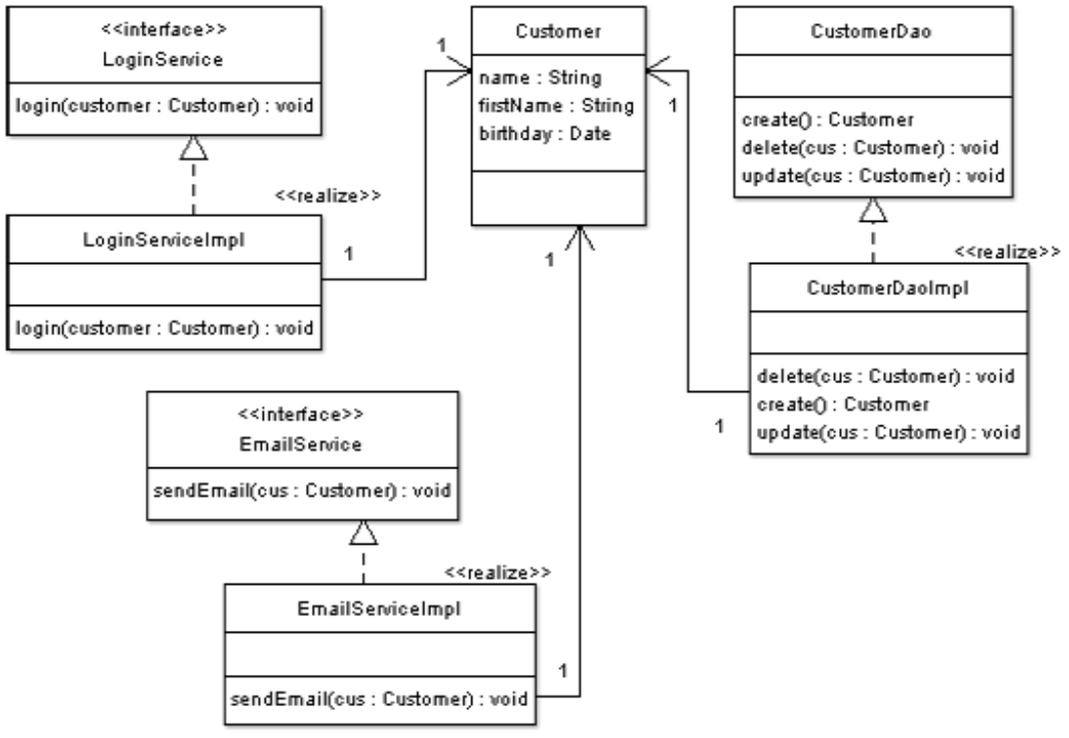
Resim 7.4 deki gibi bir sınıfa daha önce bir yerlerde rastlamışsınızdır. Bu sınıf kendisini veri tabanına ekleme, silme, müşteriye email gönderme, login yapma (shop sistemi olabilir) ve sipariş oluşturma işlemlerini yapabilmektedir.



Resim 7.4 Her işi kendi başına yapmaya çalışan bir Customer sınıfı. Bu sınıf gereğinden fazlasını biliyor.

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

Customer sınıfını test edilebilir ve bakımı kolay bir hale getirmek için sahip olduğu sorumlulukların başka sınıflara yüklenmesi gerekmektedir. Bunun bir örneğini resim 7.5 de görmekteyiz.



Resim 7.5 Sorumluluk paylaşımı

Resim 7.5 de yer alan Customer sınıfı sahip olduğu sorumlulukların başka sınıflara yüklenmesiyle hafiflemiş ve değişikliklere karşı daha dayanıklı bir hale gelmiştir. Bunun yanı sıra bu sınıfın test edilebilirliği yükselmiştir. Bu ve diğer sınıfların değişmek için artık sadece bir sebebi vardır. Bunun sebebi sadece bir sorumluluk sahibi olmalarında yatmaktadır.

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

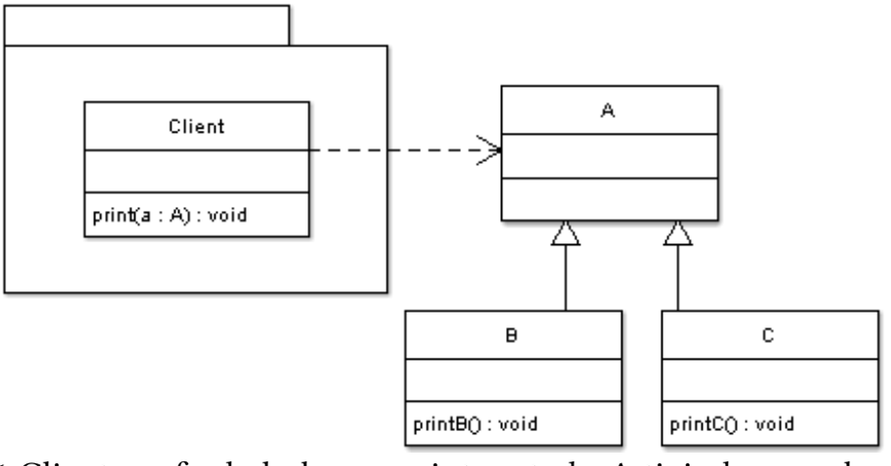
Liskov Substitution Principle (LSP) – Liskov Yerine Geçme Prensibi

Barbara Liskov tarafından geliştirilen bu prensip kısaca şöyle açıklanabilir:

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

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

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



Resim 7.6 Client sınıfında bulunan print metodu A tipinde nesneler üzerinde işlem yapmaktadır

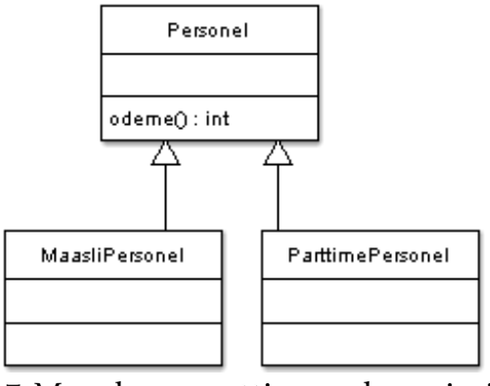
Resim 7.6 da yer alan Client sınıfındaki print() metodunun nasıl LSP prensibine ters düştüğünü yakından inceleyelim. Bu metot A sınıfından olan nesneler üzerinde işlem yapmaktadır. A sınıfı B ve C sınıfları tarafından genişletilmiştir, yani A sınıfından olan bir parametre nesnesi aynı zamanda B ve C sınıfında da olabilir.

Kod 7.15 Client.java

```
package shop;

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

```
            ((C)a).printC();
        }
    }

    public static void main(String[] args)
    {
        Client client = new Client();
        B b = new B();
        C c = new C();

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

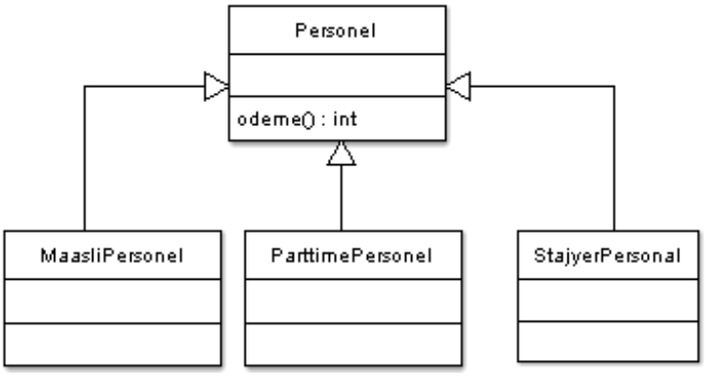
Kod 7.15 de yer alan print() metodu LSP ile uyumlu değildir. Bu metodun A sınıfına bağımlılığı vardır ve bu sınıfın nesneleri üzerinde işlem yapacak şekilde implemente edilmiş olması gerekir. Lakin print() metodu instanceof komutuyla parametre olarak verilen nesnenin hangi tipte olduğunu tespit etmeye çalışmakta ve tipe bağımlı olarak nesne üzerinde işlemi gerçekleştirmektedir. Bu durum hem OCP ye hemde LSP ye ters düşmektedir. OCP uyumlu değildir, çünkü print() metodu A nin her yeni alt sınıfı için değişikliğe uğramak zorundadır. LSP ye uyumlu değildir, çünkü B yada C sınıfından olan nesneler A sınıfından olan bir nesnenin yerini alamadığı için print() metodu instanceof ile nesnenin tipini tespit etmek zorunda bırakılmaktadır. Buradan genel bir sonuç çıkartabiliriz: LSP ye ters düşen bir implementasyon aynı zamanda OCP ye de ters düşer.



Resim 7.7 Maaşlı ve parttime çalışan işçi modeli

LSP nin nasıl uygulanabileceğini diğer bir örnekte görelim. Resim 7.7 de bir firma çalışanları Personel sınıfını genişleten MaasliPersonel ve ParttimePersonel sınıfları ile temsil edilmektedirler. Personel sınıfı soyuttur (abstract). Çalışanların maaşlarının ödenebilmesi için Personel sınıfında

bulunan soyut odeme() metodunun alt sınıflarca implemente edilmesi gerekmektedir. Firmaya yeni stajyerler alındığı için modelin resim 7.8 deki gibi adapte edildiğini farz edelim.



Resim 7.8 Maaşlı, parttime ve stajyer işçi modeli

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

Kod 7.16 StajyerPersonel.java – odeme() metodu

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

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

Bu sorunu kod 7.17 de yer aldığı gibi bir Exception oluşturarak çözebiliriz. Eğer bir stajyer için odeme() metodu kullanılacak olursa, oluşan PersonelException, bir stajyer için maaş ödenemeyeceğini metot kullanıcısına bildirir. Exception nesneleri olağan dışı durumları ifade etmek için kullanılır.

Kod 7.17 StajyerPersonel.java – odeme() metodu

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

Bu implementasyon ilk örnekte olduğu gibi LSP ile uyumlu değildir. Neden uyumlu olmadığını inceleyelim. Kod 7.18 de çalışanlara ödenen toplam maaş

miktarı hesaplanmaktadır.

```
Kod 7.18   Toplam maaş miktarı hesaplanıyor

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

StajyerPersonal sınıfının sisteme eklenmesiyle kod 7.18 de yer alan kodun değiştirilmesi gerekmektedir. Ya try/catch bloğu kullanılarak oluşabilecek bir PersonelException ın işleme tabi tutulması gerekir ya da metot imzasında throws kullanılarak PersonelException ın bir üst katmana iletilmesi gerekir. StajyerPersonel ismini taşıyan yeni bir sınıf oluştuğu için mevcut Personel sınıfı kullanıcıları (client) yapısal değişikliğe uğramıştır.

```
Kod 7.19   try/catch...

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


Kod 7.20   instanceof

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

Try/catch blokları komplike yapılardır ve kodun okunulabilirliğini azaltırlar. Try/catch bloğundan kurtulmak için kod 7.20 de yer alan implementasyon düşünülebilir. Burada instanceof ile sırada hangi tip bir nesnenin olduğunu tespit edebilir ve StajyerPersonel tipi nesneleri işlem dışı bırakabiliriz. Daha öncede gördüğümüz gibi bu implementasyon LSP ile uyumlu değildir, çünkü üst sınıf ile çalışan bir metot instanceof ile alt sınıfları tanımak zorunda bırakılmaktadır. Bunun tek sebebi StajyerPersonel sınıfından olan bir nesnenin, Personel sınıfından bir nesne ile yer değiştiremez durumda olmasıdır. Personel sınıfını kullanan diğer sınıflar alt sınıf olan StajyerPersonel sınıfı sisteme eklendikten sonra bu değişiklikten etkilenmiştir. LSP ye göre bu olmaması gereken bir durumdur. Alt sınıfların nesneleri, üst sınıflardan olan nesnelerin kullanıldığı metotlar içinde üst sınıf nesneleriyle aynı davranışı göstermek zorundadırlar, aksi taktirde kullanıcı sınıflar bu durumdan etkilenirler.

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

Dependency Inversion Principle (DIP) – Bağımlılıkların Tersine Çevrilmesi Prensibi

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

Resim 7.1 de görülen yapı DIP prensibine ters düşmektedir, çünkü RemoteControl sınıfı somut bir sınıf olan TV sınıfına bağımlıdır. TV bünyesinde meydana gelen her değişiklik doğrudan RemoteControl sınıfını etkileyecektir. Ayrıca RemoteControl sınıfını TV sınıfı olmadan başka bir yerde kullanılması mümkün değildir.

Resim 7.2 de yer alan tasarım DIP ile uyumludur, çünkü RemoteControl sınıfı somut olan TV yerine soyut olan bir interface sınıfına bağımlıdır. Bu tasarım RemoteControl sınıfı ile TV sınıfı arasındaki bağımlılığı yok eder. Ayrıca

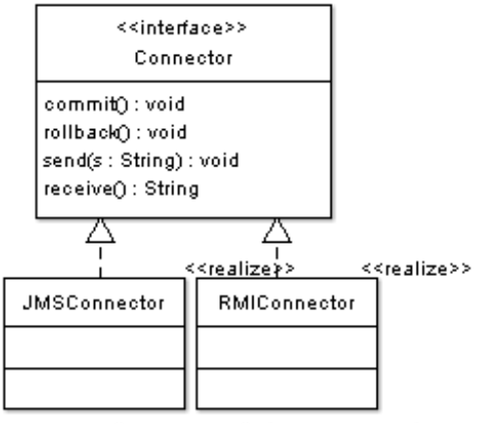
RemoteControl ve bağımlı olduğu RemoteControlInterface sınıfı beraber başka bir yerde tekrar kullanılabilir.

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

Interface Segregation Principle (ISP) – Arayüz Ayırma Prensibi

Birbiriyle ilişkili olmayan birçok metodu ihtiva eden büyük bir interface sınıf yerine, birbiriyle ilişkili (cohesive) metotların yer aldığı birden fazla interface sınıfı daha makbuldür.

ISP uygulanmadığı taktirde birden fazla sorumluluğu olan interface sınıflar oluşur. Zaman içinde yüklenen yeni sorumluluklarla bu interface sınıflar daha da büyür ve kontrol edilemez bir hale gelebilir. Bunun bir örneğini resim 7.9 da görmekteyiz.



Resim 7.9 Connector interface sınıfı bünyesinde gereğinden fazla metot barındırmaktadır

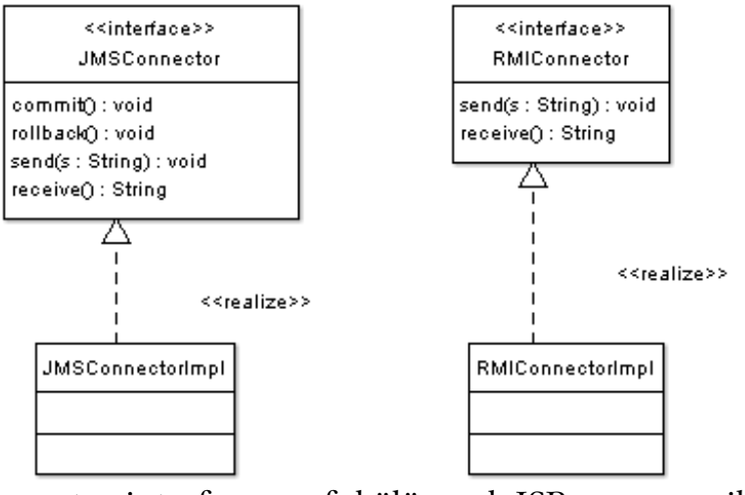
Resim 7.9 da yer alan Connector interface sınıfı bünyesinde JMS (Java Message Service) için gerekli iki metot bulunmaktadır: commit ve rollback. Bu metotların RMIConnector implementasyonunda implemente edilmeleri anlamsız olur. Büyük bir ihtimalle RMIConnector sınıfında bu metotların implementasyonu kod 7.21 de yer aldığı şekilde olacaktır.

```
Kod 7.21   RMIConnector.java

package shop;

public class RMIConnector implements Connector
```

```
{

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

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

Kod 7.21 de yer alan yapı LSP ile uyumlu değildir. Connector sınıfını kullananlar RuntimeException oluşabileceğini göz önünde bulundurmak zorunda bırakılmaktadırlar.

ISP uygulandığı taktirde resim 7.9 da yer alan Connector interface sınıfını yok ederek, yerine sorumluluk alanları belli iki yeni interface sınıf oluşturabiliriz. Resim 7.10 da bu çözüm yer almaktadır.



Resim 7.10 Connector interface sınıfı bölünerek ISP ye uygun iki yeni interface sınıf oluşturuldu

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

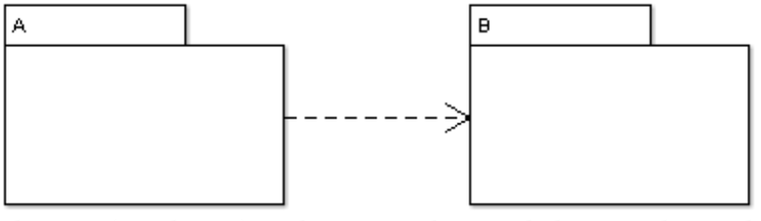
Paket Tasarım Prensipleri (Principles of Package Design)

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

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

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

Program modülleri paketler (packages) kullanılarak organize edilir. Paketler arasında sınıfların birbirlerini kullanmalarıyla bağımlılıklar oluşur. Bunun bir örneği resim 7.11 de yer almaktadır. Eğer paket B bünyesindeki bir sınıf paket A bünyesinde bulunan bir sınıf tarafından kullanılıyorsa, bu paket A nin paket B ye bağımlılığı olduğu anlamına gelir. Bu tür bağımlılıkların oluşması doğaldır. Amaç bu bağımlılıkları ortadan kaldırmak değil, kontrol edilebilir hale getirmek olmalıdır. Bu amaçla paket bazında uygulanabilecek tasarım prensipleri oluşturulmuştur. Bunlardan birisi Reuse-Release Equivalence (tekrar kullanım ve sürüm eşitliği) prensibidir.



Resim 7.11 Paket A (package) paket B ye bağımlıdır, çünkü paket A içindeki sınıf ya da sınıflar paket B den bir veya birden fazla sınıfı kullanıyorlar

Bir proje bünyesinde değişik modüllerden oluşan bir yazılım sistemini implemente etmek için çalıştığımızı düşünelim. Her modülden ayrı bir ekip sorumlu olsun. Üzerinde çalıştığımız modülün implementasyonunu yapabilmek için büyük bir ihtimalle diğer modülleri kullanmamız gerekecektir. Örneğin veri

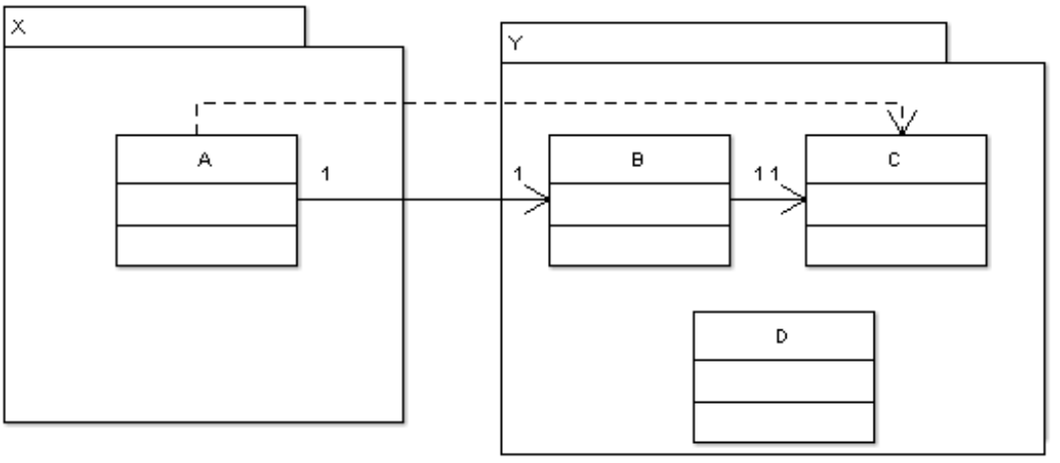
tabanı üzerinde işlem yapmamızı sağlayacak bir modülü başka bir ekip implemente etmiş olabilir. O ekip veri tabanı üzerindeki işlemler için gerekli tüm sınıfları dao isimli bir paket içine yerleştirmiş olabilir. Bizim bu paket içindeki sınıfları tekrar kullanabilmemiz (reuse) için belirli şartların yerine gelmesi gerekmektedir. Bunlar:

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

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

Common Reuse Principle (CRP) – Ortak Yeniden Kullanım Prensibi

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



Resim 7.12 Sınıf A dolaylı olarak sınıf C ye bağımlıdır.

Resim 7.12 de paket X ve paket Y arasındaki bağımlılık yer almaktadır. Sınıf A paket Y de bulunan B sınıfını kullanmaktadır. B sınıfı aynı paket içindeki C sınıfını kullanmaktadır. Bu durumda X paketinde bulunan A sınıfı dolaylı olarak Y paketinde bulunan C sınıfına bağımlı hale gelmektedir. C sınıfı üzerinde yapılan her değişiklik A sınıfını doğrudan etkileyecektir. B ve C sınıfları beraberce kullanıldıkları için aynı paket içinde yer almaları gerekir.

CRP aynı zaman hangi sınıfların paket içine konmaması gerektiğine de açıklık getirir. Birbirine bağımlılıkları olmayan, birbirini kullanmayan sınıfların aynı paket içinde olmaları sakıncalıdır. Örneğin resim 7.12 da yer alan Y paketi içindeki D sınıfı, Y paketinde bulunan hiçbir sınıf tarafından kullanılmamaktadır. Bu durumda D sınıfının Y paketinde olmaması gerekir. Eğer D sınıfı değişikliğe uğrar ve Y paketinin yeni bir sürümü (release) oluşursa, bu aslında D sınıfı ile hiçbir ilgisi olmayan X paketindeki A sınıfını etkileyecektir, çünkü Y paketinin yeni sürümü X paketinin ve dolaylı olarak A sınıfının tekrar gözden geçirilmesini zorunlu kılacaktır.

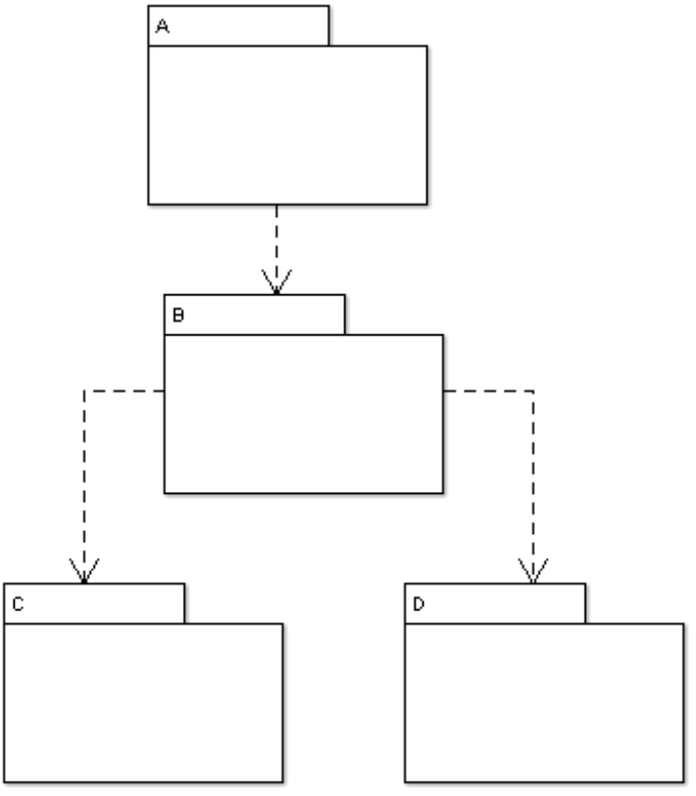
Common Closure Principle (CCP) – Ortak Kapama Prensibi

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

bir sebebi olmalıdır. CCP uygulandığı taktirde sistemin bakılabilirliği artırılır ve test ve yeni sürüm için harcanan zaman ve emek azaltılır.

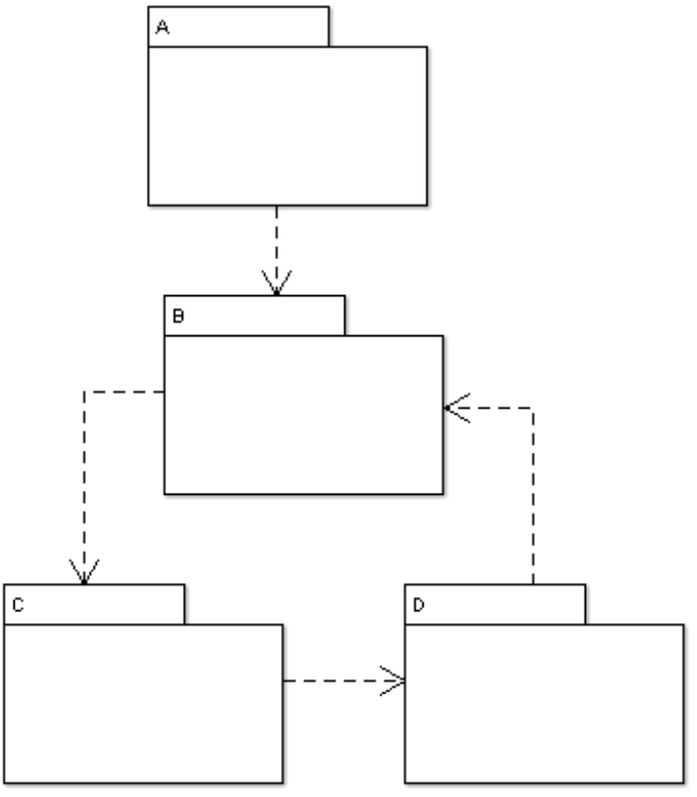
Acyclic Dependency Principle (ADP) – Çevrimsiz Bağımlılık Prensibi

Paketler arasında, çevrim olacak şekilde bağımlılık oluşması sakıncalıdır.



Resim 7.13 Çevrim olmayan paket yapısı

Resim 7.13 de yer alan paket yapısında paketlere arası bağımlılıklar mevcuttur, ama çevrim yoktur. Paket A paket B ve dolaylı olarak Paket C ve Paket D ye bağımlıdır. Paket A da bulunan bir sınıfı test etmek istediğimiz zaman Paket B, C ve D yi teste dahil etmemiz gerekmektedir. Bu durum Paket C ve D için geçerli değildir. Bu paketler diğer paketlerden izole edilmiş bir şekilde test edilebilir, çünkü diğer paketlere hiçbir bağımlılıkları yoktur.

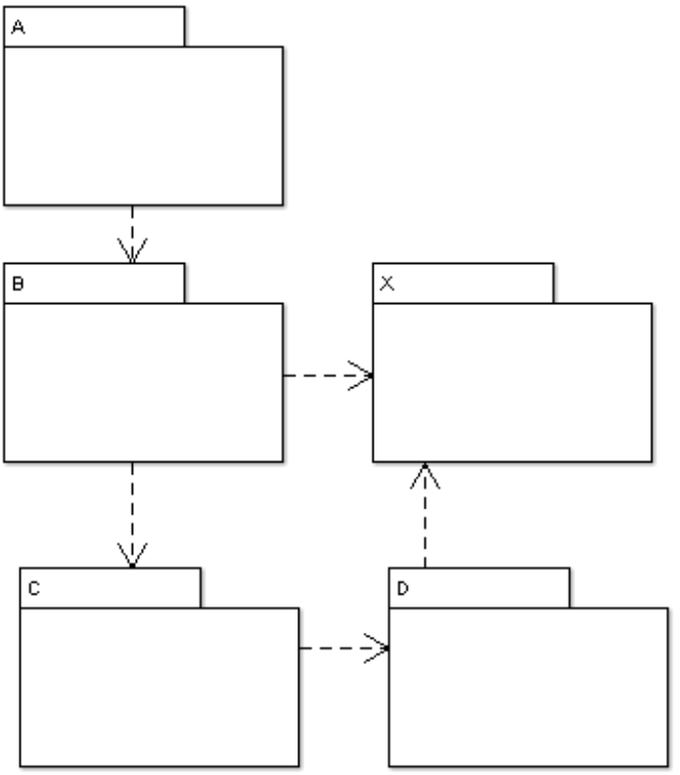


Resim 7.14 Çevrim olan paket yapısı

Resim 7.14 de yer alan paket yapısında anomalik bir durum vardır. Eğer B paketinden yola çıkarak bağımlılık yönünde ilerlersek, tekrar bu pakete D paketi üzerinden ulaşabiliriz, yani burada bir çevrim söz konusudur. Bu durumda D paketini test edebilmek için B ve C paketlerine ihtiyacımız vardır.

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

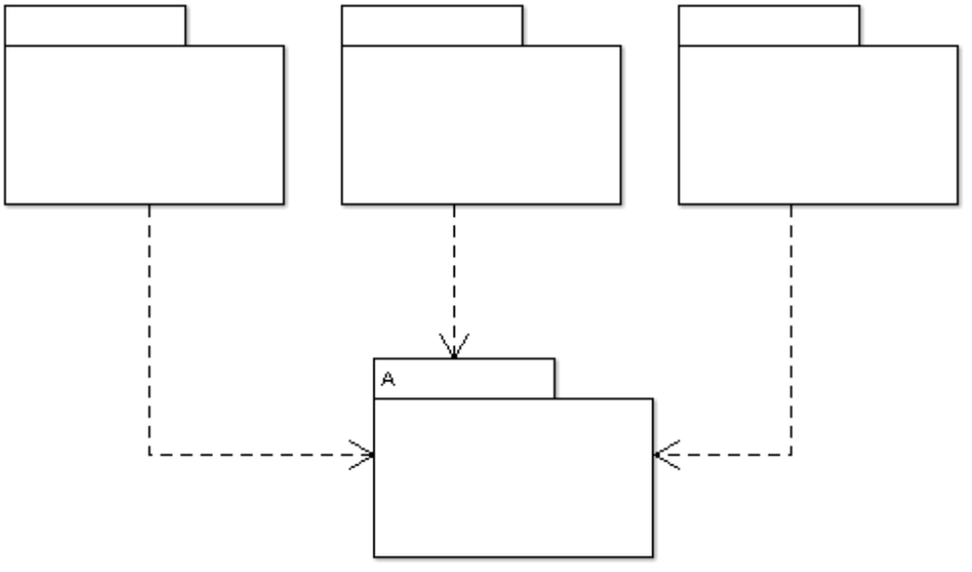
Paketler arası çevrimi yok etmek için paket B ve D nin bağımlı olacağı yeni bir paket oluşturulabilir. Bunun bir örneğini resim 7.15 de görmekteyiz.



Resim 7.15 Çevrim olmayan paket yapısı

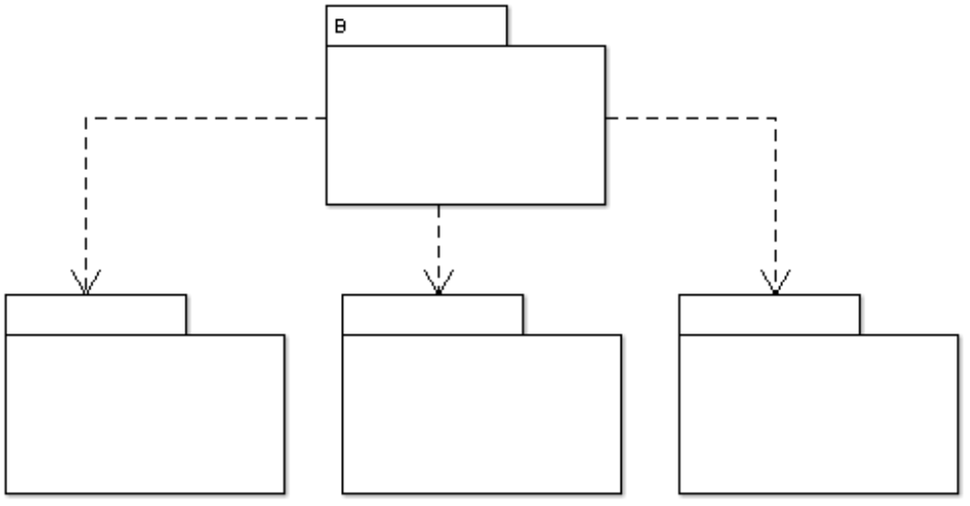
Stable Dependencies Principle (SDP) – Stabil Bağımlılıklar Prensibi

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



Resim 7.16 A stabil bir pakettir

Resim 7.16 de yer alan Paket A ya üç bağımlı paket vardır. Bu durumda paket A nın değişikliğe uğramaması için üç sebebi vardır. Paket A bu üç pakete karşı sorumludur (responsible). Bu üç paketi rahatsız etmemek için sık sık değişmemesi gerekir. Bu yüzden paket A stabil ve detiştirilmesi kolay olmayan bir pakettir. Bunun yanı sıra paket A bağımsızdır (independent), çünkü hiçbir dış bağımlılığı olmadığı için değiştirilmek zorunda kalmayabilir.



Resim 7.17 B stabil olmayan bir pakettir

Resim 7.17 de durum farklıdır. Üç değişik pakete bağımlılığı bulunan paket B yüksek derecede kırılgan bir yapıdadır. Paket B ye bağımlı olan başka bir paket bulunmamaktadır. Bu yüzden paket B sorumluluğu olmayan (irresponsible) bir pakettir. Bağımlı olduğu her üç paket bünyesinde oluşacak bir değişiklik paket B yi direk etkiler. Bu yüzden paket B bağımlı (dependent) bir pakettir.

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

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

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

Bu metriği testpit etmek için kullanabileceğimiz formül şu şekildedir:

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

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

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

Stable Abstractions Principle (SAP) – Stabil Soyutluk Prensibi

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

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

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

Soyutluk ölçülebilir bir metriktir. Bu metriği tespit etmek için

kullanabileceğimiz formül şu şekildedir:

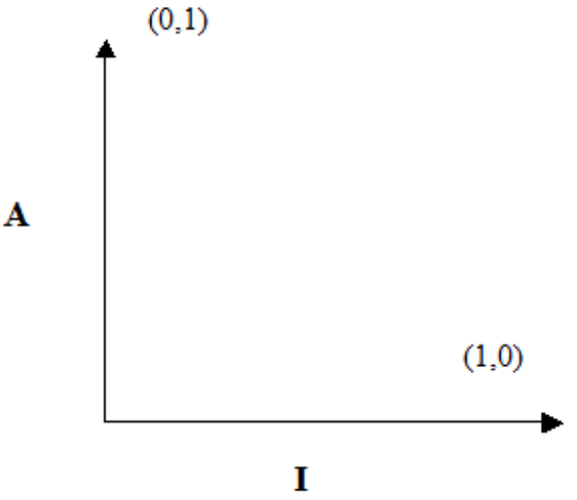
$$A = \frac{Na}{Nc}$$

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

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

Soyutluk (A) ve Instability (I) Arasındaki İlişki

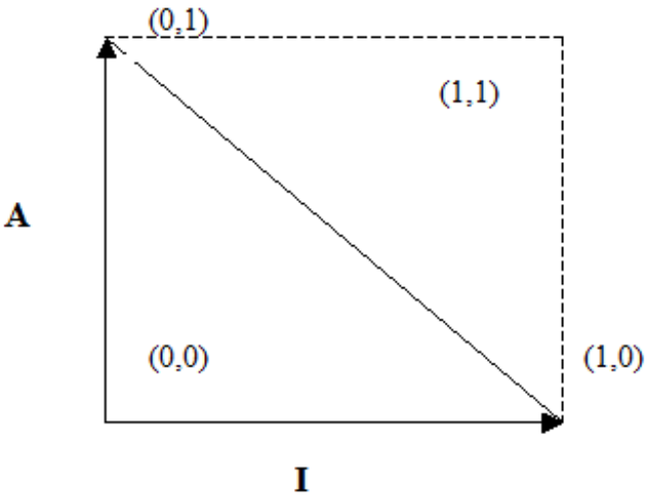
A ve I metriği arasındaki ilişkiyi incelemek için resim 7.18 de yer alan koordinat sistemini düşünebiliriz.



Resim 7.18 A ve I metrikleri

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

değildir.



Resim 7.19 A ve I metrikleri

Paketler bazen uç noktalarda (0,1 – 1,0) yer alamayacağına göre paketler için en uygun koordinatların hangileri olduğunu nasıl tespit edebiliriz? Bunun için diğer uç koordinatları da gözden geçirmemiz gerekiyor. 1,1 koordinatında (resim 7.19) bulunan paketler tamamen soyuttur ve bu paketlere başka paketler tarafından bağımlılıklar mevcut değildir. Böyle paketler kullanılmayan, faydasız paketlerdir. 0,0 koordinatında bulunan paketler tamamen somut ve stabil paketlerdir. Bu paketlerin genişletilmeleri mümkün değildir, çünkü paket içinde soyut sınıflar bulunmamaktadır. Ayrıca bu koordinattaki paketler stabil olduğundan değiştirilmeleri güçtür.

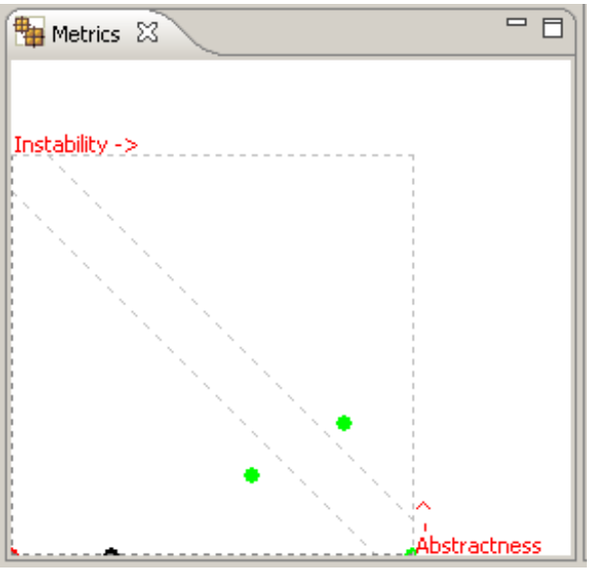
Görüldüğü gibi 1,1 ve 0,0 koordinatlarında ya da bu koordinatlara yakın bir alanda bulunan paketler tasarım eksikliklerine sahiptir. Paketler için en ideal yer 0,1 ve 1,0 koordinatları yanı sıra, bu koordinatları birleştiren çizginin yakınlarında bir yerdir. Bu çizgiye main sequence ismi verilmektedir.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

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

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

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

On beşinci bölümde yakından inceleyeceğimiz JDepend ile A, I ve D metriklerini tespit etmek mümkündür. Resim 7.20 de onuncu bölümde implementasyonunu gerçekleştireceğimiz login modülü için JDepend tarafından oluşturulan main sequence grafiği yer almaktadır.



Resim 7.20 JDepend

Tasarım Şablonları (Design Patterns)

Yazılım esnasında tekrar eden sorunları çözmek için kullanılan ve tekrar kullanılabilir tipte kod yazılımını destekleyen bir ya da birden fazla sınıftan oluşmuş modül ve program parçalarına tasarım şablonu denir. Tasarım şablonları, programcılar tarafından edindikleri tecrübeler doğrultusunda oluşmuş kalıplardır. Bu kalıplar sorunu tanımlayarak, çözümü için gerekli atılması gereken adımları ihtiva ederler. Kullanıcı kalıbı, tanımlanmış sorunu çözmek için tekrar tekeri icat etmek zorunda kalmadan kullanabilir.

Tasarım şablonları aşağıda yer alan ortak özelliklere sahiptir:

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

Program bakımı ve geliştirilmesi için ilk yazılım sürecinden daha çok enerji sarf edilir. Bu yüzden yazılım esnasında esnek bir yapının ve mimarinin oluşturulmasına dikkat edilmesi gerekmektedir. Esnek mimariler için değişik

türde tasarım şablonları kullanılabilir. En basit ve uygulaması kolay bir tasarım şablonunun kullanılması, hiçbir tasarım şablonu kullanılmamasından daha iyidir. İyi bir yazılım mühendisi olabilmek için tasarım şablonları ve kullanım alanları hakkında ihtisas yapmış olmak gerekmektedir.

Tasarım Şablonu Neden Kullanılır?

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

Tasarım Şablonu Kategorileri

Tasarım şablonları değişik kategorilere ayrılır. Bunlar:

- Oluşturucu tasarım şablonları (creational patterns)
- Yapısal tasarım şablonları (structual patterns)
- Davranışsal tasarım şablonlar (behavioral patterns)

Oluşturucu

- Abstract Factory (soyut fabrika)
- Builder (inşaatçı)
- Factory Method (fabrika)
- Prototype (prototip)
- Singleton (yalnızlık)

Yapısal

- Adapter
- Bridge (köprü)
- Facade (cephe)

- Decorator (dekoratör)
- Composite (kompozit)
- Flyweight (sinek siklet)
- Proxy (vekil)

Davranışsal

- Command (komut)
- Memento (hatıra)
- Strategy (strateji)
- Iterator (tekrarlayıcı)
- State (durum)
- Chain Of Responsibility (sorumluluk zinciri)
- Mediator (aracı)
- Observer (gözlemci)
- Template Method (şablon metot)
- Visitor (ziyaretçi)

Abstract Factory (Soyut Fabrika)

Aynı sınıf ailesine ait nesnelerin oluşturulmasında kullanılır. Kullanılan alt sınıfları gizliyerek, transparen olarak kullanılmalarını mümkün kılar.

Builder (İnşaatçi)

Kompleks yapıdakı bir nesneyi değişik parçaları bir araya getirerek oluşturmada kullanılır. Birden fazla adım içeren nesne üretim sürecinde değişik parçalar birleştirilir ve istenilen tipte nesne oluşturulur.

Factory Method (Fabrika)

Somut sınıflara bağımlı kalmadan bu sınıflardan nesneler oluşturmak için Factory tasarım şablonu kullanılır.

Prototype (Prototip)

Sistem içinde kullanılan bazı nesnelerin oluşturulmaları büyük ve değişik kaynakları kullandıklarından dolayı zaman alıcı olabilir. Bu gibi nesneleri new operatörü ile yeniden oluşturmak yerine, Prototype tasarım şablonu kullanılarak mevcut bir nesneden klonlanabilir. Bu şekilde oluşan nesne bir prototiptir ve set metotları kullanılarak istenilen özelliklere göre yapılandırılabilir.

Singleton (Tekillik)

Bazı şartlar altında bir sınıftan sadece bir nesnenin oluşturulması ve oluşturulan bu nesnenin tüm sistemde kullanılması gerekebilir. Singleton tasarım şablonu kullanılarak bir sınıftan sadece bir nesnenin oluşturulması sağlanabilir.

Adapter

Adapter tasarım şablonu yardımı ile sistemde mevcut bulunan bir sınıfın sunduğu interface (sınıf metotları) başka bir sınıf tarafından kullanılabilir şekilde değiştirilir (adapte edilir). Bu adapter yardımı ile birbiriyle beraber çalışamayacak durumda olan sınıflar birlikte çalışabilir hale getirilir.

Bridge (Köprü)

Bridge tasarım şablonu modelleme esnasında oluşan soyut oluşumlar ve bunların implementasyonunu ayırmak için kullanılır. Bu yöntem sayesinde sınıf hiyerarşileri daha esnek bir hale getirilebilir, çünkü üst sınıflar bünyelerinde barındırdıkları soyut metotları bir interface sınıfına taşıyarak, alt sınıfların istedikleri bir implementasyonu kullanmalarına izin verirler.

Facade (Cephe)

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

Decorator (Dekoratör)

Mevcut bir sınıf hiyerarşisini ya da sınıfın yapısını değiştirmeden oluşturulan nesnelere yeni özelliklerin eklenme işlemini gerçekleştirmek için decorator tasarım şablonu kullanılır.

Composite (Kompozit)

Composite tasarım şablonu bir sistemin bütünü ve parçaları arasındaki ilişkileri modellemek için kullanılır. Sistemin bütününü oluşturan parçalar kendi içlerinde alt parçalardan oluşabilir. Composite tasarım şablonu, kullanıcı sınıfın, sistem, sistemin parçaları ve alt parçalar arasında ayrım yapmadan

nesneleri kullanmasına izin verir. Bu şekilde sistem yazılımı ve kullanımı daha sadeleştirilmiş olur.

Flyweight (Sinek Siklet)

Flyweight tasarım şablonu kullanılarak, kullanılan nesne adedi aşağıya çekilebilir.

Proxy (Vekil)

Bir nesnenin kullanımını kontrol etmek için Proxy tasarım şablonu ile korunması gereken nesneye vekilen bir nesne oluşturulur.

Command (Komut)

Bir nesne üzerinde bir işleminin nasıl yapıldığını bilmediğimiz ya da kullanılmak istenen nesneyi tanımadığımız durumlarda, command tasarım şablonu ile yapılmak istenen işlemi bir nesneye dönüştürerek, alıcı nesne tarafından işlemin yerine getirilmesi sağlayabiliriz.

Memento (Hatıra)

Bir nesneyi daha önce sahip olduğu bir duruma tekrar dönüştürebilmek için Memento tasarım şablonu kullanılır.

Strategy (Strateji)

Bir işlemi yerine getirmek için birden fazla yöntem (algoritma) mevcut olabilir. Yerine göre bir yöntem seçip, uygulamak için Strategy tasarım şablonu kullanılır. Her yöntem (algoritma) bir sınıf içinde implemente edilir.

Iterator (Tekrarlayıcı)

Iterator tasarım şablonu ile bir listede yer alan nesnelere sırayla, listenin yapısını ve çalışma tarzını bilmek zorunluluğu olmadan erişilir ve bu nesneler üzerinde işlem yapılır.

State (Durum)

State tasarım şablonu kullanarak bir nesnenin davranışı, sahip olduğu değerler değiştiği zaman değiştirilebilir. Bu durumda sanki nesne sahip olduğu sınıfı değiştirmiş gibi olacaktır.

Chain Of Responsibility (Sorumluluk Zinciri)

Chain of responsibility sorumluluk zinciri anlamına gelmektedir. Sisteme gönderilen bir istediğin (komut) hangi nesne tarafından cevaplanması gerektiğini bilmediğimiz durumlarda ya da isteği yapan nesne ve servis sağlayan nesne arasında sıkı bir bağ oluşmasını engellememiz gerektiğinde Chain of Responsibility tasarım şablonu kullanılır. Bu tasarım şablonunda servis sağlayan ilgili tüm nesneler bir kolye üzerindeki boncuklar gibi birbirleriyle ilişkili hale getirilir. Bir nesne zincirdeki kendinden sonraki nesneyi tanır ve isteği kendi cevaplayamadığı durumda kendinden sonraki nesneye iletir. Bu işlem zincirde bulunan doğru servis sağlayıcı nesneyi bulana kadar devam eder.

Mediator (Aracı)

Mediator tasarım şablonunu nesnelerin yönetimi ve aralarındaki iletişim merkezi bir noktadan koordinasyonu için kullanılır. Bu nesneler arasındaki bağı azaltır ve sadece bir sınıfı iletişimi koordine etmekle sorumlu kılar.

Observer (Gözlemci)

Sistem bünyesinde bir nesnede meydana gelen değişikliklerden haberdar olmak isteyen diğer nesneler olabilir. Bu durumda haberdar olmak isteyen nesneler abone olarak, abone oldukları nesnede meydana gelen değişikliklerden haberdar edilirler. Abone olan nesne aboneliğini iptal ederek, abone olduğu nesne ile arasındaki ilişkiyi sonlandırabilir.

Template Method (Şablon Metot)

Template method ile bir algoritma için gerekli işlemler soyut olarak tanımlanır. Alt sınıflar algoritma için gerekli bir ya da birden fazla işlemi kendi bünyelerinde implemente ederek, kullanılan algoritmanın kendi istekleri doğrultusunda çalışmasını sağlayabilirler.

Visitor (Ziyaretçi)

Visitor tasarım şablonu bir sınıf hiyerarşisinde yer alan sınıflar üzerinde değişiklik yapmadan, bu sınıflara yeni metotların eklenmesini kolaylaştırır. İstenilen metot bir visitor sınıfında implemente edilir.

8. Bölüm

Birim Testleri

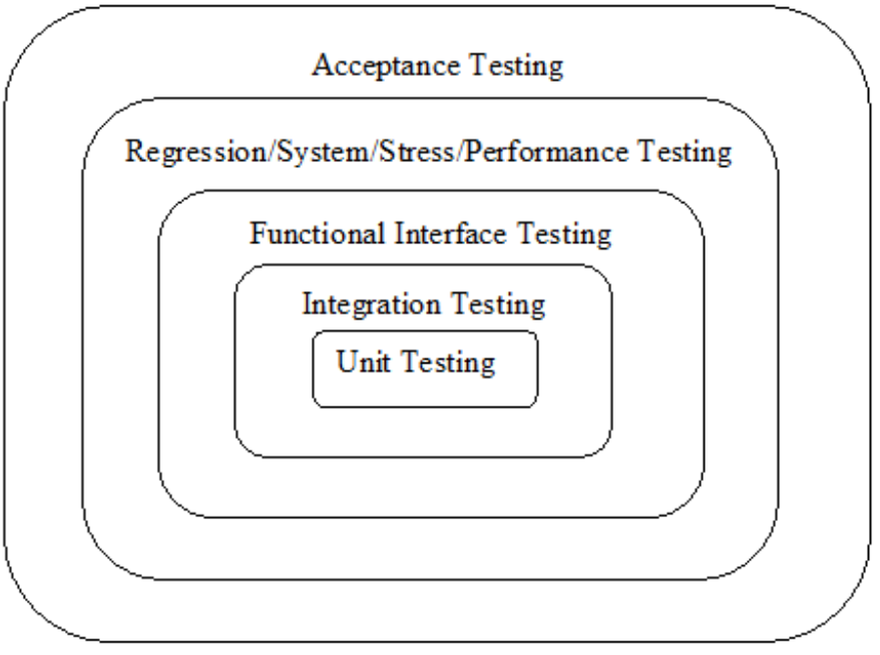
Giriş

Yazılım sürecinde oluşturulan sistemin kalite kontrolü birim testleri ile yapılır. Bu bölümde birim testlerin nasıl hazırlandığını yakından inceleyeceğiz. Bir sonraki bölümünde yer alan test güdümlü yazılımı (test driven development – tdd) uygulayabilmek için birim test konseptlerinin bilinmesi gerekmektedir.

Java tabanlı sistemlerde birim testleri JUnit çatısı kullanılarak hazırlanır. Java'da birim testleri yazabilmek için JUnit çatısından faydalanacağız.

Değişik türde testler oluşturmak mümkündür. Bunlar:

- Birim testleri (Unit Testing)
- Entegrasyon testleri (Integration testing)
- Arayüz (interface) testleri (Functional Interface Testing)
- Regresyon testleri (Regression Testing)
- Onay/kabul testleri (Acceptance Testing)
- Sistem testleri (System Testing)
- Stres testleri (Stress Testing)
- Performans testleri (Performance Testing)



Programcılar sisteme eklenen her Java sınıfı için bir JUnit test sınıfı oluştururlar. Test sınıfında, test edilen sınıfın her metodu için bir test oluşturulur. Birim (unit) ismi buradan gelmektedir. Birim testleri ile kendi içinde bütün olan bir kod birimi test edilir. Bunlar genellikle sınıfların

metotlarıdır. Test sınıfı kullanılarak, test edilen sınıfın işlevlerini doğru olarak yerine getirip, getirmediği test edilir. Metot bazında hazırlanan testlere birim testleri ismini veriyoruz. Sınıf üzerinde yapılan her değişiklik ardından o sınıf için hazırlanan birim testleri çalıştırılarak, yapılan değişikliğin yan etkilerin olup, olmadığı kontrol edilir. Testlerin olumlu netice vermesi durumda, sınıfın görevini hatasız yerine getirdiği ispat edilmiş olur. Bu açıdan bakıldığında birim testleri programcılar için kodun kalitesini korumak ve daha ileri götürebilmek için vazgeçilmezdir. Sistem üzerinde yapılan her değişiklik yan etkilere sebep olabileceği için sistemin her zaman çalışır durumda olduğunu birim testleri ile kontrol edebiliriz. Onlarca ya da yüzlerce sınıfın olduğu bir programı, her değişikliğin ardından elden olarak kontrol etmek imkansız olduğu için otomatik çalışabilen birim testlerine ihtiyacımız vardır. Daha sonra yakından inceleyeceğimiz JUnit çatısı ile otomatik test sürümü gerçekleştirilir.

Birçok komponentten oluşan bir sistemde, komponentler arası entegrasyonu test etmek için entegrasyon testleri oluşturulur. Entegrasyon testleri hakkında detaya girmeden önce, birim testleri hakkında bir açıklama daha yapma gereği duyuyorum. Birim testleri, test edilen sınıfları kullandıkları diğer sınıflardan bağımsız olarak test ederler. Örneğin bir sınıf veri tabanından veri edinmek için bir servis sınıfını kullanıyorsa, birim testinde bu servis sınıfı kullanılmaz, çünkü bu veri tabanının çalışır durumda olmasını gerektirir. Eğer birim testi içinde veri tabanı kullanılıyorsa, bu birim testi değil, entegrasyon testidir, çünkü test edilen sınıf ile veri tabanı arasındaki ilişki dolaylı olarak test edilmektedir. Daha öncede belirttiğim gibi birim testleri metot bazında ve sınıfın dış dünyaya olan bağımlılıklarından bağımsız olarak gerçekleştirilir. Amacımız bir metot içinde bulunan kod satırlarının işlevini test etmektir. Bunun için veri tabanına bağlantı oluşturulması gerekmez. Test edilen sınıfın bağımlılıklarını ortadan kaldırabilmek için Mock nesneler kullanılır. Bir Mock nesne ile servis sınıfı işlevini yerine getiriyormuşcasına simüle edilir. Birim testinde test edilen sınıf servis sınıfı yerine onun yerine geçmiş olan Mock nesnesini kullanarak işlevini yerine getirir.

Entegrasyon testlerinde Mock nesneleri kullanılmaz. Entegrasyon testlerindeki ana amaç sistemin değişik bölümlerinin (subsystem) entegre edilerek işlevlerini kontrol etmektir. Entegrasyon testlerinde test edilen sınıflar için gerekli tüm altyapı (veri tabanı, email sunucusu vs.) çalışır duruma getirilir ve entegrasyon test edilir.

Sistem komponentleri bir veya birden fazla sınıftan oluşabilir. Komponent kullanımını kolaylaştırmak için interface sınıflar tanımlanır. Komponenti

kullanmak isteyen diğer komponent ve modüller bu interface sınıfına karşı programlanır. Komponentler arası interaksiyon Arayüz (interface) testleri ile test edilir. Bu testlere fonksiyon testleri adı da verilir.

Regresyon bir adım geri atmak anlamına gelmektir. Regresyon yazılım esnasında programın yapılan değişiklikler sonucu çalışır durumdan, çalışmaz bir duruma geçtiği anlamına gelir. Regresyon testleri ile sistemde yapılan değişikliklerin bozulmaları neden olup, olmadığı kontrol edilir. Sistem üzerinde yapılan her değişiklik istenmeyen yan etkiler doğurabilir. Her değişikliğin ardından regresyon testleri yapılarak, sistemin bütünlüğü test edilir. Regresyon testlerinde sistem için kullanılan altyapı tanımlanmış bir duruma getirildikten sonra testler uygulanır. Örneğin test öncesi veri tabanı silinerek, test için gerekli veriler tekrar yüklenir. Her test başlangıcında aynı veriler kullanılarak, sistemin nasıl reaksiyon gösterdiği test edilir. Regresyon testlerinin uygulanabilmesi için test öncesinde tüm altyapının başlangıç noktası olarak tanımlanan bir duruma getirilmesi gerekmektedir.

Onay/Kabul (acceptance) testleri ile sistemin bütünü kullanıcı gözüyle test edilir. Bu tür testlerde sistem kara kutu olarak düşünülür. Bu yüzden onay/kabul testlerinin diğer bir ismi kara kutu testleridir (black box testing). Kullanıcının sistemin içinde ne olup bittiğine dair bir bilgisi yoktur. Onun sistemden belirli beklentileri vardır. Bu amaçla sistem ile interaksiyona girer. Onay/kabul testlerinde sistemden beklenen geri dönüm test edilir.

Stres testleri tüm sistemin davranışını sıra dışı şartlar altında test eden testlerdir. Örneğin bir web tabanlı programın eşli zamanlı yüz kullanıcı ile gösterdiği davranış, bu rakam iki yüze çıktığında aynı olmayabilir. Stres testleri ile sistemin belirli kaynaklar ile (donanım, işletim sistemi) stres altındaki davranışı test edilmiş olur.

Performans testleri ile sistemin, tanımlanmış kaynaklar (donanım, işletim sistemi) yardımıyla beklenen performansı ölçülür. Test öncesi sistemden beklenen davranış biçimi tayin edilir. Test sonrası beklentiler test sonuçlarıyla kıyaslanır ve kullanılan kaynakların beklenen davranış için yeterli olup olmadıkları incelenir. Sistemin davranış biçimi kullanılan kaynakların yetersiz olduğunu göstermesi durumunda ya sistem üzerinde değişikliğe gidilir ve sistemin mevcut kaynaklar ile yetinmesi sağlanır ya da kaynaklar genişletilerek sistemin istenilen davranışı edinmesi sağlanır.

Çevik süreçlerde birim testleri büyük önem taşımaktadır. Extreme

Programming bir adım daha ileri giderek, test güdümlü yazılım (Test Driven Development – TDD) konseptini geliştirmiştir. Test güdümlü yazılımda baş rolü birim testleri oynar. Sınıflar oluşturulmadan önce test sınıfları oluşturulur. Bu belki ilk bakışta çok tuhaf bir yaklaşım gibi görünebilir. Var olmayan bir sınıf için nasıl birim testleri yazılabilir diye bir soru akla gelebilir. TDD uygulandığı taktirde oluşturulan testler sadece sistemde olması gereken fonksiyonların programlanmasını sağlar.

Programcılar kafalarında oluşan modelleri program koduna dönüştürürler. Bu süreçte çoğu zaman sistemi kullanıcı gözlüğüyle değil, programcı gözlüğüyle görürler. Bu da belki ilk etapta gerekli olmayan davranışların sisteme eklenmesine sebep verebilir. Birim testlerinde bu durum farklıdır. Örneğin onay/kabul testlerinde tüm sistem bir kullanıcının perspektifinden test edilir. Bu testlerde sistemin mutlaka sahip olması gerektiği davranışlar test edilmiş olur. Eğer onay/kabul testleri oluşturarak yazılım sürecine başlarsak, testin öngördüğü fonksiyonları implemente ederiz. Böylece gereksiz ve belki bir zaman sonra kullanılabileceğini düşünerek oluşturduğumuz fonksiyonlar programlanmaz.

TDD ile testler oluşturulan sisteme paralel olarak oluşur. Sonradan bir sistem için onlarca ya da yüzlerce birim testi oluşturmak çok zor olacağı için birim testlerini oluşturarak yazılama başlamak çok daha mantıklıdır. TDD konseptini bir sonraki bölümde detaylı olarak yakından inceleyeceğiz.

JUnit Konseptleri

Aşağıda yer alan SimpleTest sınıfı basit bir JUnit test sınıfıdır.

Kod 8.1 CustomerManager.java

```
package org.cevikjava.test;

import junit.framework.TestCase;

/**
 * Basit bir JUnit test sınıfı
 *
 * @author Oezcan acar
 *
 */
public class SimpleTest extends TestCase
{
```

```
    public void testCalculate()
    {
        assertTrue( (1+1 == 2) );
    }

    public void setUp() throws Exception
    {
        super.setUp();
    }

    public void tearDown() throws Exception
    {
        super.tearDown();
    }
}
```

Java dilinde birim testleri yazabilmek için junit.jar kütüphanesine ihtiyaç duyulmaktadır. Jar dosyasını http://www.junit.org adresinden temin edebilirsiniz. Bu Jar dosyasında birim testleri için gerekli Java sınıfları bulunmaktadır.

Her JUnit sınıfı junit.framework.TestCase sınıfını genişleterek bir JUnit test sınıfı haline gelir. Her JUnit sınıfında setUp() ve tearDown() metotları yer alır. Test metotlarının isimleri test ile başlar, örneğin testCalculate(). JUnit çatısı JUnit sınıfında tanımlanmış ve test ile başlayan tüm metotları test sınıfı olarak kabul eder ve çalıştırır. Her test metoduna girilmeden önce setUp(), test bittikten sonra tearDown() metodu çalışır. setUp() metodu içinde test için gerekli ortam oluşturur. tearDown() metodunda test için oluşturulmuş kaynaklar yok edilir ve bir nevi testin arkasından temizlik yapılmış olur.

Java 5 ve yeni JUnit sürümleri ile kod 8.1 de yer alan birim testini anotasyonlar yardımı ile şu şekilde şekillendirmek mümkündür:

```
Kod 8.1.1   CustomerManager.java

public class SimpleTest
{
    @Test
    public void buBirTestMetodu()
    {
        assertTrue( (1+1 == 2) );
    }

    @Before
    public void buSetupMetodu() throws Exception
```

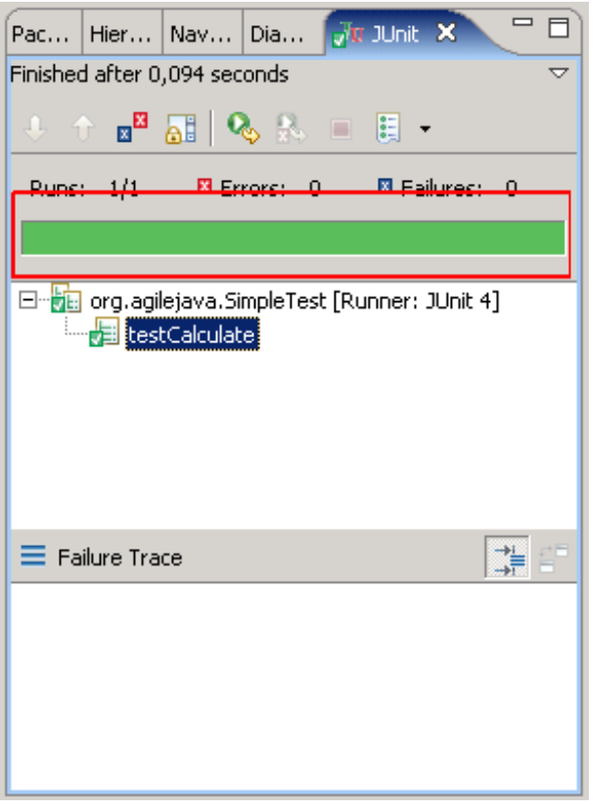
```
    {
    }

    @After
    public void buTearDownMetodu() throws Exception
    {
    }
}
```

Görüldüğü gibi artık birim test sınıfımızın TestCase sınıfını genişletmesi gerekmiyor. Bunun yanı sıra test metodu @Test anotasyonu ile işaretlenmiş ve herhangi bir ismi taşıyabilmekte. @Before ve @After anotasyonları ile test öncesi ve sonrası koşması gereken metotları belirliyoruz. Ayrıca bu metotların isimlerini de istediğimiz şekilde tayin edebiliyoruz.

SimpleTest.testCalculate() metodunu yakından inceleyelim. Bu metot içinde assertTrue ile parantez içinde yer alan değerin doğru (true) olup, olmadığı kontrol edilmektedir. 1+1 = 2 denklemi true sonucunu verecektir. Bu metot içinde biri test konseptinin en basit halini görmekteyiz. Birim testleri sisteminin bütününü oluşturan Java sınıflarından olan beklentilerimizi ifade ederler (SimpleTest herhangi bir Java sınıfı test etmemektedir, sadece örnek olarak verilmiştir). Her Java sınıfı görevini yerine getirdikten sonra bir değer oluşturur ya da belirli bir duruma (state) sahip olur. Bu değer ya da durum JUnit test metodunda assertTrue gibi komutlarla kontrol edilir. Eğer beklediğimiz değer ya da durum oluşmuş ise, o zaman test edilen Java sınıfı üzerine düşen görevi doğru olarak yerine getirmiştir. JUnit çatısında oluşan değer ve durumları kontrol etmek için aşağıda yer alan komutlar tanımlanmıştır:

```
assertEquals
assertFalse
assertNotNull
assertNotSame
assertNull
assertTrue
```

Eclipse bünyesinde JUnit testlerini çalıştırmak için bir plugin bulunmaktadır. Test sınıfı Run As / JUnit Test menüsü üzerinden çalıştırılabilir.



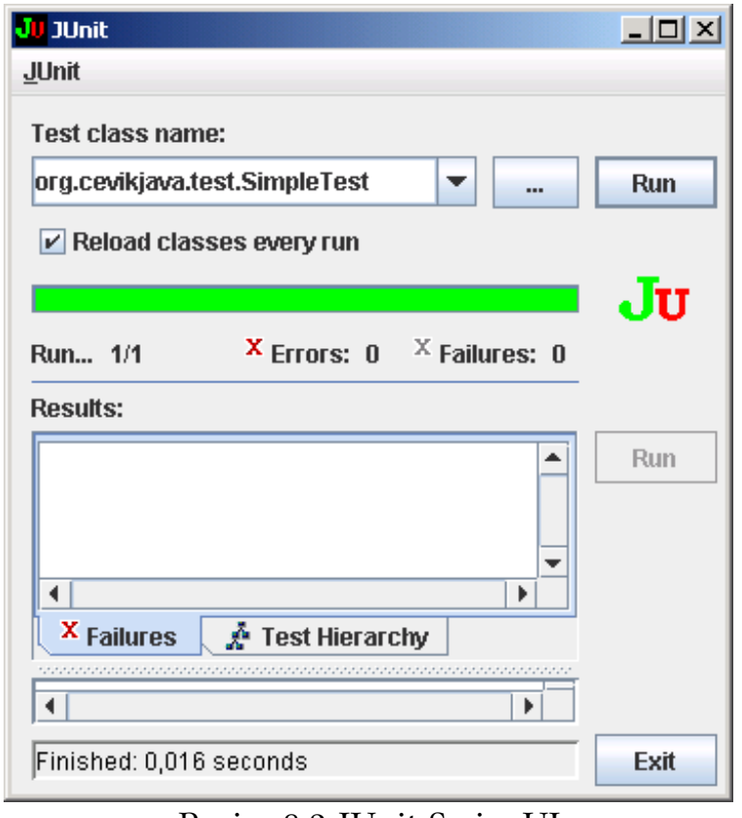
Resim 8.1 JUnit Eclipse Plugin

Resim 8.1 de Eclipse JUnit plugini ve test sonucu görmekteyiz. Bir JUnit test sınıfında yer alan tüm testler hata vermeden çalışmaları durumunda resmin üst bölümünde yer alan panelde yeşil bir çizgi oluşur. Burada trafik lambalarından tanıdığımız yeşil ve kırmızı ışık metafor olarak kullanılmıştır. Hatasız çalışan testler için yeşil ışık, hata oluşması durumunda kırmızı ışık yanar. Amacımız her zaman yeşil ışık görmektir. Sadece bu durumda testler hatasız çalışmış ve test edilen modül beklentilerimizi yerine getirmiş olacaktır.

JUnit testleri junit.jar içinde bulunan junit.swingui.TestRunner ve junit.textui.TestRunner programlarıyla da çalıştırılabilir. İlk program Swing tabanlıdır ve masa üstü ortamında çalışır, ikinci program console üzerinde testleri çalıştırır.

SimpleTest sınıfı SwingUI kullanılarak şu şekilde çalıştırılabilir:

```
C:\Programme\junit> java -cp junit.jar;.junit.swingui.TestRunner
                    org.cevikjava.test.SimpleTest
```

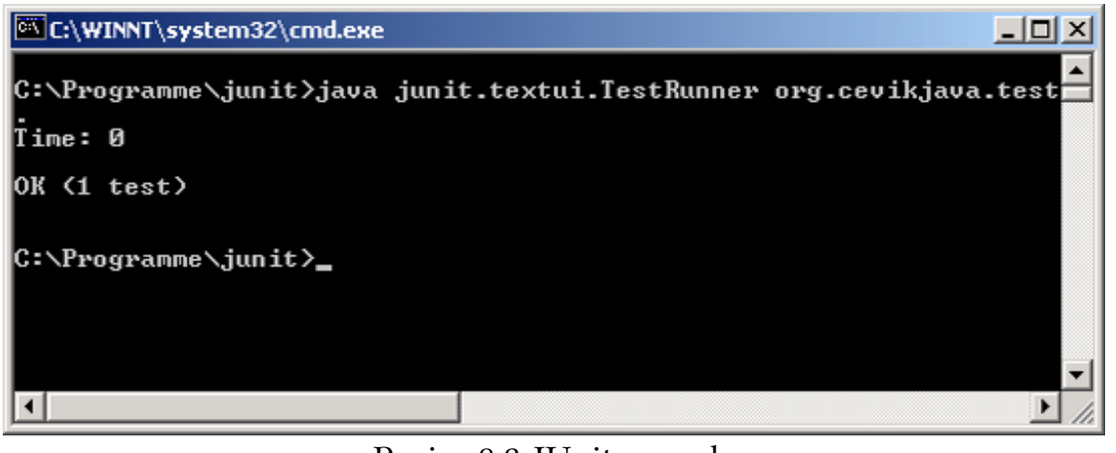


Resim 8.2 JUnit SwingUI

Resim 8.2 de görüldüğü gibi TestRunner swing tabanlı bir programdır. Parametre olarak test sınıfının ismi verildiği taktirde, bu test otomatik olarak çalıştırılır.

Test sınıfları console altında junit.textui.TestRunner programı ile şu şekilde çalıştırılır:

```
C:\Programme\junit> java -cp junit.jar;.junit.textui.TestRunner
                    org.cevikjava.test.SimpleTest
```



Resim 8.3 JUnit console

Birden fazla test sınıfını gruplamak ve aynı anda çalıştırmak için JUnit

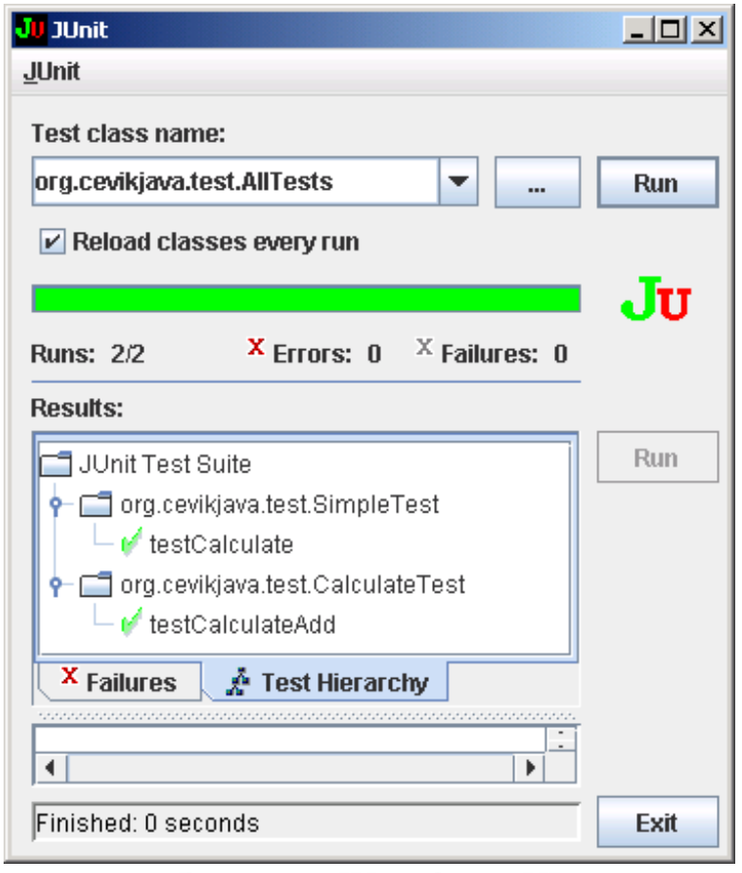
@SuiteClasses anotasyonu kullanılır.

```
Kod 8.2   AllTests.java

package org.cevikjava.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

/**
 * Birden fazla testi gruplamak icin
 * kullanilan Test Suite sinifi
 *
 * @author Oezcan Acar
 *
 */
@RunWith(Suite.class)
@SuiteClasses({ SimpleTest.class, CalculateTest.class })
public final class AllTests {
}
```

Gruplamak istediğimiz testler için AllTests isminde yeni bir sınıf oluşturuyoruz. Kosturmak istediğimiz tüm sınıfların @SuiteClasses bünyesindeki küme parantezi içinde yer alması gerekiyor.

Oluşturduğumuz yeni test suite sınıfını SwingUI programı ile çalıştırıyoruz:

```
C:\Programme\junit> java -cp junit.jar;.junit.swingui.TestRunner
                    org.cevikjava.test.AllTests
```



Resim 8.4 JUnit SwingUI

Test gruplarının yer aldığı daha büyük gruplar oluşturmak ta mümkündür. Genelde her Java package için bir TestSuite oluşturulur. Daha sonra tüm TestSuite ler bir araya getirilerek büyük bir TestSuite grubu oluşturulur. Böylece program için oluşturulan tüm testler bir TestSuite üzerinden çalıştırılabilir.

Kod 8.3 AllTestsSuite.java

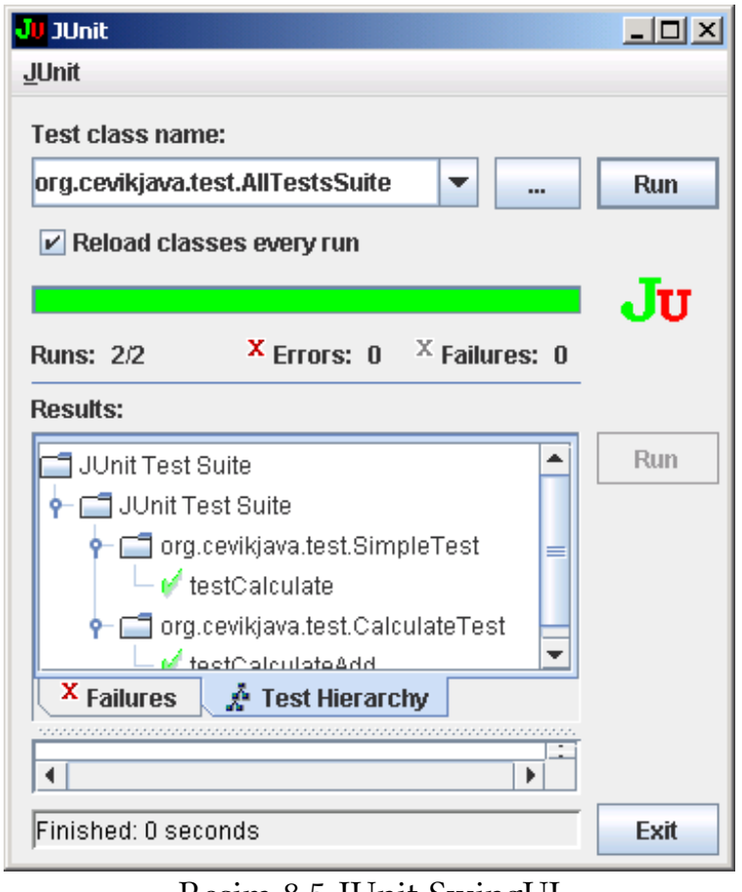
```
package org.cevikjava.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

/**
 * Junit test gruplari (TestSuite) bir araya
 * getirilerek daha büyük gruplar
 * olusturulabilir.
 *
 * @author Oezcan Acar
 *
 */
@RunWith(Suite.class)
```

```
@SuiteClasses({ AllTests.class })
public final class AllTestsSuite {
}
```

Test suite sınıflarından oluşan sınıfları yine bir test suite sınıfı bünyesinde bir araya getirerek, tüm testleri topluca koşturabiliriz. Kod 8.3 de böyle bir test suite sınıfı yer almaktadır.

Test gruplarından oluşan test grubu aşağıdaki şekilde çalıştırılır:

```
C:\Programme\junit> java -cp junit.jar;.junit.swingui.TestRunner
                    org.cevikjava.test.AllTestsSuite
```



Resim 8.5 JUnit SwingUI

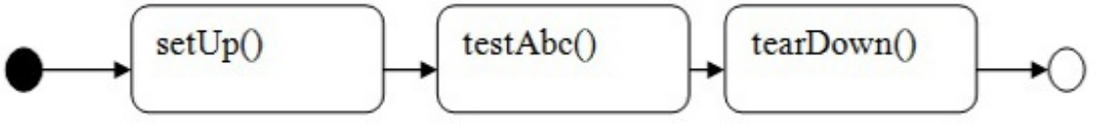
JUnit Anatomisi

@SuiteClasses anotasyonu yardımıyla JUnit testlerini gruplayabileceğimizi gördük. Oluşturulan test suite sınıfları kullanılarak testler gruplanır ve çalıştırılır.

Bazı testler için test öncesi çalışabilecekleri bir ortamın oluşturulması

gerekmektedir. Örneğin test veri tabanına bağlanarak, veri okuyabilir ya da veri depolayabilir. Bu durumda veri tabanına olan bağlantının test öncesi oluşturulması gerekmektedir. Testin içinde çalışabileceği bir ortam için gerekli kaynaklara fikstür (fixture) adı verilmektedir.

Test fikstürü JUnit test sınıfının setUp() metodunda oluşturulur. TestCase sınıfı test başlamadan önce setUp() metodunu çalıştırarak, test için gerekli ortamın oluşmasını sağlar. Test bitiminde otomatik olarak tearDown() metodu çalıştırılarak fikstür temizlenir. Bu her test öncesi yapılan bir işlemdir. Eğer test sınıfımızda beş değişik test metodu bulunuyorsa, JUnit otomatik olarak her test öncesi setUp() ve sonrası tearDown() metotları ile test için gerekli ortamı oluşturacak ve tekrar yok edecektir.



Bir örnek sınıf üzerinde JUnit test konseptlerini yakından inceleyelim. Aşağıda yer alan Calculate sınıfında verilen iki değeri toplayan add() isminde bir metot yer almaktadır.

Kod 8.4 Calculate.java

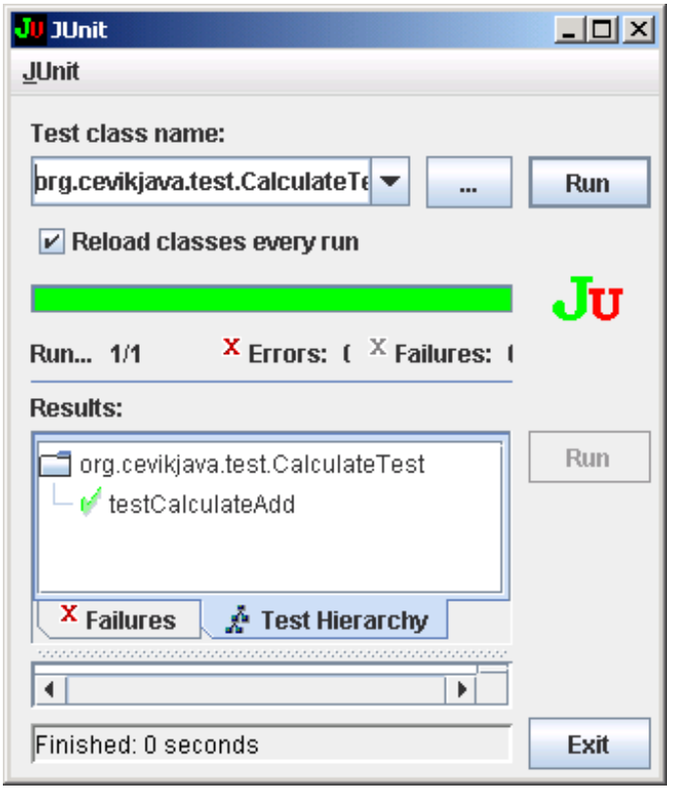
```
package org.cevikjava.test;

public class Calculate
{
    public int add(int a, int b)
    {
        return a+b;
    }
}
```

Bu sınıfı test etmek için CalculateTest isminde bir JUnit test sınıfı oluşturuyoruz.

Kod 8.5 CalculateTest.java

```
package org.cevikjava.test;

import static org.junit.Assert.assertTrue;
import org.junit.After;
import org.junit.Before;
```

```
import org.junit.Test;

public class CalculateTest {

    private Calculate calculate;

    /**
     * add() metodunu test eder.
     *
     */
    @Test
    public void testCalculateAdd() {

        assertTrue(calculate.add(1, 1) == 2);
    }

    /**
     * setUp() bünyesinde bir calculate nesnesi olusturuyoruz.
     */
    @Before
    public void setUp() throws Exception {
        calculate = new Calculate();
    }

    /**
     * teardown() bünyesinde test bittikten sonra
     * calculate nesnesini ortadan kaldiriyoruz.
     */
    @After
    public void tearDown() throws Exception {
        calculate = null;
    }

    public Calculate getCalculate() {
        return calculate;
    }

    public void setCalculate(Calculate calculate) {
        this.calculate = calculate;
    }
}
```



Resim 8.6 SwingUI ile çalıştırılan CalculateTest test sınıfı

CalculateTest.testCalculateAdd() metodunda Calculate sınıfının sahip olduğu add() metodunu test ediyoruz. Test metodunda bu sınıftan olan beklentimizi tanımlıyoruz. Bize göre bu metot 1 ve 1 değerlerini topladıktan sonra 2 değerini geri vermelidir. Eğer sonuç 2 değil ise, add() metodu işlevini doğru olarak yerine getirmemiştir ve bu bir hatadır. Sadece 2 değeri geri verildiği taktirde add() metodu beklenildiği şekilde çalışmış olacaktır. Test sonucunu kontrol etmek için assertTrue() komutunu kullanıyoruz.

Calculate tipinde olan bir sınıf değişkeni tanımlıyoruz. Bu test etmek istediğimiz sınıftır. Testin ihtiyaç duyduğu bir kaynak olduğu için test fikstürü konumundadır. Fikstürü oluşturma işlemi setUp() metodunda gerçekleşir. setUp() metodunda new ile yeni bir Calculate nesnesi oluşturuyoruz. Örneğin Calculate sınıfı işlem için bir veri tabanına ihtiyaç duysaydı ya da başka bir sınıfı kullansaydı, fikstür içine Calculate için gerekli kaynaklar dahil edilir ve `setUp()` metodunda fikstür oluşturulurdu. Test sonuçlandıktan sonra tearDown() metodu işleme girmektedir. Bu metot içinde test için oluşturulan fikstür temizlenir.

Görüldüğü gibi JUnit testleri yazmak o kadar zor bir iş değildir. JUnit testleri ilk etapta programın çalışır durumda olduğunu kanıtlamak ve yazılım kalitesini belirli bir seviyede tutmak için oluşturulur. JUnit testleri ile program hatalarını

lokalize etmek daha kolaydır. Tespit edilen her hatadan sonra JUnit testleri oluşturulduğu taktirde, bu hatanın giderilmesinden sonra herhangi bir sebepten dolayı tekrar ortaya çıkması engellenmiş olur.

JUnit testlerinin faydalarını şu şekilde sıralayabiliriz:

- Kodun daha iyi anlaşılmasını ve takım çalışmasını destekler
- Emma gibi araçlar aracılığıyla kodun hangi bölümlerinin JUnit testleri ile kullanılığı (code coverage) tespit edilebilir. Hiç çalışmayan kod satırlarının lokalizyonu kolaylaşır.
- JUnit testleri programcının oluşturduğu bir nevi dokümentasyondur. JUnit testlerine bakarak, programın nasıl çalıştığı öğrenilebilir.
- JUnit testleri kodun yeniden yapılanması (refactoring) işlemi için programcının cesaretini artırır. Programcı her değişikliğin ardından testleri çalıştırarak, yaptığı değişikliklerin yan etkilerini kontrol edebilir.

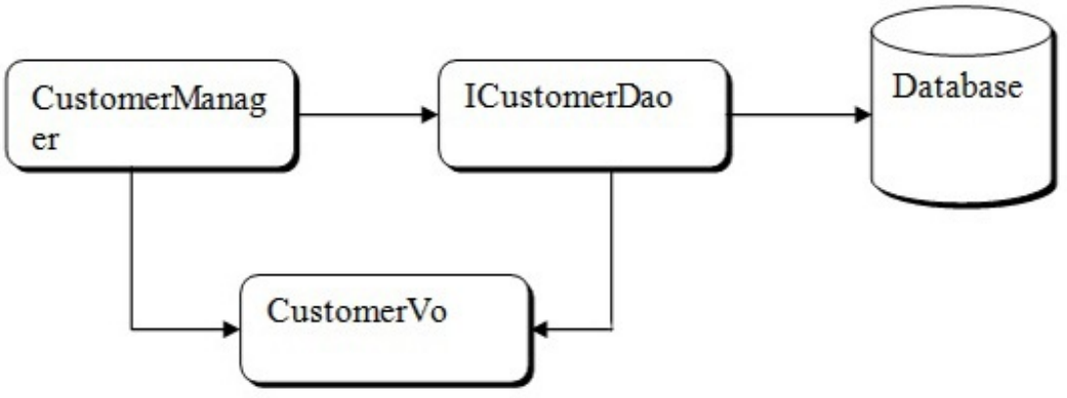
Mock Nesneler

Mock İngilizce sahte ya da taklit anlamına gelmektedir. JUnit testlerinde mock nesneler sıkça kullanılır.

Bir Java sınıfı başka bir Java sınıfını import ettiği taktirde, bu sınıfa bağımlı hale gelir. Bir Java sınıfını test ederken bağımlı olduğu sınıflarında çoğu zaman dikkate alınması gerekmektedir. JUnit testlerinin amacı Java sınıfını ve sahip olduğu tüm bağımlılıkları test etmek değildir. JUnit testlerinde Java sınıfları isole edilmiş olarak düşünülür ve işlevleri test edilir. Mock nesneler kullanılarak test edilen Java sınıfının bağımlılıkları test esnasında varmış gibi simüle edilir. Test edilen Java sınıfı kullanılan Mock ve gerçek sınıflar arasında ayrım yapamaz. Bu yüzden nasıl bağımlı olduğu sınıflarla beraber çalışıyorsa, test esnasında da bağımlılıklarını temsil eden mock nesneler ile interaksiyona girer. Mock nesneler nasıl davranmaları gerektiğine dair test öncesi programlanır. Test edilen Java sınıfı mock nesnesinin metotlarını kullanarak kendi üzerine düşen görevi yerine getirir. Test sonunda mock nesnesi incelenerek, programlandığı şekilde kullanılıp, kullanılmadığı incelenir. Eğer düşünüldüğü şekilde kullanıldı ise, test edilen Java sınıfı doğru davranış göstermiş demektir. Aksi taktirde test sınıfında hatalı bir davranış mevcuttur.

Çevik sürecimizde jMock mock kütüphanesini kullanacağız. jMock un güncel sürümünü http://www.jmock.org adresinden edinebilirsiniz.

Bir örnekle jMock un JUnit testlerinde nasıl kullanılabileceğine bir göz atalım. CustomerManager isminde müşteri bilgilerini yöneten bir Java sınıfımız var:



Kod 8.6 CustomerManager.java

```
package org.cevikjava.customer;

public class CustomerManager {
    private ICustomerDao dao;

    public CustomerManager(ICustomerDao _dao) {
        this.dao = _dao;
    }

    public CustomerVo getCustomer(long id) {

        CustomerVo vo = this.dao.getCustomer(id);
        System.out.println("Firstname: " + vo.getFirstname());
        System.out.println("Nme: " + vo.getName());
        return vo;
    }
}
```

CustomerManager sınıfı müşteri bilgilerini edinmek için ICustomerDao isminde bir interface sınıf kullanmaktadır. ICustomerDao şöyle bir yapıya sahiptir:

Kod 8.7 ICustomerDao.java

```
package org.cevikjava.customer;

public interface ICustomerDao {
    CustomerVo getCustomer(long id);
}
```

ICustomerDao bünyesinde getCustomer() isminde bir metod yer almaktadır. Bu metod aracılığıyla id si belli bir müşterinin bilgileri veri tabanında bulunarak CustomerVo nesnesi olarak geri verilmektedir.

Şu an itibariyle ICustomerDao sadece bir interface sınıftır ve implementasyonu yoktur. CustomerManager sınıfı ICustomer interface sınıfına bağımlıdır. CustomerManager.getCustomer() metodunun çalıştırılabilmesi için mutlaka bir ICustomerDao implementasyonuna ihtiyaç duyulmaktadır, çünkü getCustomer() metodu ICustomer.getCustomer() metodunu kullanmaktadır. Bu noktada ne sistemin çalışması mümkündür ne de CustomerManager sınıfı test edilebilir durumda görünüyor. CustomerManager sınıfını nasıl test edebiliriz? Bunu bir mock nesne kullanarak gerçekleştirebiliriz.

Kod 8.8 CustomerManagerTest.java

```
package org.cevikjava.customer;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.junit.Assert.assertThat;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Before;
import org.junit.Test;

public class CustomerManagerTest {

    Mockery context = new Mockery();

    /**
     * Test edilen sinif.
     */
    private CustomerManager manager;

    /**
     * Kullanilan Dao Interface
     */
    private ICustomerDao dao;

    private CustomerVo vo;

    /**
     * setUp() icinde CustomerManager ve mock nesnesi olusturulur.
     */
    @Before
    public void setUp() throws Exception {
```

```
        dao = context.mock(ICustomerDao.class);
        manager = new CustomerManager(dao);
        vo = new CustomerVo();
        vo.setName("Acar");
        vo.setFirstname("Oezcan");
    }

    /**
     * CustomerManager.getCustomer() metodunu test etmek icin
     * kullanilir.
     *
     * CustomerManager sinifi ICustomerDao interface sinifina
     * bagimli olduguicin direk test edilmesi zordur. Bu
     * sebepten dolayi bir mock nesne kullanarak, ICustomer
     * sinifini simule edebiliriz.
     *
     */
    @Test
    public void testGetCustomer() {

        /**
         * Mock nesneyi beklentilerimiz dogrultusunda
         * programliyoruz
         */

        context.checking(new Expectations() {
            {
                oneOf(dao).getCustomer(1);
                will(returnValue(vo));
            }
        });

        CustomerVo cusVo = manager.getCustomer(1);

        assertThat(cusVo, notNullValue());
        assertThat(cusVo.getName(), equalTo(vo.getName()));
        assertThat(cusVo.getFirstname(), equalTo(
                        vo.getFirstname()));

        // verify
        context.assertIsSatisfied();
    }
}
```

Ben ilk mock konseptiyle tanıştığımda bir sihirbazın gösterisini seyreden bir seyirci gibi etkilenmiştim. Gerçekte var olmayan bir implementasyon varmış gibi mock nesneler tarafından taklit edilerek bağımlı olan sınıflar kolaylıkla test

edilebilmektedir.

Diğer birim testlerinde olduğu gibi ilk önce bir test fikstürünün oluşturulması gerekiyor. Bunu setUp() metodu içinde yapıyoruz. Test edeceğimiz CustomerManager sınıfı test fikstürünün bir parçasıdır. Bu sınıfın bağımlı olduğu ICustomerDao isminde başka bir sınıf bulunmaktadır. Fikstürü oluştururken fikstür içinde yer alan sınıfların ihtiyaç duyduğu diğer kaynakların oluşturulması gerekmektedir. ICustomerDao sadece bir interface sınıf olduğu için new operatörü ile bu sınıftan bir nesne oluşturmamız mümkün değildir. Bu yüzden Mock konseptini kullanarak, ICustomerDao interface sınıfını implemente eden bir mock nesnesi oluşturacağız.

Mock nesneyi aşağıda yer alan satır ile oluşturuyoruz:

```
dao = context.mock(ICustomerDao.class);
```

Mockery sınıfı jMock çatısında yer alan bir sınıftır. Bu sınıfta yer alan mock() metodunu kullanarak ICustomerDao sınıfını taklit eden bir mock nesnesi oluşturuyoruz.

dao değişkeni bir mock nesnedir ve CustomerManager tarafından kullanılmak üzere konstrüktör parametresi olarak kullanılır. Aşağıda yer alan satır ile bir CustomerManager nesnesi oluşturuyoruz.

```
manager = new CustomerManager(dao);
```

setUp() metodunda test fikstürü için gerekli işlemler tamamlandıktan sonra CustomerManager sınıfını test etmek için testGetCustomer() metodunu tanımlıyoruz. Amacımız CustomerManager sınıfında bulunan getCustomer() metodunu test etmektir.

ICustomerDao interface sınıfı implemente eden mock nesneyi kullanmadan önce programlamamız gerekiyor, çünkü mock nesne sadece bu durumda CustomerManager tarafından nasıl kullanacağını bilebilir ve ona göre reaksiyon gösterir. CustomerManager sınıfın ICustomerDao sınıfı nasıl kullandığını bildiğimiz için, mock nesnemizi aşağıdaki şekilde programlıyoruz:

```
context.checking(new Expectations() {
    {
        oneOf(dao).getCustomer(1);
        will(returnValue(vo));
    }
```

```
});
```

Mock programlama işlemini kısaca şu şekilde açıklayabiliriz. Eğer ICustomerDao.getCustomer(1) şeklinde metot kullanılırsa, vo ismindeki nesneyi geri ver. Vo nesnesi CustomerVo tipindedir ve setUp() metodunda test fikstürünün bir parçası olarak oluşturulmuştur. oneOf(dao).getCustomer() ile getCustomer() metodunun sadece bir kez kullanılabileceğini ifade etmiş oluyoruz. Daha sonra göreceğimiz gibi, eğer CustomerManager programladığımız şekilde mock nesneyi kullanmazsa, bu bir hatadır ve test hata verir.

```
CustomerVo cusVo = manager.getCustomer(1);
assertThat(cusVo, notNullValue());
assertThat(cusVo.getName(), equalTo(vo.getName()));
assertThat(cusVo.getFirstname(), equalTo(vo.getFirstname()));
```

manager.getCustomer(1) metodu kullanılarak cusVo nesnesi oluşturulur. CustomerManager.getCustomer() metoduna bakıldığında, cusVo nesnesinin ICustomerDao tarafından oluşturulduğunu görmekteyiz. Bu durumda kullandığımız mock nesnesinin (dao) cusVo nesnesini oluşturması gerekmektedir. Hangi nesnenin mock nesne tarafından geri verileceğini mock nesnesini programlarken tespit etmiştik:

```
will(returnValue(vo));
```

Mock nesnesi getCustomer(1) metodunu çalıştırdıktan sonra vo ismindeki nesneyi geriye verecektir.

setUp() içinde oluşturduğumuz vo isimli nesne şu şekildedir:

```
vo = new CustomerVo();
vo.setName("Acar");
vo.setFirstname("Oezcan");
```

assert komutlarını kullanarak, manager ve dolaylı olarak mock nesnesi tarafından oluşturulan nesneleri test edebiliriz.

Bu örnekte mock nesnesini CustomerManager.getCustomer() metodunu test edecek şekilde programladık. CustomerManager sınıfı bir mock nesne ile kooperasyon içinde olduğunu bilemez, çünkü mock nesne CustomerManager için gerekli olan ICustomerDao interface sınıfını implemente etmiştir. CustomerManager sınıfı mock nesneyi normal bir ICustomerDao

implementasyonu gibi kullanır. CustomerManager.getCustomer() metodunun doğru çalışabilmesi için mock nesnesini gerekli şekilde programlıyoruz.

Test son bulmadan aşağıda yer alan kod satırı ile mock nesnesinin uygulamamız tarafından düşünülen şekilde kullanılıp, kullanılmadığını kontrol etmemiz gerekmektedir.

```
context.assertIsSatisfied();
```

Eğer CustomerManager mock nesneyi programlandığı gibi kullanmassa test hata verir. Bunun bir örneği aşağıda yer almaktadır.

```
unexpected invocation: iCustomerDao.getCustomer(<2L>)
expectations:
  expected once, never invoked: iCustomerDao.getCustomer(<1L>);
    returns <org.cevikjava.customer.CustomerVo@17055e90>
      parameter 0 did not match: <1L>, because was <2L>
            what happened before this: nothing!
    at org.jmock.api.ExpectationError.unexpected(
                    ExpectationError.java:23)
    at org.jmock.internal.InvocationDispatcher.dispatch(
                        InvocationDispatcher.java:85)
    at org.jmock.Mockery.dispatch(Mockery.java:231)
    at org.jmock.Mockery.access$100(Mockery.java:29)
    at org.jmock.Mockery$MockObject.invoke(Mockery.java:271)
    at org.jmock.internal.InvocationDiverter.invoke(
                        InvocationDiverter.java:27)
    at org.jmock.internal.FakeObjectMethods.invoke(
                        FakeObjectMethods.java:38)
    at org.jmock.internal.SingleThreadedPolicy$1.invoke(
                        SingleThreadedPolicy.java:21)
    at org.jmock.lib.JavaReflectionImposteriser$1.invoke(
                        JavaReflectionImposteriser.java:33)
    at com.sun.proxy.$Proxy5.getCustomer(Unknown Source)
    at org.cevikjava.customer.CustomerManager.getCustomer(
                        CustomerManager.java:18)
    at org.cevikjava.customer.CustomerManagerTest.
                testGetCustomer(CustomerManagerTest.java:68)
```

Yukarda yer lan hata getCustomer() metodunun metot parametresi olarak 1 yerine 2 rakamı kullanıldığında oluşmaktadır. Mock nesnesini programlarken metot parametresi olarak 1 rakamının kullanılacağını teyit etmiştik:

```
context.checking(new Expectations() {
    {
```

```
        oneOf(dao).getCustomer(1);
        will(returnValue(vo));
    }
});
```

Bu durumda test esnasında manager.getCustomer(1); şeklinde kullanım gerekmektedir. Aksi taktirde jMock testi geçersiz olarak durduracak ve yukarda yer alan hata mesajını verecektir.

Mock nesneleri oluşturmak için jMock gibi bir çatıyı kullanmak zorunda değiliz. Kendi mock sınıflarımızı oluşturarak, testlerde kullanabiliriz. Stub adı verilen bu mock nesneler örneğin ICustomerDao interface sınıfı implemente eden dummy bir implementasyon sınıfından olabilir.

Kod 8.9 CustomerDummyImpl.java

```
package org.cevikjava.customer;

public class CustomerDummyDaoImpl implements ICustomerDao {

    public CustomerVo getCustomer(long id) {
        CustomerVo vo = new CustomerVo();
        vo.setName("Acar");
        vo.setFirstname("Oezcan");
        return vo;
    }
}
```

Bu implementasyon sınıfı ICustomerDao interface sınıfını implemente etmektedir. Bu yüzden CustomerManager tarafından kullanılır hale gelir. getCustomer() metodu bir CustomerVo nesnesi oluşturarak, geriye vermektedir. CustomerVo nesnesi gerçek bir ICustomerDao implementasyonunda veri tabanına bağlandıktan sonra edinilen veriler aracılığıyla oluşturulur. Amacımız sadece CustomerManager sınıfını test etmek oldugu için CustomerDummyDaoImpl sınıfı bu amaç için yeterlidir.

Kod 8.10 CustomerManagerTest2.java

```
package org.cevikjava.customer;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.junit.Assert.assertThat;
import org.junit.Before;
import org.junit.Test;
```

```
public class CustomerManagerTest2 {

    /**
     * Test edilen sinif.
     */
    private CustomerManager manager;

    /**
     * Kullanilan Dao Interface
     */
    private ICustomerDao dao;

    private CustomerVo vo;

    /**
     * setUp() icinde CustomerManager ve mock nesnesi olusturulur.
     */
    @Before
    public void setUp() throws Exception {
        dao = new CustomerDummyDaoImpl();
        manager = new CustomerManager(dao);
        vo = new CustomerVo();
        vo.setName("Acar");
        vo.setFirstname("Oezcan");

    }

    /**
     * CustomerManager.getCustomer() metodunu test etmek icin
     * kullanilir.
     *
     * CustomerManager sinifi ICustomerDao interface sinifina
     * bagimli oldugu icin direk test edilmesi zordur. Bu sebepten
     * dolayi bir mock nesne kullanarak, ICustomer sinifini simule
     * edebiliriz.
     *
     */
    @Test
    public void testGetCustomer() {
        CustomerVo cusVo = manager.getCustomer(1);
        assertThat(cusVo, notNullValue());
        assertThat(cusVo.getName(), equalTo(vo.getName()));
        assertThat(cusVo.getFirstname(), equalTo(vo.getFirstname()));
    }

}
```

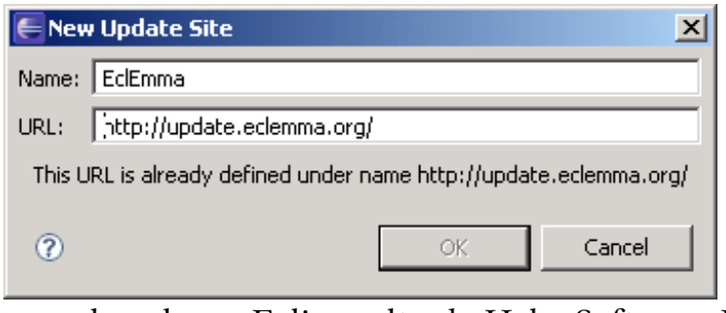
Yeni oluşturduğumuz CustomerManagerTest2 test sınıfında ICustomerDao implementasyonu olarak CustomerDummyDaoImpl stub sınıfını kullanıyoruz. setUp() metodu içinde new CustomerDummyDaoImpl() ile yeni bir dao nesnesi oluşturuyoruz. Dao nesnesi konstrüktör parametresi olarak yeni bir CustomerManager nesnesi oluşturma işleminde kullanılmaktadır. testGetCustomer() test metodunda manager.getCustomer() aracılığıyla CustomerDummyDaolImpl sınıfında oluşturulan CustomerVo nesnesi kullanılmaktadır.

CustomerDummyDaoImpl gibi stub sınıflar basit nesneleri oluşturmak için kullanılabilir. Lakin karmaşık yapıdaki nesneleri taklit etmek için gerçek mock (jMock gibi) nesneleri kullanılmalıdır. Mock nesnelerini programlamak ve davranış biçimlerini tayin etmek karmaşık nesnelerden daha kolay olacaktır.

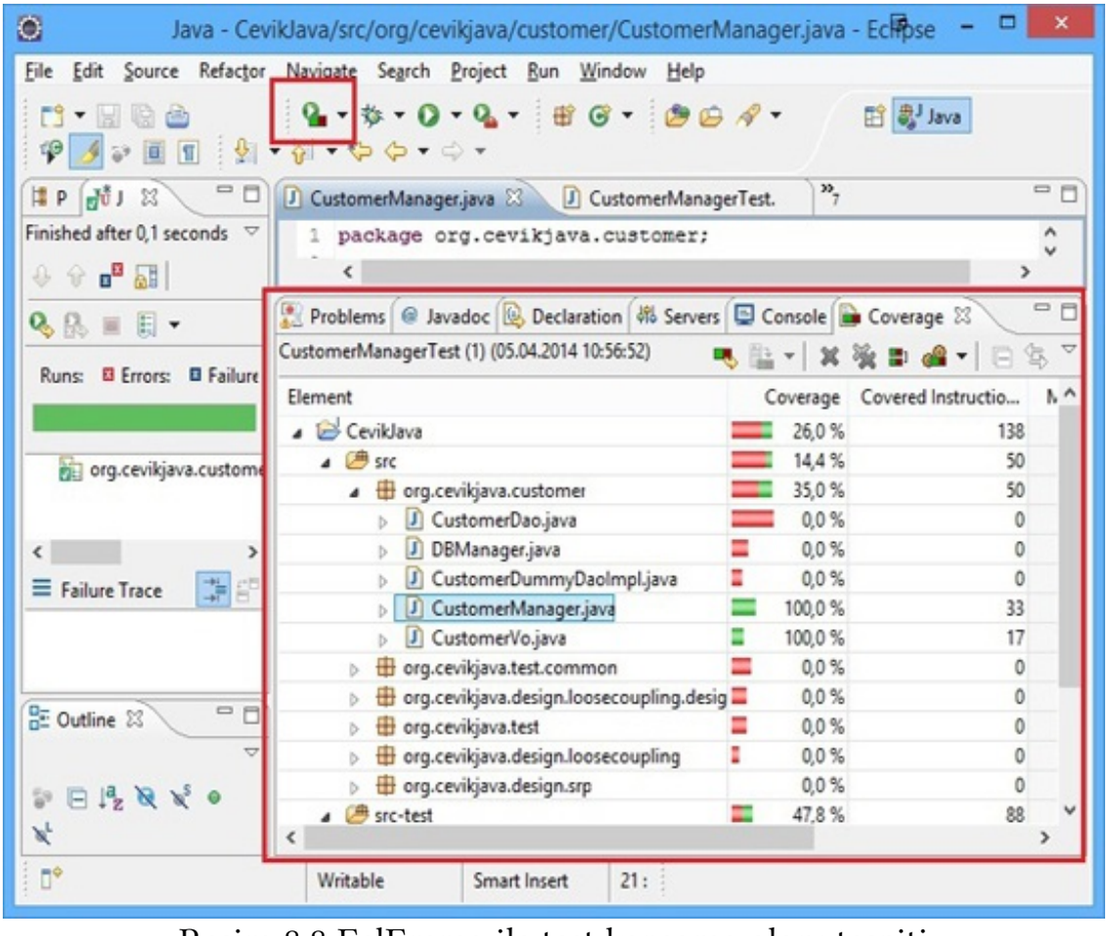
Test Kapsama Alanı (Test Coverage)

JUnit testleri ve gerekli araçlar yardımıyla kodun hangi satırlarının işlem gördüğünü, hangi satırların hiç kullanılmadığını ve ölü olduğunu tespit edebiliriz. Bu işleme testin kapsama alanı tespiti işlemi ismi verilmektedir. Test kapsama alanı ne kadar geniş olursa, programın o kadar büyük bölümü test edilmiş demektir. Test kapsama alanının ölçümü doğru testlerin yapıldığı anlamına gelmez, ama en azından kodun hangi bölümlerinin işlem gördüğünü tespit etmiş oluruz ve gerekli durumlarda test sayısını artırarak test kapsama alanını genişletebiliriz.

Test kapsama alanını EclEmma programıyla tespit etmek mümkündür. EclEmma Eclipse plugindir ve Resim 8.7 de görüldüğü gibi kurulabilir.



Resim 8.7 EclEmma kurulumu Eclipse altında Help>Software Updates > Find and Install bölümünden yapılabilir.



Resim 8.8 EclEmma ile test kapsama alanı tespiti

Plugin kurulduktan sonra üst panelde Resim 8.8 de görüldüğü gibi yeni bir buton oluştu (kare içinde). Bu buton aracılığıyla herhangi bir JUnit test sınıfını seçerek koşturabiliriz. EclEmma arka planda hangi sınıf ve metotların kullanıldığını paket ve sınıf bazında tespit ederek alt panelde görüntüler. Resim 8.8 de yer alan örnekte CustomerManagerTest sınıfı kullanılmıştır. Bu testin proje (CevikJava) genelindeki eriştiği kapsama alanı %26 dır. org.cevikjava.customer paketi için test kapsama alanının %35 olduğunu görüyoruz. Bu paket içinde CustomerManager ve CustomerVo sınıflar %100 test kapsama alanına girmiştir. Bu demektir ki CustomerManagerTest sınıfı CustomerManager ve CustomerVo sınıfında bulunan her satır kodun çalışmasına sebep olmuş ve %100 test kapsamı seviyesine ulaşılmıştır. EclEmma bunun yanı sıra hangi metotların hangi oranda test kapsama alanında olduğunu gösterir. Örneğin CustomerManager sınıfında bulunan CustomerManager() konstrüktör metodu ve getCustomer() metodu %100 işlem görmüştür.

EclEmma test sonucunda test edilen sınıfların işlem gören satırlarını yeşil, işlem görmeyen satırlarını kırmızı renkte gösterir. Bunun bir örneğini Resim 8.9

da görmekteyiz.

```
package org.cevikjava.customer;

/**
 * Müsteri yönetimini yapan sinif.
 *
 * @author Oezcan Acar
 *
 */
public class CustomerManager {
    private ICustomerDao dao;

    public CustomerManager(ICustomerDao _dao) {
        this.dao = _dao;
    }

    public CustomerVo getCustomer(long id) {

        CustomerVo vo = this.dao.getCustomer(id);
        System.out.println("Firstname: " + vo.getFirstname());
        System.out.println("Nme: " + vo.getName());
        return vo;
    }
}
```

Resim 8.9 Test kapsama alanı EclEmma tarafından renkli olarak gösterilir.

9. Bölüm

Test Güdümlü Yazılım

Giriş

Günümüzde kurumsal projelerin büyük bir bölümü geleneksel yazılım metotları ile gerçekleştirilmektedir. Müşteri gereksinimleri en son detayına kadar kağıda döküldükten sonra programcılar tespit edilen gereksinimler doğrultusunda yazılımı gerçekleştirmektedirler. Eğer proje bütçesi yeterli ise yazılım süreci sona erdikten sonra testler hazırlanarak, yazılım sistemi test edilmektedir. Çoğu zaman hiçbir birim testin yapılmadığı sistemlerin firmalar tarafından kritik iş alanlarında kullanıldığını görmek mümkündür. Bu tür yazılım sistemlerinde oluşan hatalar (bug) firmanın sunduğu hizmetleri kısıtlamakta ve en kötü ihtimalle firmanın para kaybetmesine sebep olmaktadır.

Geleneksel tarzda oluşturulan yazılım sistemlerinde oluşan hataları gidermek çok pahalıya mal olabilmektedir, çünkü yazılım bittikten sonra tespit edilen hatalar yazılım sistemindeki tasarım açıklarını gözler önüne serebilir ve bu gibi hataların ortadan kaldırılması ya imkansız ya da çok zor olabilir. Bunun yanı sıra yazılım sona erdikten sonra oluşturulan testlerin test kapsama alanı geniş olmadığı için kodun bazı bölümleri test edilememekte ve böylece hata tespiti zorlaşmaktadır. Bu şekilde yazılım esnasında ortaya çıkmayan hatalar, daha sonra sistem kullanıcıları tarafından keşfedilmektedirler. Bu aşamada geç kalınmıştır: ya sistem çalışmaz durumdadır ya da sistem kullanıcısı istediği işlemi doğru olarak gerçekleştirememiştir. Durumu kısaca şöyle özetleyebiliriz: **“yazılım sistemi müşterinin gereksinimlerini tatmin edecek kaliteye sahip değildir”.**

Ne yazık ki birçok firma için kullandıkları yazılım sistemlerindeki kalite problemleri firmaya zarar verici durumdadır. Bu kalite problemleri bir taraftan oluşan sistem hataları, diğer taraftan kodun bakımı ve geliştirilmesinin zor olmasından kaynaklanmaktadır. Oluşan sistem hataları firmanın giderlerini artırmakta ve yazılım sisteminin istikrarsız ve güvenilmez olmasına sebep vermektedir. Bu sorunların temelinde test konseptlerinin bir yazılım sistemi için **hayat sigortası** olduğunun anlaşılamamış olması yatmaktadır.

Sistem hatalarının oluşmasını engellemek ve kaliteyi yüksek tutabilmek için yeni test konseptlerinin geliştirilmesi ve uygulanması gerekmektedir. XP (Extreme Programming) gibi çevik süreçlerde bu sorunlar çözmek için test güdümlü yazılım (test driven development – TDD) konsepti geliştirilmiştir.

Kent Beck **Test-Driven Development By Example** isimli kitabında test güdümlü yazılımı şu şekilde tanımlıyor:

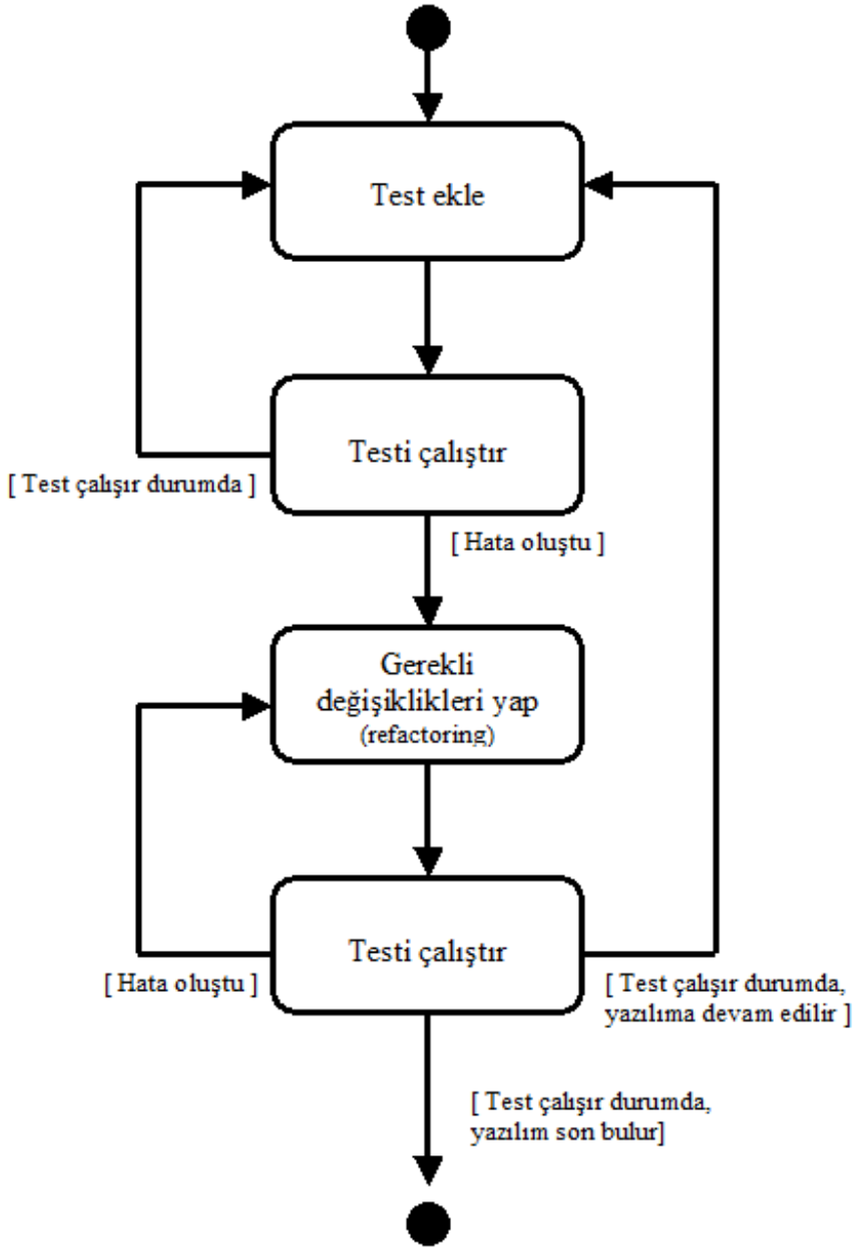
Test-driven development is a set of techniques that any software engineer can follow, which encourage simple design and test suites that inspire confidence.

Test güdümlü yazılım yazılım mühendislerinin kullanabileceği iyi tasarım ve testleri destekleyen ve dolaylı olarak güven artıran metotlardır.

Kent Beck aynı kitapta TDD için şu iki kuralı tanımlıyor:

1. Write a failing automated test before you write any code (program kodu yazmadan önce çalışmaz durumda olan bir test oluştur)
2. Remove duplication (dublike kodu yok et)

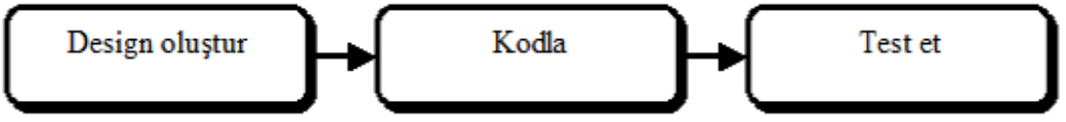
TDD geleneksel yazılım tarzını tamamen tersine çevirir ve yazılıma birim testleri ile başlar. Kent Beck'in tanımladığı gibi herhangi bir satır program kodu oluşturmadan önce bir test sınıfı oluşturularak yazılım işlemine başlanır. TDD için atılması gereken adımlar bir sonraki diyagramda yer almaktadır.



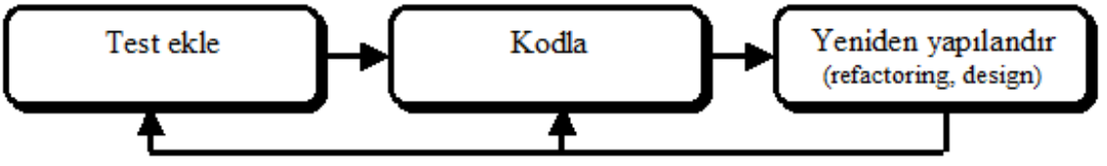
Kent Beck Test-Driven Development By Example isimli kitabında TDD için atılması gereken adımların şu şekilde olması gerektiğini yazıyor:

1. Quickly add a test (hemen bir test oluştur)
2. Run all tests and see the new one fail (testleri çalıştır ve en son eklenen testin çalışmadığını gör)
3. Make a little change (testin çalışması için ufak bir değişiklik yap; refactoring)
4. Run all tests an see them all succeed (testleri çalıştır ve hepsinin hatasız çalışır durumda olduğunu gör)
5. Refactor to remove duplication (dublikasyon - kopyaları yok et)

Herhangi bir program yazılmadan önce bir JUnit test sınıfı oluşturulur. Programcı kafasında olan modeli test edecek şekilde test metotlarını oluşturmaya başlar. Birçok programcı için bu tarz yazılım yapmak alışagelmedik bir yöntemdir, çünkü okulda öğretilen yöntemlerin tam tersini ihtiva etmektedir. Yazılım ihtisası yapılırken, müstakbel programcılara aşağıdaki şemaya göre yazılım yapmaları öğretilir.



İlk önce program için gerekli tasarım (design) oluşturulur. Akabinde bu tasarım implemente edilir. Son aşama olarak sistem hatalarını bulabilmek için testler yapılır. TDD bu süreci tam tersine çevirir:



"Test ekle – kodla - refactor et (yeniden yapılandır)" olarak tanımlayabileceğimiz TDD sürecinde, sadece test metotlarının öngördüğü sınıflar oluşturulur. Burada dikkat edilmesi gereken nokta, gerekli sınıfların en basit şekilde oluşturulmalarıdır. Programcı testi bırakıp, testin gerek duymadığı sınıfları oluşturmamalıdır. Test edilen sınıf metotları ilk etapta null değerini geri verecek ya da hiç bir şey yapmayacak şekilde programlanır. Test çalıştırıldığında hata verecektir, çünkü kullanılan metotlar hiç bir şey yapmamaktadır. Bu noktada test çalışacak şekilde kod üzerinde değişiklik yapılır. Yine metotların en basit şekilde implemente edilmelerine dikkat edilir. Bu işlemler ve yeni testler program için gerekli tüm sınıflar oluşturulana kadar devam eder. Testler ve gerekli sınıflar yavaş yavaş oluştukça, test edilen sınıflar üzerinde gerekli değişiklikler yapılarak, istenilen tasarım oluşturulur.

Bu yeniden yapılandırma işlemi (refactoring) mevcut testlerin desteğiyle çok daha kolay bir hal alır, çünkü yapılan her değişikliğin ardından testler yardımıyla yapılan değişikliklerin yan etkileri kolayca tespit edilebilir. Buradan, refactoring işlemlerinin sağlıklı yapılabilmesi için mutlaka birim testlerinin olması gerektiği sonucunu çıkartıyoruz. Yazılım son bulduğunda sistemde bulunan her sınıf ve metot için JUnit birim testleri var olacaktır.

TDD yöntemiyle oluşturulan testler sadece test olarak algılanmamalıdırlar. Programcının oluşturduğu testler onu programın nasıl kullanıldığı hakkında düşünmeye zorlar. Programcı bir interface sınıfın ne ihtiva etmesi gerektiğini, o interface sınıfını kullanmadığı sürece bilemez. Interface sınıfı oluşturulmadan önce bir test ile işe başlandığı taktirde, programcı sistem kullanıcısı gözüyle testleri oluşturacağı için oluşan interface sınıflarında sadece kullanıcı için gerekli metotlar yer alacaktır.

TDD yeni bir programlama tarzıdır ve doğru uygulandığı taktirde son satırına kadar test edilmiş bir yazılım sisteminin oluşmasını sağlar. TDD ile iyi ve zaman içinde gerekli değişikliklere ayak uyduracak bir design oluşturmak kolaylaşır. TDD uygulandığı taktirde sağladığı avantajları şu şekilde sıralayabiliriz:

- Birim testleri programcıyı sınıfların nasıl kullanıldıklarını düşünmeye zorlar. Kullanıcı gözüyle sınıflara bakıldığı taktirde, daha basit ve kullanışlı yapılandırılmaları kolaylaşır.
- Programcı birim testlerini hazırlarken sistemin nasıl çalışması gerektiğini hayal etmek zorundadır. TDD ile sadece gerekli sınıflar ve metotlar oluşturulur. TDD programcının "belki ilerde kullanılır, bu metodu eklemekte fayda var" tarzı düşünmesini engeller. Böylece TDD proje maliyetini düşürür, çünkü sadece gerekli sınıf ve metotlar için zaman harcanır.
- TDD ile test kapsama alanı (test coverage) geniş olur. Hemen hemen her satır kod test metotları tarafından çalıştırılır.
- Birim testlerinden yola çıkarak oluşturulan sınıflar daha sağlam bir yapıda olurlar. Programcı birim testlerini oluştururken, oluşturduğu sınıfların kullanılış tarzı, olabilecek hatalar ve performansı hakkında düşünür ve buna göre sınıfı yapılandırır.
- Birim testleri koda olan güveni artırır. Kod üzerinde yapılan değişiklikler yan etkilere sebep verebilir. Birim testleri olmadan oluşabilecek yan etkilerin tespiti çok zordur. Birim testleri ile kodun yeniden yapılandırılması (refactoring) kolaylaşır, çünkü oluşabilecek hatalar birim testleri ile lokalize edilebilir.
- JUnit testleri sistemin nasıl çalıştığını gösteren dokümentasyon olarak düşünülebilir. Programcılar birim testlerini inceleyerek sistemin nasıl çalıştığını çok kısa bir zaman içinde öğrenebilirler.
- TDD tarzı programlama programcının debugger ile hata arama zamanını kısaltır ya da tamamen ortadan kaldırır.

Gereksinimlerden Testler Doğar

Programların ham maddesini müşteri gereksinimleri oluşturur. Geleneksel veya TDD tarzı programlar oluşturmak için gereksinimlerden (requirement) yola çıkılır. Gereksinimler tespit edilmeden programın ne yapması gerektiği bilinemez. Bu yüzden yazılım öncesi müşterinin dile getirdiği gereksinimlerin tespiti büyük önem taşımaktadır.

Geleneksel yazılım yöntemlerinde müşteri tarafından dile getirilen gereksinimlerden görevler (task) oluşturulur. Programcı bu görevleri tek tek implemente eder ve gereksinimi tatmin edecek kodu oluşturur. TDD nin bu sürece bakış açısı farklıdır. Görevler yerine gereksinimleri tatmin etmek için testler oluşturulur. Her gereksinim için bir test listesi oluşturulur. Bu testlerden yola çıkılarak uygulama oluşturulur. TDD yapabilmemiz için testlere ihtiyacımız vardır. Bu yüzden görev yerine test mantığında düşünmemiz gerekiyor. Bir gereksinim için üretilmiş tüm testler implemente edildiği zaman gereksinimi tatmin edici kod oluşturulmuş olur. Nasıl bir gereksinim için testler oluşturarak yazılım yapabileceğimizi bir örnekle yakından inceleyelim.

Bir müşterimiz DVD film alım-satım ve kiralama işiyle uğraşmaktadır. Bizi sahip olduğu filmleri yönetebileceği bir programın hazırlanması için görevlendirir. Programdan beklentileri (gereksinimler) şu şekildedir.

- Filmler alfabetik sıraya göre listelenir. Listeye yeni film eklenebilir.
- Film isimleri değiştirilebilir.
- Film ismine ya da oyuncu ismine göre arama yapılabilir.

Bunlar sadece onlarca olabilecek müşteri gereksinimlerinden sadece üç tanesidir. TDD tarzı programın nasıl yapıldığını göstermek için yeterli olacaklardır.

Müşteri filmleri alfabetik olarak bir listenin içinde görmek istemektedir. Bunun yanı sıra listeye yeni film eklenebilmelidir. Böyle bir gereksinim için geleneksel yöntemler kullanılacak olursa, aşağıdaki şekilde bir görev listesi çıkartılabilir. Programcı bu görevleri baz alarak yazılım yapacaktır.

1. Filmlerin konduğu bir liste oluştur. ArrayList ya da Vector olabilir. Filmlerin alfabetik sırası önemli.
2. Filmlerin listelenebileceği bir arayüz oluştur.
3. Yeni bir film eklemek için arayüze “Film Ekle” butonu ekle. Filmler bu

buton üzerinden listeye eklenir.

Şimdi aynı gereksinimleri baz alarak TDD için gerekli testleri oluşturalım ve görev listesi ve test listesi arasındaki farkı inceleyelim.

1. Boş bir listenin büyüklüğü (size) 0 olmalı.
2. Listeye bir film eklendiğinde listenin büyüklüğü 1 olmalı.
3. Listeye iki film eklendiğinde listenin büyüklüğü 2 olmalı.
4. BBB ve AAA ismini taşıyan iki film listeye eklendiğinde AAA ismini taşıyan film listede BBB isimli filmden önce yer almalıdır.

Görev listesini ve test listesini kıyasladığımızda ne dikkatinizi çekiyor? Görev listesi bize sadece ne yapılması gerektiği hakkında bilgi veriyor. Nasıl yapılması gerektiği bilgisini görev listesinden edinemiyoruz. Ayrıca bu liste yazılımda ne kadar ilerlediğimizi gösterecek özellikte değildir, yani hangi görevi tamamladıktan sonra programın % kaçını tamamlamış olacağız, bu konuda fikir sahibi olamıyoruz, çünkü görev tanımlamaları detaylı bir şekilde ne yapılması gerektiğini ihtiva etmiyor ve somut değiller. Test listesini gözden geçirdiğimizde durumun farklı olduğunu görmekteyiz. Testler ilk bakışta ne yapılması gerektiğini ifade edebilen, daha somut ve işletilebilir yapıdadır. Bu yüzden somut testlerden yola çıkarak programı oluşturmak daha mantıklı ve doğal olanıdır.

Bu kıyaslamanın ardından ilk TDD örneğimize geçebiliriz. Programlamak istediğimiz ilk müşteri gereksinimi şöyledir:

Gereksinim 1: Filmler alfabetik sıraya göre listelenir. Listeye yeni film eklenebilir.

İlk testimiz şu şekildedir.

1. Test Boş bir listenin büyüklüğü (size) 0 olmalı.

İlk önce çalışmayan bir test oluşturmamız gerekiyor. TDD nin ilk kuralı budur. Herhangi bir satır program kodu yazmadan önce çalışmayan bir birim testi ile işe başlıyoruz.

Kod 9.1 İlk JUnit test sınıfı

```
package dvd;

public class DvdManagerTest
{
```

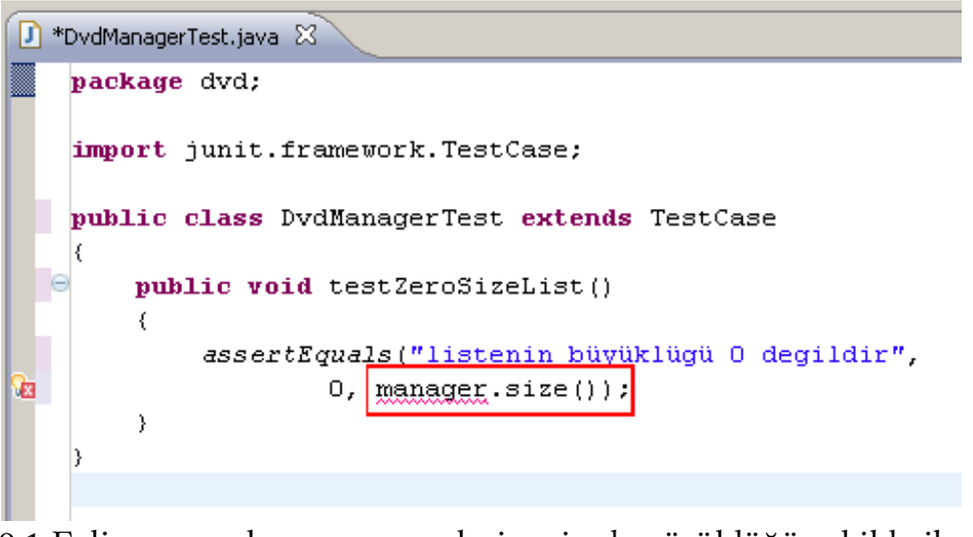
```
    @Test
    public void empty_lists_size_should_be_zero()
    {
        assertEquals("listenin büyüklügü 0 degildir",
                0, manager.size());
    }
}
```

DvdManagerTest isminde bir test sınıfı oluşturuyoruz. İlk test metodunun ismi empty_lists_size_should_be_zero şeklindedir. Test metot isimlerinin seçimi büyük önem taşımaktadır. Metot ismi yapılan işlemi ifade edebilecek şekilde seçilmelidir. Seçtiğimiz metot isminden anlaşıldığı gibi sıfır büyüklükte bir listeyi test ediyoruz.

Sınıflardan önce testleri oluşturduğumuza göre var olmayan sınıfları nasıl test edebiliriz? Bu sorunun cevabı çok basit: kullanmak istediğimiz sınıfların var olduğunu düşünerek! Bu nasıl olur diye soruyor olabilirsiniz. Bu sınıfları nasıl gözümüzde canlandırmamız gerekiyor? Gerekli sınıfları testler tarafından en ideal kalıpta kullanılacak şekilde düşünmemiz gerekiyor. Bu süreç bizi programın bütününü oluşturan sınıfların ilerde diğer kullanıcı sınıflar tarafından nasıl kullanılabileceklerini düşünmeye zorlamaktadır. Eğer kullanıcı perspektifinden bakabilirsek, sınıf içinde gereksiz metotların oluşumunu engellemiş ve diğer sınıfların kullanabileceği temiz bir API (Application Programming Interface) oluşturmuş oluruz. Bu açıdan bakıldığında TDD temiz ve sade API lerin oluşmasını sağlamaktadır, çünkü testler sınıfları bu API ler üzerinden kullanır. Testler olmadan geleneksel yöntemler kullanılarak oluşturulan sınıflarda gereksiz ve daha sonra kullanılabileceği düşünülerek eklenmiş birçok metot bulmak mümkündür. Programcı ne yapması gerektiğini tam olarak görev listesinden algılayamadığı için kendi deneyimleri ve değer yargısı doğrultusunda sınıfları oluşturur. Programcı çoğu zaman sınıfı oluştururken kullanıcı gözüyle sınıfa bakmadığı için sınıf için oluşturulan metotlar (API) yetersiz olabilir sınıf ya da gereğinden fazlasını ihtiva edebilir.

Test metotlarına assert komutları ile başlanması faydalı olacaktır. Assert komutunu sistemden beklentimizi ifade etmek için kullanıyoruz. İlk kullandığımız assert komutunda manager isminde bir nesnenin size() isimli metodunu kullanıyoruz ve beklentimizin sıfır olduğunu belirtiyoruz. Şu ana kadar manager isminde ne bir nesne var ne de bu nesnenin oluşturulacağı bir sınıf. Ama bu yakında değişecek! Ben bu müşteri gereksinimini programlarken ilk testten yola çıkarak DvdManager isminde bir sınıfın olması gerektiğini ve bu sınıfın filmlerin yer aldığı bir listeyi ihtiva ettiğini hayal ettim. Bu sınıftan ilk

beklentimi de assertEquals() komutu ile ifade ettim. DvdManager sınıfı Dvd lerin yer aldığı bir listeyi ihtiva ettiği için size() ismindeki bir metot aracılığıyla listede kaç tane filmin olduğunu tespit edebilirim, yani size() isminde bir metoda ihtiyacımız bulunmaktadır. Görüldüğü gibi şimdiye kadar oluşan program tamamen hayal ürünü ve kafamda oluşmuş durumda. Bu test sınıfının beni DvdManager sınıfının nasıl kullanıldığını düşünmeye zorladığını gördünüz! Tabiri caizse ortada fol yok, yumurta yok, ama DvdManager sınıfının nasıl yapılandırılması gerektiği hakkında kafamızda net bir resim oluştu. İşte TDD nin güzelliği burada yatmaktadır. Alışkanlıklarımız doğrultusunda herhangi bir şey programlama yerine, bizden testler aracılığıyla neyin programlanması gerektiğini daha iyi anlayarak, gerekli metotları programlıyoruz.



Resim 9.1 Eclipse var olmayan nesneleri resimde görüldüğü şekilde ikaz eder.

TDD tarzı yazılım yaparken Eclipse gibi yazılım araçları çok faydalı olmaktadır. Eclipse var olmayan nesneleri ve metotları ikaz ederek, bu sınıfların ve metotların otomatik olarak oluşturulmaları için yardımcı olur. Bizde Eclipse in bu özelliklerinden faydalanarak testimize devam ediyoruz.

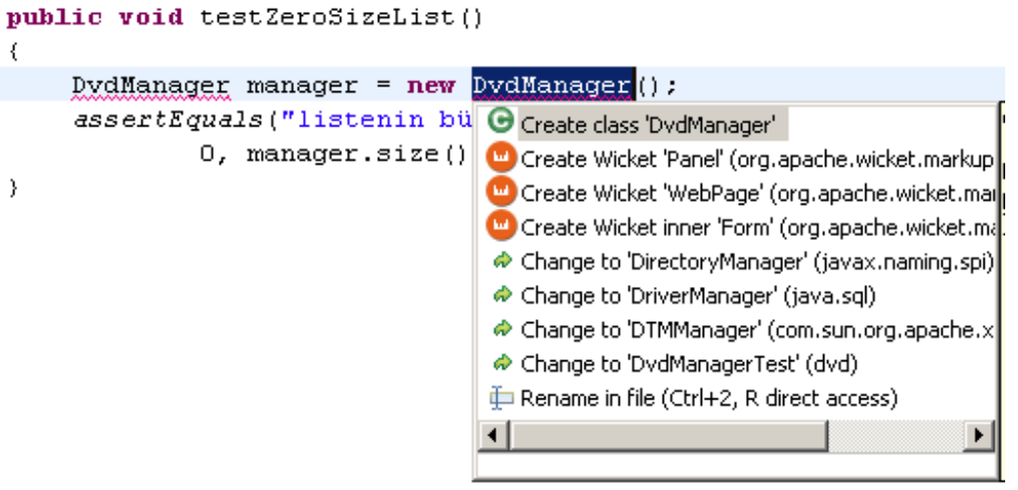
Kod 9.2 İlk JUnit test sınıfı

```
package dvd;

public class DvdManagerTest
{
    @Test
    public void empty_lists_size_should_be_zero()
    {
        DvdManager manager = new DvdManager();
        assertEquals("listenin büyüklügü 0 degildir",
```

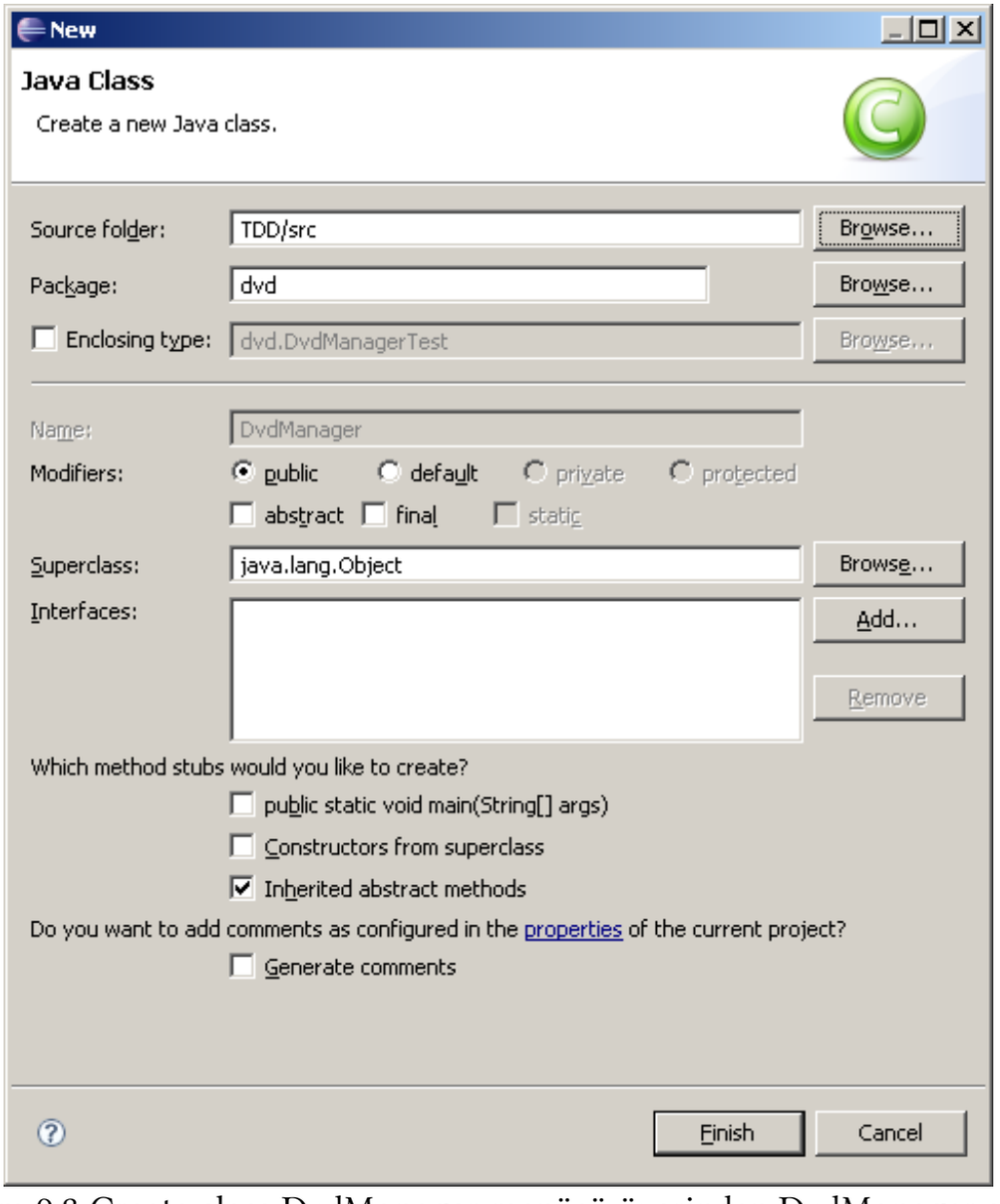
```
                0, manager.size());
    }
}
```

Eclipse in ikazı üzerinde manager isminde bir değişken oluşturuyoruz. Bu değişkenin tipi, oluşturmak istediğimiz DvdManager dir. Eğer Eclipse altında bu örneği uygularsanız, bu noktada Eclipse in DvdManager isminde bir sınıfı bulamadığını ve bu sınıfın altını Resim 9.1 görüldüğü gibi kırmızı renkte çizdiğini göreceksiniz. Bu noktada sadece hayalimizde var olan bir sınıfı veri tipi olarak kullanmaya başladık ve adım adım istediğimiz programı oluşturuyoruz. Sırası gelmişken onuda yazayım: bu şekilde var olmayan sınıfların hayal edilerek yavaş yavaş geliştirilme işlemine TDD terminolojisinde **programming by intention** adı verilmektedir.

Yine Eclipse in bize sunduğu hizmetlerden faydalanarak DvdManager sınıfını oluşturuyoruz:



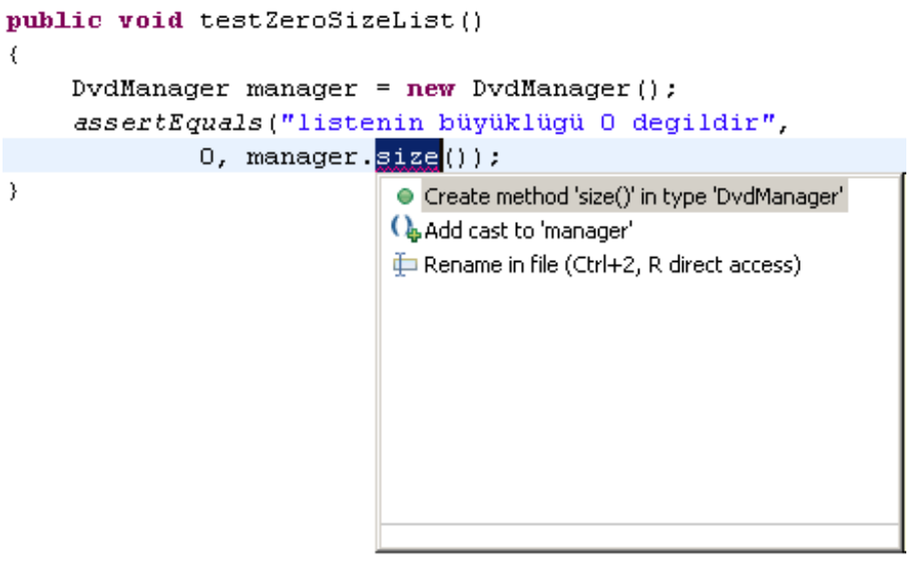
Resim 9.2 Eclipse var olmayan sınıfların oluşturulmasında yardımcı olur



Resim 9.3 Create class DvdManager menüsü üzerinden DvdManager sınıfı oluşturulur

```
Kod 9.3   DvdManager sınıfı

package dvd;

public class DvdManager
{

}
```

Eclipse DvdManager sınıfını kod 9.3 de yer aldığı gibi en basit haliyle oluşturur. Tekrar DvdManagerTest sınıfına göz attığımızda, Eclipse in DvdManager sınıfında size() isminde bir metodun bulunmadığını ikaz ettiğini görürüz.



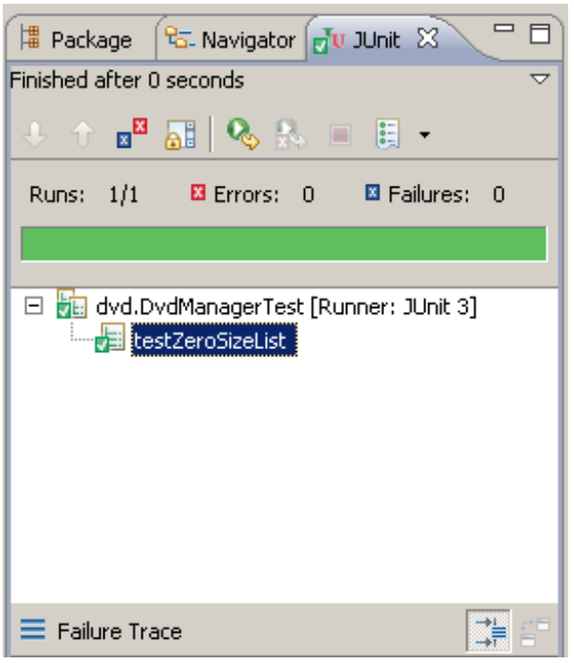
Resim 9.4 size() metodu DvdManager sınıfında henüz bulunmamaktadır

"Create method size() in type DvdManager" opsiyonu üzerinden size() metodunu oluşturuyoruz. Bu değişikliğin ardından DvdManager kod 9.4 deki şekli alacaktır:

Kod 9.4 DvdManager sınıfı

```
package dvd;

public class DvdManager
{
    public int size()
    {
        return 0;
    }
}
```

DvdManager sınıfını ve test içinde kullandığımız size() metodunu oluşturduktan sonra DvdManagerTest sınıfını çalıştırıyoruz. JUnit yeşil ışık yakarak, testin başarılı olduğunu gösteriyor.



Resim 9.5 testZeroSizeList() metodu başarılı bir şekilde çalıştırıldı

DvdManager sınıfında bulunan size() metodunu en basit haliyle oluşturduk. Amacımız gerekli sınıf ve metotları en basit halleriyle oluşturmak ve bir an önce derleme (compile) hatası olmayan bir test sınıfı oluşturmak olmalıdır. Derleme hataları ortadan kalktıktan sonra testi çalıştırarak, sonucuna bakmamız gerekiyor. Genelde ilk oluşturulan testleri çalıştırırken hataların oluşması, yani yeni testin aslında olumlu sonuç vermemesi gerekir. Amacımız ilk etapta derleme hatalarını giderdikten sonra, testi çalıştırıp, olumlu cevap vermediğini görmek ve akabinde gerekli değişiklikleri yapmak ve olumlu neticeye ulaşmak olmalıdır. Bu yüzden testlerin ilk halleriyle hemen olumlu sonuç vermeleri bizi programcı olarak bir şeylerin yanlış gittiği konusunda uyarmalıdır.

DvdManagerTest.empty_lists_size_should_be_zero() metodu DvdManager sınıfını ve size() metodunu oluşturduktan sonra hemen olumlu sonuç verdi. Acaba bir hata mı söz konusu? Hayır! Durumu şöyle açıklayabiliriz: Java gibi kesin veri tiplerinin kullanıldığı dillerde oluşturulan metotların geri verdikleri değerlerin belirli bir veri tipinde olması gerekmektedir. size() metodu int veri tipinde bir değer geri verdiği için burada en basit haliyle 0 değerini geri vermek zorundayız. TDD tarzı sınıf ve ihtiva ettikleri metotları oluştururken ilk etapta 0, false ve null gibi değerleri geri vererek, yanı verinin en basit haliyle işe başlarız. size() metodunu yeni oluşturduğumuz için 0 değerini geri veriyoruz. Tesadüfen test içinde beklentimiz 0 (assertEquals) olduğu için test sınıfı ilk çalıştırılışında olumlu cevap verdi. Bu sadece bir tesadüf! Daha öncede belirttiğim gibi çıkış noktamız her zaman olumlu sonuç vermeyen bir test

metodu olmalıdır. Bu bize sınıflar ve metotlar üzerinde gerekli değişikliklerin yapılması gerektiğini gösterir. Gerekli değişiklikler yapıldıktan sonra test tekrar çalıştırılarak sonuç kontrol edilir. Testten olumlu sonuç alınana kadar sınıf ve metotlar üzerinde değişiklik yapılmaya devam edilir.

2. Test Listeye bir film eklendiğinde listenin büyüklüğü 1 olmalı.

Yeni bir test metodu oluşturarak, ikinci testi implemente etmeye başlıyoruz.

```
Kod 9.5   İkinci test  metodu

@Test
public void when_adding_first_item_to_the_list_size_should_be_one()
{
    assertEquals("listenin büyüklügü 1 degildir", 1, manager.size());
}
```

Test metotları için seçilen isimler büyük önem taşımaktadır. Seçilen metot isimleri, metot içinde olup, bitenleri ifade edebilecek güce sahip olmalıdır İkinci test için when_adding_first_item_to_the_list_size_should_be_one yani bir film eklendikten sonra listenin büyüklüğü şeklinde bir isim seçiyoruz. İlk iş olarak beklentilerimizi ifade etmek için assertEquals komutunu kullanıyoruz. Bu test bünyesinde DvdManager sınıfının hakimiyetinde olan listeye bir film ekledikten sonra listenin büyüklüğünün 1 olması gerekiyor.

Testte ifade ettiğimiz beklentiyi elde edebilmek için listeye bir filmi ekleyebilmemiz gerekiyor. DvdManager sınıfında add() isminde bir metot oluşturmamız gerekmektedir. DvdManager sınıfına atlayıp, add() isminde bir metot oluşturmak yerine, test içinde bu beklentimizi dile getiriyoruz:

```
Kod 9.6   add()  metodunu kullanıyoruz

@Test
public void when_adding_first_item_to_the_list_size_should_be_one()
{
    manager.add(gora);
    assertEquals("listenin büyüklügü 1 degildir", 1, manager.size());
}
```

manager isminde bir değişken olmadığı için derleme hatası oluşmaktadır. Bu sorunu gidermek için kod 9.7 de yer alan eklemeyi yapıyoruz.

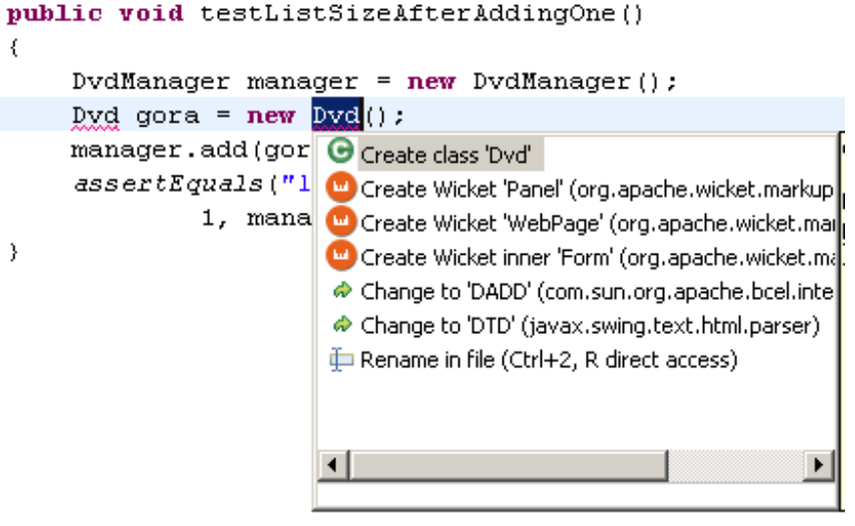
```
Kod 9.7   manager değişkenini tanımlıyoruz
```

```
@Test
public void when_adding_first_item_to_the_list_size_should_be_one()
{
    DvdManager manager = new DvdManager();
    manager.add(gora);
    assertEquals("listenin büyüklügü 1 degildir", 1, manager.size());
}
```

Son olarak Eclipse gora isminde bir değişkenin olmadığı uyarısında bulunuyor. Bu değişken nedir? Film listesine film eklenebileceği için bu değişkenin film (Dvd) tipinde olması gerekiyor. Dvd isminde yeni bir sınıf (veri tipi) tanımlıyoruz.

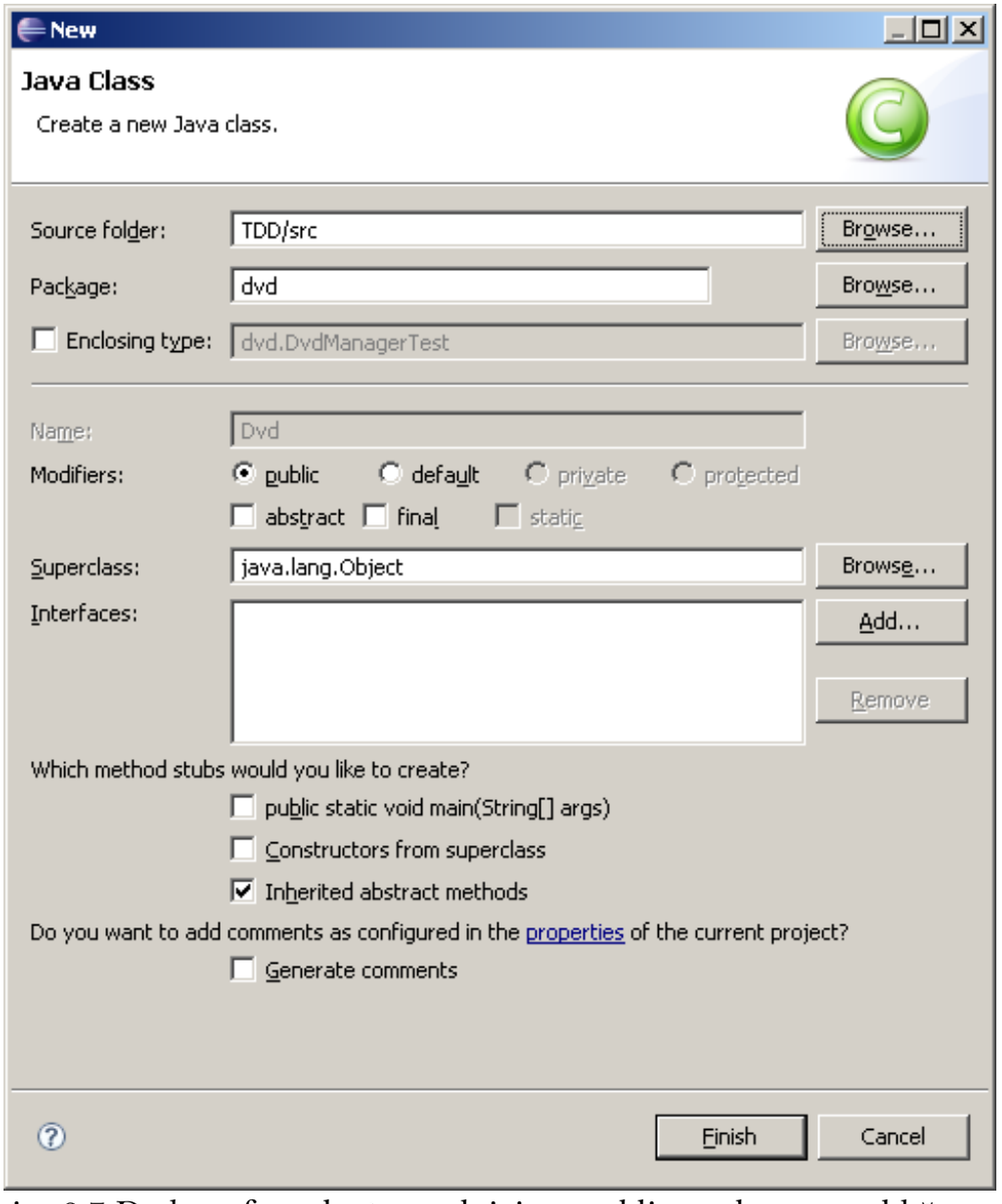
Kod 9.8 Dvd sınıfı ilk kez kullanılıyor

```
@Test
public void when_adding_first_item_to_the_list_size_should_be_one()
{
    DvdManager manager = new DvdManager();
    Dvd gora = new Dvd();
    manager.add(gora);
    assertEquals("listenin büyüklügü 1 degildir",1, manager.size());
}
```



Resim 9.6 Eclipse Dvd sınıfını oluşturmak için yardımcı oluyor

Yine Eclipse bizden yardımlarını esirgemiyor ve Dvd isimli sınıfı oluşturmamıza yardımcı oluyor:



Resim 9.7 Dvd sınıfını oluşturmak için gerekli ayarların yapıldığı panel

Eclipse in yardımıyla Dvd sınıfını en basit haliyle oluşturuyoruz. Bu sayede DvdManager sınıfında yer alan derleme hatalarının bir kısmını gidermiş olduk.

```
Kod 9.9   Dvd sınıfı

package dvd;

public class Dvd
{

}
```

Yeni test metodunda sadece bir derleme hatası kaldı, o da DvdManager sınıfında eksik olan add() metodunun kullanılıyor olması. Bu sınıfı kod 9.10 da

yer aldığı gibi oluşturuyoruz.

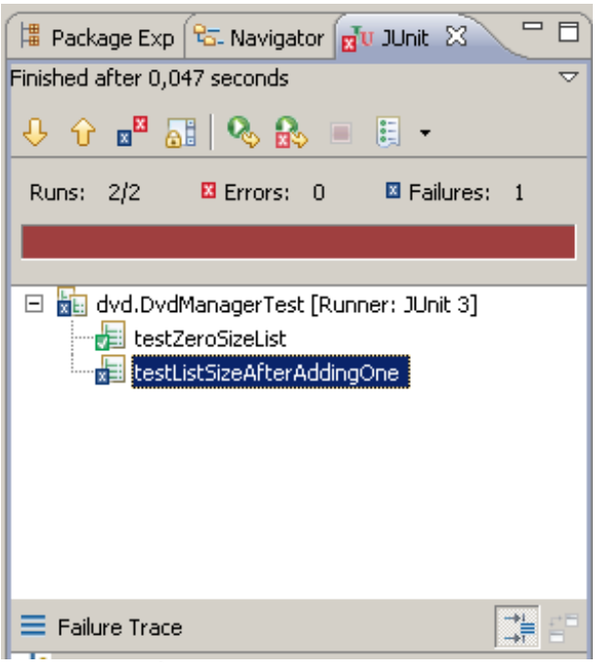
```
Kod 9.10   DvdManager sınıfına add()  metodunu ekliyoruz

package dvd;

public class DvdManager
{
    public int size()
    {
        return 0;
    }

    public void add(Dvd gora)
    {

    }
}
```

add() metodunun geri verdiği değer void olduğu için bu metodun gövdesini boş bırakıyoruz. Unutmayalım: metotları en basit şekilde oluşturuyoruz ve testler doğrultusunda yapılandırıyoruz. add() metodunun en basit hali boş bir gövdeye sahip olmasıdır.

Bu değişiklerin ardından test sınıfında derleme hatası kalmıyor ve testi çalıştırıyoruz. Sonuç düşündüğümüz gibi: çalışmayan bir test metodu ile karşı karşıyayız ve aslında beklentimizde buydu. Şimdi sınıflar üzerinde gerekli değişiklikleri yaparak, testi olumlu sonuç verecek hale getirmeye çalışacağız.

JUnit söyle bir hata mesajı verdi:

junit.framework.AssertionFailedError: listenin büyüklügü 1 degildir expected:<1> but was:<0>



Resim 9.8 İkinci test çalışmaz durumda

Resim 9.8 de görüldüğü gibi ikinci test çalışmaz durumdadır. Bu bizim beklediğimiz bir durumdur. Çalışmayan bir testten yola çıkarak sınıflar üzerinde gerekli değişiklikleri yapabilir ve testi çalışır duruma getirebiliriz. Burada dikkat etmemiz gereken önemli iki husus vardır: Aynı zamanda sadece bir testin çalışmaz durumda olduğuna dikkat etmemiz gerekiyor. Ayrıca oluşturduğumuz yeni testlerin diğer testleri olumsuz etkilemelerini önlemeliyiz. Resim 9.8 de görüldüğü gibi ikinci test çalışmaz durumda iken, birinci test çalışır durumdadır, yani ikinci test beraberinde birinci testi de olumsuz etkilememiştir. İkinci testin derlenmesi ve olumsuz sonuç vermesi için yeterli kod yazdık. Bu noktadan itibaren sınıflar üzerinde gerekli değişiklikleri yaparak, testin olumlu sonuç vermesini sağlayacağız.

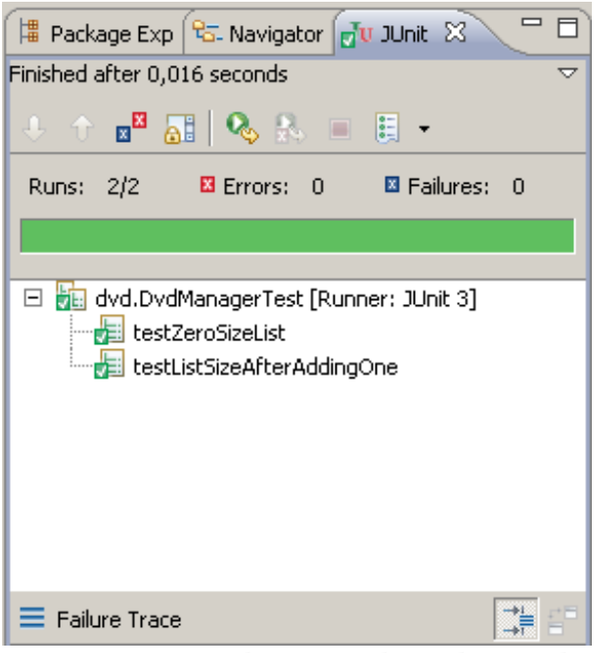
İkinci test olumsuz sonuç vermekte, yani çalışmıyor. Bu testi olumlu sonuç verecek hale getirmek için ne yapmamız gerekiyor? Testten beklentimizi tekrar gözden geçirerek, bu soruya cevap bulabiliriz. Testten beklentimiz şu şekildedir: add() metodunu kullanıp, listeye yeni bir film ekledikten sonra, film listesinin büyüklüğü 1 olmalıdır. Şimdi sonucu elde edebilmek için en kolay işlemin ne olabileceğini beraber düşünelim. Örneğin size() metodunu 1 değerini geri verecek şekilde değiştirebiliriz. Ama bu durumda ilk test çalışmaz hale gelir, çünkü ilk testin beklentisi içinde film bulunmayan bir listenin büyüklüğünün 0 olmasıdır. Eğer size() metodunu bu şekilde değiştirirsek, ilk test çalışmaz hale gelir. Bu sakınılması gereken bir durumdur. Kesinlikle yaptığımız değişikliklerle daha önce hazırladığımız testlerin kırılmalarını engellememiz gerekiyor. listSize

isminde bir değişken tanımlayarak, add() işleminin ardından bu değişkenin 1 değerine yükselmesini sağlayabiliriz. add() metodunu şu şekilde değiştiriyoruz:

```
Kod 9.11   add()  metodu değişiyor

package dvd;

public class DvdManager
{
    private int listSize = 0;

    public int size()
    {
        return listSize;
    }

    public void add(Dvd gora)
    {
        listSize = 1;
    }
}
```

Bu noktada şunu söylediğinizi duyabiliyorum: “Bunlara ne gerek var, bir ArrayList oluşturarak, filmi bu listeye ekler ve size() metodunda bu listenin büyüklüğünü geri verebilirim.” TDD ileriyi düşünmenizi istemiyor. TDD için anlık sorunları çözmeniz yeterli. Bu yüzden mümkün olan en basit implementasyonları oluşturarak ilerlememiz gerekiyor. Benim aklıma gelen en basit çözüm kod 9.11 de yer almaktadır. listSize isminde yeni bir sınıf değişkeni tanımlıyoruz. add() metodu kullanıldıktan sonra bu değişkene 1 değerini veriyoruz. size() metodunda listSize değişkeninin sahip olduğu değeri geri veriyoruz. İkinci testi geçmek için bu yeterlidir. Bu basit implementasyon ile birinci testide kırmamış oluyoruz. Amacımız ikinci testi olumlu sonuç verecek hale getirmektir. Bir sonraki testin gereksinimleri doğrultusunda DvdManager sınıfı üzerinde değişikliklere devam edeceğiz. Ama bunun şimdi sırası değil. İkinci test için ne gerekli ise onu yapıyoruz ve diğer testler için gerekli değişiklikler hakkında şimdilik düşünce sarf etmiyoruz.

Testi çalıştırdığımız zaman her iki testinde olumlu sonuç verdiğini göreceğiz.



Resim 9.9 Her iki test çalışır durumda

Tekrar DvdManager sınıfı implementasyonunu kontrol ediyoruz. add() metodu iyi bir implementasyona sahip değil. Bu şu an için bir sorun teşkil etmiyor. Yeni testler bu metot üzerinde yeniden yapılandırma işlemini destekleyecektir. İkinci testi burada tamamlıyoruz ve üçüncü test ile devam ediyoruz.

3. Test Listeye iki film eklendiğinde listenin büyüklüğü 2 olmalı.

Şimdiye kadar film listemiz için taban testleri oluşturduk. Testler 0 ve 1 filmli listeler için olumlu sonuç vermekte. Şimdi testleri bir adım daha ileri götürerek, 2 filmin bulunduğu film listesini test edelim. Eğer testimiz 2 filmin yer aldığı liste için olumlu sonuç verirse, bu ikiden fazla filminde yer aldığı listenin çalışır durumda olduğunun kanıtıdır.

Üçüncü test için kod 9.12 yer alan test metodunu oluşturuyoruz.

Kod 9.12 Test 3

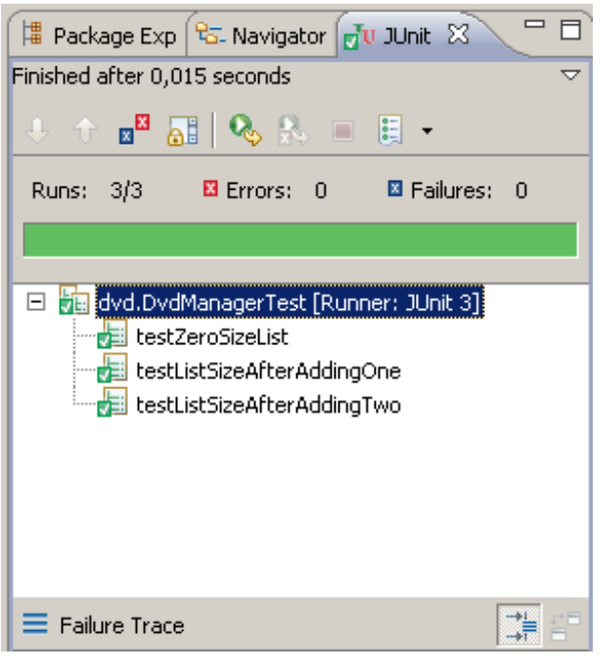
```
@Test
public void when_adding_second_item_list_size_should_be_two()
{
    DvdManager manager = new DvdManager();
    Dvd gora = new Dvd();
    Dvd arog = new Dvd();
    manager.add(gora);
    manager.add(arog);
    assertEquals("listenin büyüklügü 2 degildir", 2, manager.size());
}
```

Doğal olarak bu test çalışır durumda değil! add() metodu her kullanıldığında listSize değişkenine 1 değerini eşitler. Bu yüzden add() metodu iki sefer arka arkaya kullanılmış olsa bile listSize değişkeni 1 değerine sahip olacağından, size() metodu 1 değerini geri verecektir. Bizim beklentimiz 2 olduğu için test olumlu sonuç vermeyecektir.

add() metodunu kod 9.13 de yer aldığı gibi değiştirirsek, test çalışır hale gelecektir.

```
Kod 9.13   add()  metodunda değişiklik yapıyoruz

package dvd;

public class DvdManager
{
    private int listSize = 0;

    public int size()
    {
        return listSize;
    }

    public void add(Dvd gora)
    {
        listSize++;
    }
}
```

Testi tekrar çalıştırdığımızda olumlu sonuç alıyoruz.



Resim 9.10 Her üç test çalışır durumda

Şu ana kadar oluşturduğumuz üç test add() metoduna gönderilen Dvd nesneleri dikkate almadı, çünkü testler bunu gerektirmedi. Sadece testlerin gereksinimleri doğrultusunda sınıf ve metotları yapılandırmamız gerekiyor. Her zaman düşünebileceğimiz en basit implementasyon tarzını seçerek, testleri ve sınıfları oluşturmamız, test ve sınıfların gereksiz veri ve değişkenlerle donatılmasını engelleyecektir.

Diğer testlere geçmeden önce şimdiye kadar oluşturduğumuz testleri bir gözden geçirelim. Oluşturduğumuz testler kod 9.14 de yer almaktadır.

Kod 9.14 DvdManagerTest sınıfı

```
package dvd;

public class DvdManagerTest
{
    @Test
    public void empty_lists_size_should_be_zero()
    {
        DvdManager manager = new DvdManager();
        assertEquals("listenin büyüklügü 0 degildir",
                0, manager.size());
    }

    @Test
    public void when_adding_first_item_to_the_list_size_should_be_one()
    {
        DvdManager manager = new DvdManager();
```

```
        Dvd gora = new Dvd();
        manager.add(gora);
        assertEquals("listenin büyüklügü 1 degildir",
                1, manager.size());
    }

    @Test
    public void when_adding_second_item_list_size_should_be_two()
    {
        DvdManager manager = new DvdManager();
        Dvd gora = new Dvd();
        Dvd arog = new Dvd();
        manager.add(gora);
        manager.add(arog);
        assertEquals("listenin büyüklügü 2 degildir",
                2, manager.size());
    }
}
```

Bu test sınıfını gözden geçirdiğimizde bazı değişkenlerin birden fazla test metodunda kullanıldığını görmekteyiz. Örneğin manager değişkeni DvdManager tipinde olup, her test metodunda kullanıldı. Bunun yanı sıra Dvd tipinde olan gora değişkeni son iki metotta yer almış. Burada kodun tekrarı (duplication) söz konusu. TDD kurallarına göre kod tekrarının yok edilmesi gerekiyor. Yeniden yapılandırma (refactoring) metotlarını kullanarak aynı değişkenin birden fazla yerde kullanılmasını engellememiz gerekiyor. Örneğin tekrar eden bu değişkenleri setUp() metoduna çekerek, merkezi bir yerde bulunmalarını, ama her test öncesinde yeniden oluşturulmalarını sağlayabiliriz. Bu amaçla setUp() metodunu oluşturuyoruz. JUnit her test öncesinde setUp() metodunu çalıştırarak, test için gerekli altyapıyı oluşturur. Bu her test için geçerlidir. Örneğin test sınıfında üç değişik test metodu varsa, bizim örneğimizde olduğu gibi setUp() metodu her test öncesi otomatik olarak devreye girerek test için gerekli değişkenleri oluşturur, yani üç test metodu için setUp() metodu üç defa test öncesi çalıştırılmış olur. Böylece her test ihtiyaç duyduğu değişkenleri kullanabilir.

setUp() metodunu kod 9.15 de yer aldığı gibi yapılandırıyoruz. manager, gora ve arog değişkenleri tüm testler tarafından ortak olarak kullanıldığı için bu değişkenleri sınıf değişkenleri olarak tanımlıyoruz.

Kod 9.15 DvdManagerTest sınıfında setUp() metodunu ekliyoruz

```
private DvdManager manager = null;
```

```
private Dvd gora = null;
private Dvd arog = null;

@Before
public void setUp()
{
    manager = new DvdManager();
    gora = new Dvd();
    arog = new Dvd();
}
```

Görüldüğü gibi DvdManagerTest sınıfında test metotlarını etkileyecek bir değişiklik yapmadık, çünkü test metotlarında kullanılan değişkenler lokaldir ve bu testler hala çalışır durumda. Refactoring yaparken bu konuya dikkat edilmesi gerekmektedir. Yaptığımız her değişikliğin ardından testleri çalıştırarak, yaptığımız değişikliklerin yan etkilerinin olup, olmadığını incelememiz gerekiyor. Yan etkilerin oluşması durumunda testlerin bazıları kırılacaktır. Yaptığımız her değişikliğin ardından testlerin mutlaka çalışır durumda olmalarını sağlamamız gerekiyor.

Bu değişikliğin ardından ilk test metodunda yer alan manager isimli değişkeni uzaklaştırabiliriz.

Kod 9.16 İlk test metodu

```
public void empty_lists_size_should_be_zero()
{
    assertEquals("listenin büyüklügü 0 degildir", 0, manager.size());
}
```

manager değişkeni artık bir sınıf değişkeni olduğu ve setUp() metodunda oluşturulduğu için empty_lists_size_should_be_zero() metodundan bu değişkeni uzaklaştırabiliriz. Bu değişikliğin ardından testi çalıştırarak, kırılmalar olup, olmadığını inceliyoruz. Bu değişikliklerin adım adım yapılması gerekiyor, yani her değişikliğin ardından testlerin tekrar çalıştırılarak, kırılmalar olup, olmadığının incelenmesi lazım, aksi taktirde tüm değişiklikler birden yapılırsa, birden fazla kırılma oluşabilir ve bunların ortadan kaldırılması zaman alıcı olabilir. Ayrıca aynı zamanda bir kırılma yerine birden fazla kırılmayla uğraşmak, odaklandığımız testten uzaklaşmamıza sebep verebilir.

İkinci testide kod 9.17 de olduğu şekilde değiştiriyoruz.

Kod 9.17 İkinci test metodu

```
public void when_adding_first_item_to_the_list_size_should_be_one()
{
    manager.add(gora);
    assertEquals("listenin büyüklügü 1 degildir", 1, manager.size());
}
```

Bu değişikliğin ardından tekrar tüm testleri çalıştırarak, test sonuçlarını inceliyoruz. Bu değişiklik de bir kırılmaya sebep vermediği için üçüncü teste geçiyoruz. Üçüncü testin son hali kod 9.18 de yer almaktadır.

```
Kod 9.18 Üçüncü test metodu

public void when_adding_second_item_list_size_should_be_two()
{
    manager.add(gora);
    manager.add(arog);
    assertEquals("listenin büyüklügü 2 degildir", 2, manager.size());
}
```

Bu değişikliklerin ardından DvdManagerTest sınıfı kod 9.19 da yer alan yapıya kavuşuyor.

```
Kod 9.19 DvdManagerTest sınıfının son hali

package dvd;

public class DvdManagerTest
{

    private DvdManager manager = null;
    private Dvd gora = null;
    private Dvd arog = null;

    @Before
    public void setUp()
    {
        manager = new DvdManager();
        gora = new Dvd();
        arog = new Dvd();
    }

    @Test
    public void empty_lists_size_should_be_zero()
    {
        assertEquals("listenin büyüklügü 0 degildir",
                0, manager.size());
```

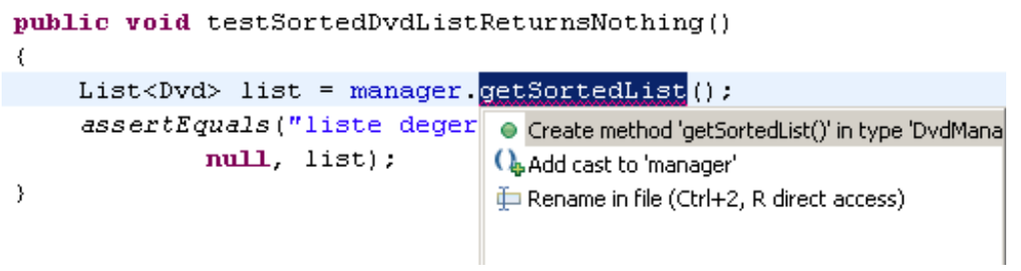
```
    }

    @Test
    public void when_adding_first_item_to_the_list_size_should_be_one()
    {
        manager.add(gora);
        assertEquals("listenin büyüklügü 1 degildir",
                1, manager.size());
    }

    @Test
    public void when_adding_second_item_list_size_should_be_two()
    {
        manager.add(gora);
        manager.add(arog);
        assertEquals("listenin büyüklügü 2 degildir",
                2, manager.size());
    }
}
```

4. Test: BBB ve AAA ismini taşıyan iki film listeye eklendiğinde AAA ismini taşıyan film listede BBB isimli filmden önce yer almalıdır.

Son testi oluşturmak için kolları sıvıyoruz. Bizden alfabetik sıralama yapabilen bir liste oluşturmamız isteniyor. Bu bizi DvdManager sınıfının yapısını değiştirmeye zorlayacak gibi görünüyor. Kod 9.20 de yer aldığı gibi bir test düşünülebilir.

```
Kod 79.20 Dördüncü test metodu

@Test
public void sorted_empty_list_should_return_nothing()
{
    List<Dvd> list = manager.getSortedList();
    assertEquals("liste bos degil", 0, list.size());
}
```

Alfabetik sıraya sahip bir listeye sahip isek, o zaman getSortedList() isminde bir metot ile bu listeyi edinebiliriz diye düşünüyorum ve testi bu şekilde yapılandırıyorum. Eclipse doğal olarak getSortedList() metodunun henüz DvdManager sınıfında bulunmadığını resim 9.11 de görüldüğü gibi ikaz ediyor.

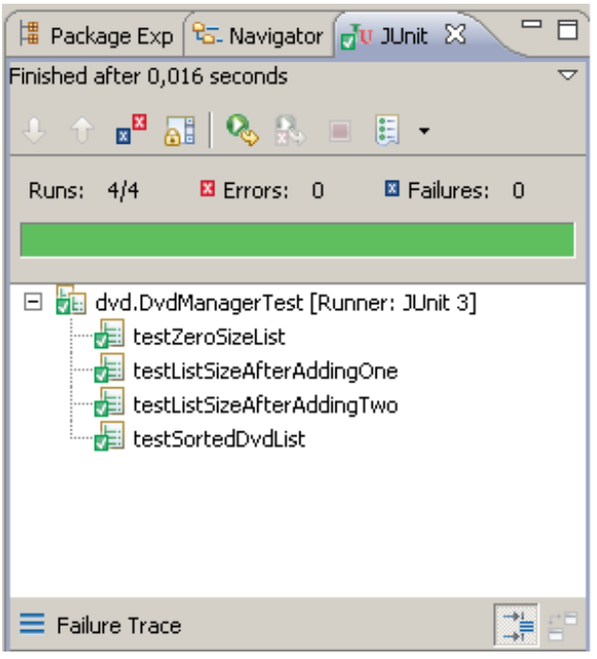


Resim 9.11 Eclipse getSortedList() isimli bir metodun DvdManager sınıfında bulunmadığını tespit etti

getSortedList() metodunu en basit haliyle kod 9.21 de görüldüğü gibi oluşturuyoruz.

```
Kod 9.21 getSortedList()  metodunu ekliyoruz

package dvd;

import java.util.List;

public class DvdManager
{
    private int listSize = 0;

    public int size()
    {
        return listSize;
    }

    public void add(Dvd gora)
    {
        listSize++;
    }

    public List<Dvd> getSortedList()
    {
        return new ArrayList<Dvd>();
    }
}
```

TDD kurallarına göre yeni metotlar oluşturulurken en basit halleriyle implemente edilmeleri gerekiyor. Bu bir listeyi geri veren bir metot için null değeridir ya da boş bir listedir. Bu değişikliğin ardından testleri çalıştırıyoruz ve tüm testlerin olumlu sonuç verdiğini görüyoruz.



Resim 9.12 Tüm testler çalışır durumda

Dördüncü test (Test 4) kapsamında yeni bir test metodu oluşturarak listeye bir film eklendiği zaman nasıl bir sonuç alabileceğimizi inceleyelim. Bunun için bir sonraki test metodunu oluşturuyoruz.

```
Kod 9.22 Yeni test  metodu

@Test
public void sorted_list_with_only_one_element_should_return
                                        _this_element()
{
    manager.add(gora);
    gora.setTitle("Gora");
    List<Dvd> list = manager.getSortedList();
    assertEquals("listedeki ilk film Gora degildir",
                                "Gora", list.get(0).getTitle());
}
```

Alfabetik bir liste oluşturabilmek için bir kıyaslama kriterine ihtiyacımız var. Örneğin alfabetik listeyi filmin ismine göre oluşturabiliriz. Kıyaslamayı yapabilmek için listeye bir filmi eklerken filmin ismini de kaydedebilmemiz gerekmektedir. Bu amaçla Dvd sınıfında setTitle() isminde bir metot düşünüyoruz. Kod 9.22 de yer alan testte Gora ismini taşıyan bir filmi listeye ekliyoruz. setTitle() metodunu kullanarak, filmin ismini tanımlıyoruz. Bu değişikliğin ardından Dvd sınıfı kod 9.23 yer aldığı şekilde olacaktır.

```
Kod 9.23 Dvd sınıfına setTitle() ve getTitle()  metotları ekleniyor
```

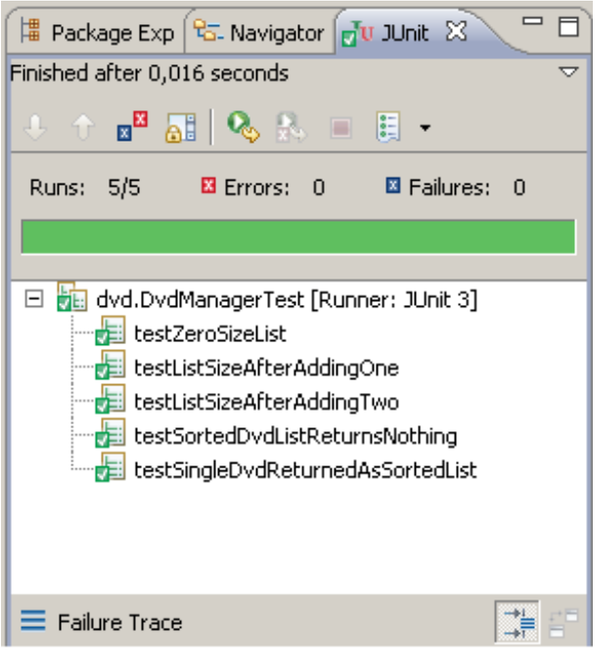
```
package dvd;

public class Dvd
{
    private String title;

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String pTitle)
    {
        this.title = pTitle;
    }
}
```

Bu yeni testi çalıştırdığımız taktirde IndexOutOfBoundsException hatası oluşacaktır, çünkü getSortedList() boş bir listeyi geri vermektedir, ama bizim beklentimiz listenin sıfırıncı alanında Gora isimli filmi bulmaktır. Bu sorunu kod 9.24 de yer aldığı gibi ortadan kaldırıyoruz.

```
Kod 9.24 DvdManager sınıfına nihayet gerçek bir liste ekleniyor

package dvd;

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

public class DvdManager
{
    private int listSize = 0;

    private List<Dvd> list = new ArrayList<Dvd>();

    public int size()
    {
        return listSize;
    }

    public void add(Dvd gora)
    {
        listSize++;
        list.add(gora);
    }
```

```
    public List<Dvd> getSortedList()
    {
        return list;
    }
}
```

Sorunu çözmek için gerçek listeyi sınıf değişkeni olarak tanımlıyor ve getSortedList() metodunun bu listeyi geriye vermesini sağlıyoruz. Filmlerin listeye eklenmesi için kullanılan add() metodunu değiştiriyor ve parametre olarak gelen filmi list.add() metodu ile listeye eklenmesini sağlıyoruz. Şu andan itibaren DvdManager sınıfı gerçek bir listeye kavuştu. Tüm testleri çalıştırarak, mevcut tüm testlerin olumlu cevap verdiğini kontrol ediyoruz.



Resim 9.13 Tüm testler çalışır durumda

Testler çalışır durumda olduğuna göre kod içinde tekrar (code duplication) olup, olmadığına bir göz atalım. DvdManager.add() metodunda listSize++ şeklinde bir işlem var. Bu ilk testlerde kullandığımız bir değişkendi. DvdManager sınıfına gerçek bir liste eklendikten sonra bu değişkeni ortadan kaldırabiliriz, çünkü aynı veriyi list.size() üzerinden de edinmek mümkündür. Aynı zamanda DvdManager.size() metodunu da listenin büyüklüğünü geri verecek şekilde değiştirdiğimiz taktirde hem kod kod tekrarını ortadan kaldırmış olacağız hem de ilk testlerin bu değişiklikler sonucunda kırılmasını engelleyeceğiz. Bu değişikliklerin ardından DvdManager sınıfı kod 9.25 deki yapıya sahip olacaktır.

```
Kod 9.25 DvdManager sınıfından gereksiz değişkenleri uzaklaştırıyoruz

package dvd;
```

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

public class DvdManager
{
    private List<Dvd> list = new ArrayList<Dvd>();

    public imt size()
    {
        return list.size();
    }

    public void add(Dvd gora)
    {
        list.add(gora);
    }

    public List<Dvd> getSortedList()
    {
        return list;
    }
}
```

Bu değişikliğin ardından tekrar tüm testleri çalıştırıp, hepsinin çalışır durumda olduğunu kontrol etmemiz gerekiyor.

Listemizin bir film ile çalışır durumda olduğunu en son testimizde kanıtlamış olduk. Şimdi iki filmin eklendiği ve alfabetik bir listenin oluştuğu durumu test edelim.

Kod 9.26 Alfabetik sıranın test edildiği test metodu

```
@Test
public void list_items_should_be_returned_as_sorted_list()
{
    manager.add(gora);
    gora.setTitle("Gora");

    manager.add(arog);
    arog.setTitle("Arog");

    List<Dvd> list = manager.getSortedList();
    assertEquals("listedeki ilk film Arog degildir",
            "Arog", list.get(0).getTitle());

    assertEquals("listedeki ikinci film Gora degildir",
```

```
            "Gora", list.get(1).getTitle());
}
```

Bu test çalıştırıldığı taktirde, aşağıdaki şekilde bir hata mesajı verecektir:

junit.framework.ComparisonFailure: listedeki ilk film Arog degildir expected: but was:

Test metodunda listeye önce Gora daha sonra Arog isimlerini taşıyan iki film ekliyoruz. Liste alfabetik olmak zorunda olduğu için ilk beklentimizi de ona göre oluşturuyoruz: listenin ilk filmi Arog ismini taşımak zorunda. JUnit in verdiği hata mesajında beklentinin Arog isimli bir film olduğu, ama elde edilen film isminin Gora olduğu yer almaktadır. Durumu şöyle açıklayabiliriz: listeye eklediğimiz filmler alfabetik sıraya göre değil, listeye ekleniş sırasına göre dizilmiştir. Bu sorunu gidermek için getSortedList() metodunu şu şekilde değiştirebiliriz:

```
Kod 9.27 getSortedList()  metodu listeyi alfabetik hale getiriyor

public List<Dvd> getSortedList()
{
    Collections.sort(list);
    return list;
}
```

Kullandığımız liste Dvd tipinde nesnelere göre ayarlanmıştır (List). Bu durumda Collectios.sort() metodu bizim listemizi bu halı ile alfabetik hale getiremez, çünkü liste içinde yer alan Dvd nesneleri kıyaslayabileceği bir mekanizmaya sahip değil. Dvd nesnelerini birbirleri ile kıyaslamak için Comparable interface sınıfını kullanabiliriz. Comparable interface sınıfını implemente eden bir sınıf compareTo() isminde bir metot aracılığıyla başka sınıflar tarafından bu sınıfın sahip olduğu nesneler üzerinde kıyaslama yapılmasını mümkün kılmaktadır.

Kod 9.27 da yer alan metot derlenmez durumda, çünkü sort metodu listeyi bu hali ile alfabetik hale sokamamaktadır. Tek çözüm biraz öncede bahsettiğim gibi Dvd sınıfının Comparable interface sınıfını implemente etmesi. Şimdi doğal olarak hemen Dvd sınıfına gidip gerekli değişikleri yapmamız gerekiyor değil mi? Hayır, kesinlikle! Eğer bunu yaparsak, testi olmayan kod yazmış oluruz ve bu kesinlikle TDD kurallarına karşıdır. Önce test, daha sonra gerekli kod!

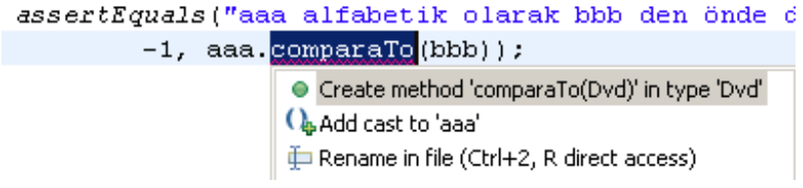
Bu noktada DvdManagerTest test sınıfını ve üzerinde çalıştığımız en son test metodunu yarıda bırakarak, yeni bir test sınıfı oluşturuyoruz. Bu yeni test sınıfı

iki Dvd nesnesinin nasıl kıyaslanabileceğini test edecek. DvdTest isminde yeni bir test sınıfı oluşturuyoruz.

Kod 9.28 DvdTest sınıfı

```
package dvd;

public class DvdTest
{
    @Before
    public void aaa_comes_before_bbb()
    {
        Dvd aaa = new Dvd();
        aaa.setTitle("aaa");

        Dvd bbb = new Dvd();
        bbb.setTitle("bbb");

        assertEquals("aaa'nin alfabetik olarak bbb'den önde
                        olmasi gerekiyor",
                -1, aaa.compareTo(bbb));
    }

}
```

İki Dvd film oluşturarak, assertEquals komutu ile ilk filmin ikinci filmden alfabetik olarak önde olması gerektiğini test ediyoruz. Dvd sınıfında comparaTo() isminde bir metot olmadığı için Eclipse gerekli uyarıyı yapıyor:



Resim 9.14 Dvd sınıfında compareTo metodu bulunmuyor

compareTo() metodunu Dvd sınıfında oluşturuyoruz. Comparable interface sınıfını implemente eden her sınıfın compareTo() isminde bir metodunun bulunması gerekiyor. Kıyaslama compareTo() metodunda gerçekleşiyor. Bu değişikliklerin ardından Dvd sınıfı kod 9.29 daki halini alıyor.

Kod 9.29 Yeni Dvd sınıfı

```
package dvd;

public class Dvd implements Comparable<Dvd>
```

```
{
    private String title;

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String pTitle)
    {
        this.title = pTitle;
    }

    public int compareTo(Dvd bbb)
    {
        return this.getTitle().compareTo(bbb.getTitle());
    }
}
```

Dvd sınıfı Comparable interface sınıfını implemente ederek kendinden olan nesnelerin kıyaslanmalarını mümkün kılıyor. compareTo() metodunda mevcut filmin ismi parametre olarak verilen filmin ismi ile kıyaslanıyor. Eğer değer −1 ise, mevcut film parametre olarak gelen filmden alfabetik olarak öndedir. 0 durumunda iki filmin ismi de aynıdır. Eğer değer 1 ise mevcut film parametre olarak gelen filmden alfabetik olarak sonra gelmektedir.

Test içinde yer alan beklentimizde de bunu ifade ettik. aaa ismi bbb isminden önce geldiği için beklentimiz −1 değeridir, yani aaa nın bbb den önce geldiğidir.

İkinci bir test ile yeni implemente ettiğimiz kıyaslama işlemini doğruluğunu teyit edelim.

```
Kod 9.30 İkinci test metodu

@Test
public void aaa_should_equal_to_aaa()
{
    Dvd aaa = new Dvd();
    aaa.setTitle("aaa");

    assertEquals("aaa alfabetik olarak aaa ile ayni siradadir",
                0, aaa.compareTo(aaa));
}
```

Aynı ismi taşıyan iki film kıyaslandığında elde edeceğimiz değer sıfırdır, yani iki

filmde alfabetik olarak aynı sıradadır. Bu testte çalışır durumda. Son bir testle kıyaslama işleminin doğru yapıldığını kontrol edelim.

Kod 9.31 Üçüncü test metodu

```
public void bbb_comes_after_aaa()
{
    Dvd aaa = new Dvd();
    aaa.setTitle("aaa");

    Dvd bbb = new Dvd();
    bbb.setTitle("bbb");

    assertEquals("bbb'nin alfabetik olarak aaa'dan sonra " +
            "gelmesi gerekiyor",
            1, bbb.compareTo(aaa));
}
```

İlk testte olduğu gibi aaa ve bbb isimlerini taşıyan iki film oluşturduktan sonra, bbb isimli filmin alfabetik olarak aaa isimli filmden sonra geldiğini test ediyoruz. İmplementasyon doğru olduğu için bu testte olumlu sonuç veriyor ve tekrar yarıda bıraktığımız DvdManagerTest test sınıfına geri dönüyoruz. En son üzerinde çalıştığımız test metodu kod 9.26 da yer almaktadır. Eğer bu testi en son yaptığımız değişikliklerden sonra çalıştırırsak olumlu sonuç verdiğini görürüz, çünkü filmler arası kıyaslama doğru bir şekilde implemente edildiği için testte yer alan beklentilerimiz gerçekleşmektedir.

Böylece ilk gereksinim için oluşturduğumuz 4 testi ve gerekli sınıfları TDD tarzı implemente etmiş olduk. Son olarak oluşturduğumuz tüm sınıfları tekrar bir gözden geçirelim.

Kod 9.32 DvdManagerTest sınıfı

```
package dvd;

import java.util.List;

public class DvdManagerTest
{

    private DvdManager manager = null;
    private Dvd gora = null;
    private Dvd arog = null;

    @Before
```

```
    public void setUp()
    {
        manager = new DvdManager();
        gora = new Dvd();
        arog = new Dvd();
    }

    @Test
    public void empty_lists_size_should_be_zero()
    {
        assertEquals("listenin büyüklügü 0 degildir",
                0, manager.size());
    }


    public void when_adding_first_item_to_the_list_size_
                            should_be_one()
    {
        manager.add(gora);
        assertEquals("listenin büyüklügü 1 degildir",
                1, manager.size());
    }


    public void when_adding_second_item_list_size_should_
                        be_two()
    {
        manager.add(gora);
        manager.add(arog);
        assertEquals("listenin büyüklügü 2 degildir",
                2, manager.size());
    }


    public void sorted_empty_list_should_return_nothing()
    {
        List<Dvd> list = manager.getSortedList();
        assertEquals("liste bos degil",
                0, list.size());
    }


    public void sorted_list_with_only_one_element_should_
                            return_this_element()
    {
        manager.add(gora);
        gora.setTitle("Gora");
        List<Dvd> list = manager.getSortedList();
        assertEquals("listedeki ilk film Gora degildir",
```

```
                "Gora", list.get(0).getTitle());
    }


    public void list_items_should_be_returned_as_sorted_list()
    {
        manager.add(gora);
        gora.setTitle("Gora");

        manager.add(arog);
        arog.setTitle("Arog");

        List<Dvd> list = manager.getSortedList();
        assertEquals("listedeki ilk film Arog degildir",
                "Arog", list.get(0).getTitle());

        assertEquals("listedeki ikinci film Gora degildir",
                "Gora", list.get(1).getTitle());
    }
}
```

Kod 9.33 DvdTest sınıfı

```
package dvd;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class DvdTest {

    @Test
    public void aaa_comes_before_bbb() {
        Dvd aaa = new Dvd();
        aaa.setTitle("aaa");

        Dvd bbb = new Dvd();
        bbb.setTitle("bbb");

        assertEquals("aaa'nin alfabetik olarak bbb'den önde
                olmasi gerekiyor",
                            -1, aaa.compareTo(bbb));
    }

    @Test
    public void aaa_should_equal_to_aaa() {
        Dvd aaa = new Dvd();
        aaa.setTitle("aaa");
```

```
        assertEquals("aaa alfabetik olarak aaa ile ayni siradadir",
            0, aaa.compareTo(aaa));
    }

    @Test
    public void bbb_comes_after_aaa() {
        Dvd aaa = new Dvd();
        aaa.setTitle("aaa");

        Dvd bbb = new Dvd();
        bbb.setTitle("bbb");

        assertEquals("bbb'nin alfabetik olarak aaa'dan sonra " +
                    "gelmesi gerekiyor", 1, bbb.compareTo(aaa));
    }

}
```

Kod 9.34 Dvd sınıfı

```
package dvd;

public class Dvd implements Comparable<Dvd> {
    private String title;

    public String getTitle() {
        return title;
    }

    public void setTitle(String pTitle) {
        this.title = pTitle;
    }

    public int compareTo(Dvd bbb) {
        return this.getTitle().compareTo(bbb.getTitle());
    }
}
```

Kod 9.35 DvdManager sınıfı

```
package dvd;

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

public class DvdManager {
```

```
    private List<Dvd> list = new ArrayList<Dvd>();

    public int size() {
        return list.size();
    }

    public void add(Dvd gora) {
        list.add(gora);
    }

    public List<Dvd> getSortedList() {
        Collections.sort(list);
        return list;
    }
}
```

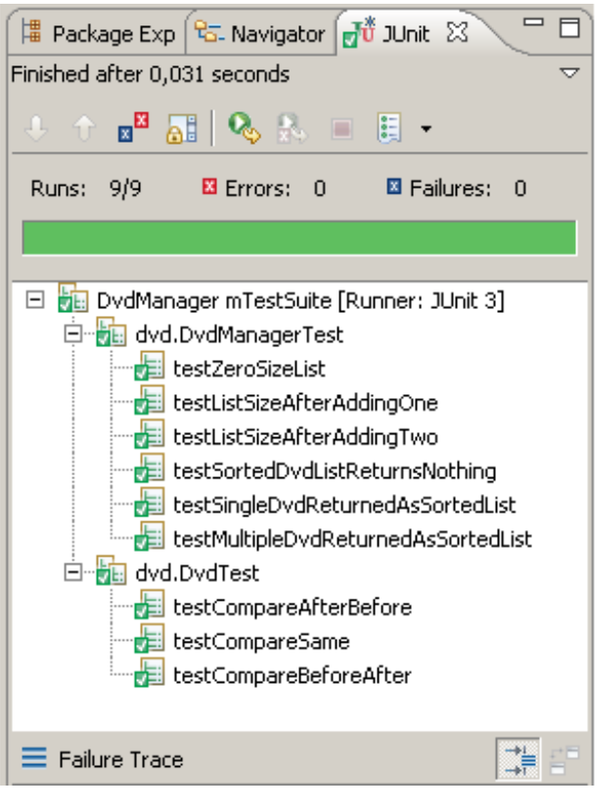
Test Kapsama Alanı (Test Coverage)

Test kapsama alanı konusunu 8. bölümde ele almıştık. Hazırladığımız testlerin kodun yüzde kaçını işletir hale getirdiğini ölçmek için Eclipse Plugin olan EclEmma programından yararlanabiliriz.

Test kapsama alanını ölçebilmek için tüm testlerin bir araya getirildiği ve aynı anda çalıştırılabildiği bir TestSuite sınıfının oluşturulması gerekiyor. Kod 9.36 da iki test sınıfının yer aldığı AllTests sınıfı yer almaktadır.

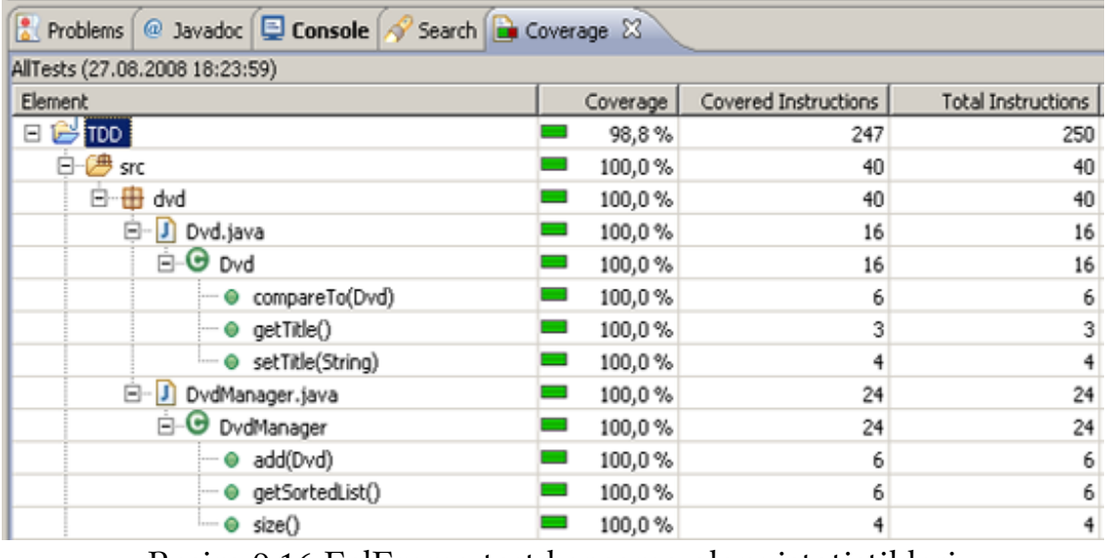
```
Kod 7.36 DvdManager TestSuite
package dvd;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ DvdManagerTest.class, DvdTest.class })
public class AllTests {
}
```

AllTests sınıfında mevcut test sınıflarını bir araya getirerek, aynı anda tüm testlerin çalıştırılmasını sağlayabiliriz. Kodun yüzde kaçının test kapsama alanına girdiğini ölçebilmek için mevcut tüm testlerin aynı anda çalıştırılmaları önemlidir.



Resim 9.15 DvdManager TestSuite

EclEmma Plugin üzerinden AllTests sınıfını çalıştırdığımızda, resim 9.16 da görülen tablo karşımıza çıkacaktır.



Element	Coverage	Covered Instructions	Total Instructions
TDD	98,8 %	247	250
src	100,0 %	40	40
dvd	100,0 %	40	40
Dvd.java	100,0 %	16	16
Dvd	100,0 %	16	16
compareTo(Dvd)	100,0 %	6	6
getTitle()	100,0 %	3	3
setTitle(String)	100,0 %	4	4
DvdManager.java	100,0 %	24	24
DvdManager	100,0 %	24	24
add(Dvd)	100,0 %	6	6
getSortedList()	100,0 %	6	6
size()	100,0 %	4	4

Resim 9.16 EclEmma test kapsama alanı istatistikleri

EclEmma istatistiklerinde görüldüğü gibi kodun (DvdManager ve Dvd sınıfları) testler aracılığıyla %100 kapsandığını görüyoruz. TDD harici yapılan implementasyonlarda %100 test alanı kapsamı sağlamak hemen hemen imkansız gibidir. Çoğu zaman implementasyon bittikten sonra hazırlanan testler ile %30 yada %40 oranında kapsama alanı yakalanabilmektedir ve bu

yeterli değildir. %100 ü yakalamak varken, neden %30, %40 larla yetinelim? Umarım şimdi TDD nin avantajların görmüş ve anlayabilmişsinizdir.

10. Bölüm

XP ile Shop Sistemi İmplementasyonu

Giriş

Bu bölümde tanıştığımız Extreme Programming tekniklerini kullanarak shop sistemini implemente etmeye başlayacağız. Bütün bir shop sisteminin nasıl implemente edildiğini ne yazık ki bu kitaba sığdırmam mümkün değil. Bu yüzden projeyi oluştururken XP nin en önemli bölümlerini tematize edebileceğim bölümler üzerinde yoğunlaşacağım. Bu bölümde

- Eclipse ile bir projenin nasıl oluşturulduğunu,
- Üç katmanlı mimarinin ne olduğunu,
- Top-down TDD nin nasıl uygulandığını,
- Onay/Kabul testlerinin nasıl oluşturulduğunu,
- Selenium ile onay/kabul testlerinin nasıl implemente edildiğini,
- Login modülü için gerekli tasarımın nasıl oluşturulduğunu,
- Mock sınıflar yardımıyla gösterim, işletme ve veri katmanının nasıl implemente edildiğini,
- DBUnit kullanılarak veri katmanının nasıl implemente edildiğini,
- Sürekli entegrasyon için gerekli olan JUnit-Ant entegrasyonunu nasıl yapıldığını

yakından inceleyeceğiz.

Her Şeyin Başı Eclipse

Bir Eclipse projesi oluşturarak projemize başlayabiliriz. Çıkış noktamız beşinci bölümde oluşturduğumuz tablo 5.1 de yer alan kullanıcı hikayeleridir. Müşteri isteklerini kullanıcı hikayesi olarak oluşturdu ve bizler de programcı olarak tatminlerde bulunduk. Bu aşamada keşif ve planlama safhalarını arkamızda bırakmış oluyoruz. Sürüm planımız tablo 10.1 de yer aldığı gibi hazırlamıştır.

Tablo 10.1: Sürüm ve iterasyon planı

İterasyon	Yapılacak olanlar	Bitiş tarihi
0	Programcı ekip için ortam kurulumu (Eclipse, ant, Hibernate, JDK setup)	03.10.2014
1	İlk iterasyon. 1, 2, 4 ve 6 nolu kullanıcı hikayeleri implemente edilecek	10.10.2014
2	9, 10, 3 ve 8 nolu kullanıcı hikayeleri implemente edilecek	17.10.2014
3	5 ve 7 nolu kullanıcı hikayeleri implemente edilecek	24.10.2014

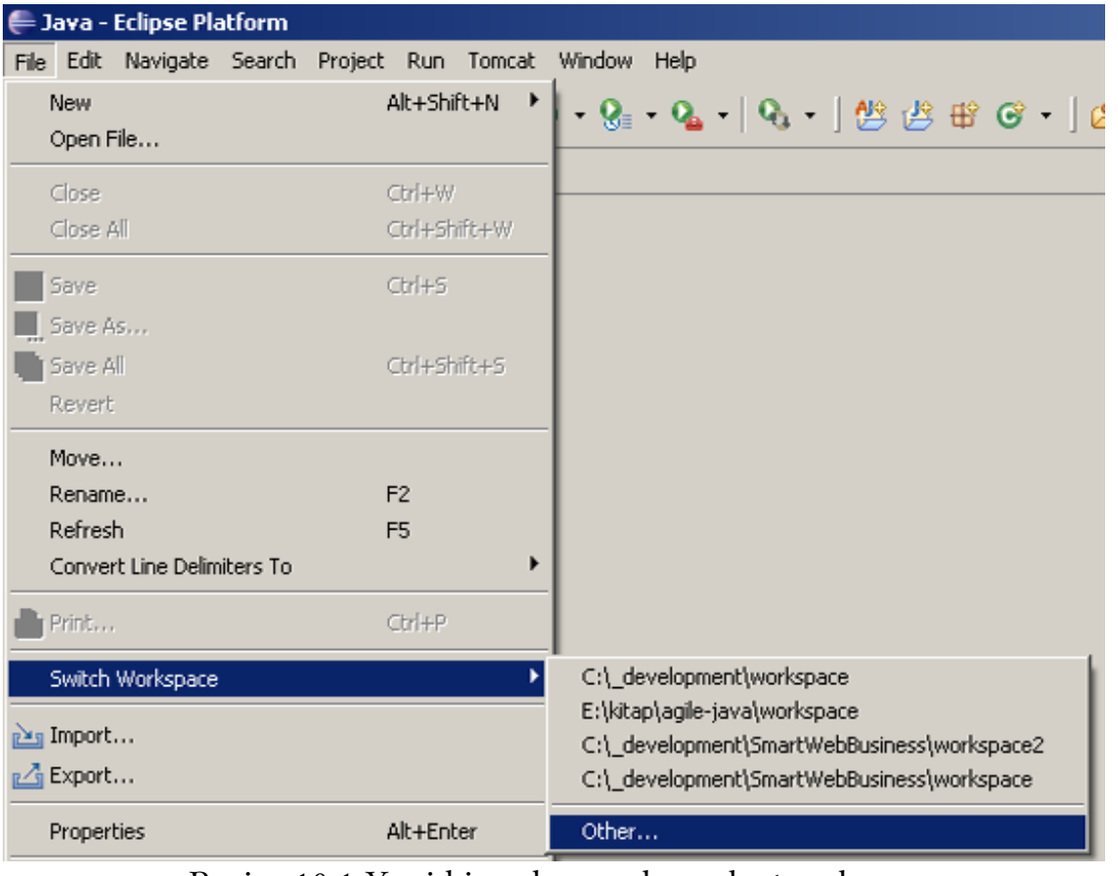
Çalışma ortamımızı kitabın altıncı bölümde olduğu gibi oluşturduk. Yani sıfırıncı iterasyon bitti ve biz birinci iterasyon için hazırız. Birinci iterasyonda tablo 10.2 yer alan kullanıcı hikayelerini implemente edeceğiz.

Tablo 10.2: Kullanıcı hikayelerini, öncelik sıralarını ve implementasyon zamanını ihtiva eden liste

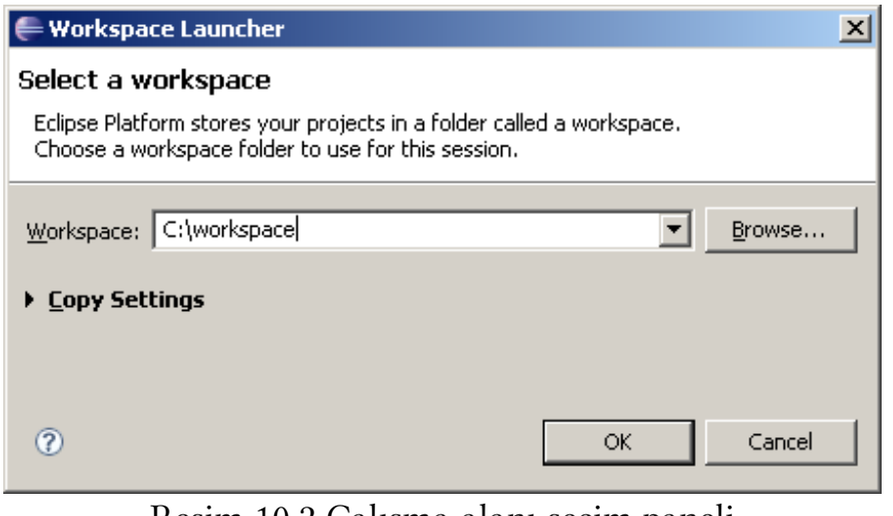
#	Hikaye İsmi	Açıklama	Öncelik Sırası	Tahmin
1	Kayıt	Kullanıcı kişisel bilgilerini kullanarak sisteme kayıt olur	5	1
2	Login	Kullanıcı email adresini ve şifreni kullanarak sisteme login yapar.	5	1
4	Kategori seçimi	Kullanıcı bir kategori seçerek, o kategoride yer alan kitapları edinir.	5	2
6	Sipariş	Kullanıcı sanal kasaya geçerek sipariş verir.	5	1

Yaptığımız tahminlere göre implementasyon için beş güne ihtiyacımız var. Bu zaten seçtiğimiz iterasyon süresi olan bir hafta ile örtüşüyor. Beş iş gününde (günde 8 saat) tablo 10.2 yer alan tüm kullanıcı hikayelerini implemente etmemiz gerekiyor. Şayet bazı nedenlerden dolayı bunu başaramazsak, implemente edemediğimiz kullanıcı hikayelerini bir sonraki iterasyona devredeceğiz.

İterasyon planımız hazır olduğuna göre kolları sıvayabiliriz. Önce Eclipse altında kendimize yeni bir çalışma alanı (Workspace) oluşturuyoruz.



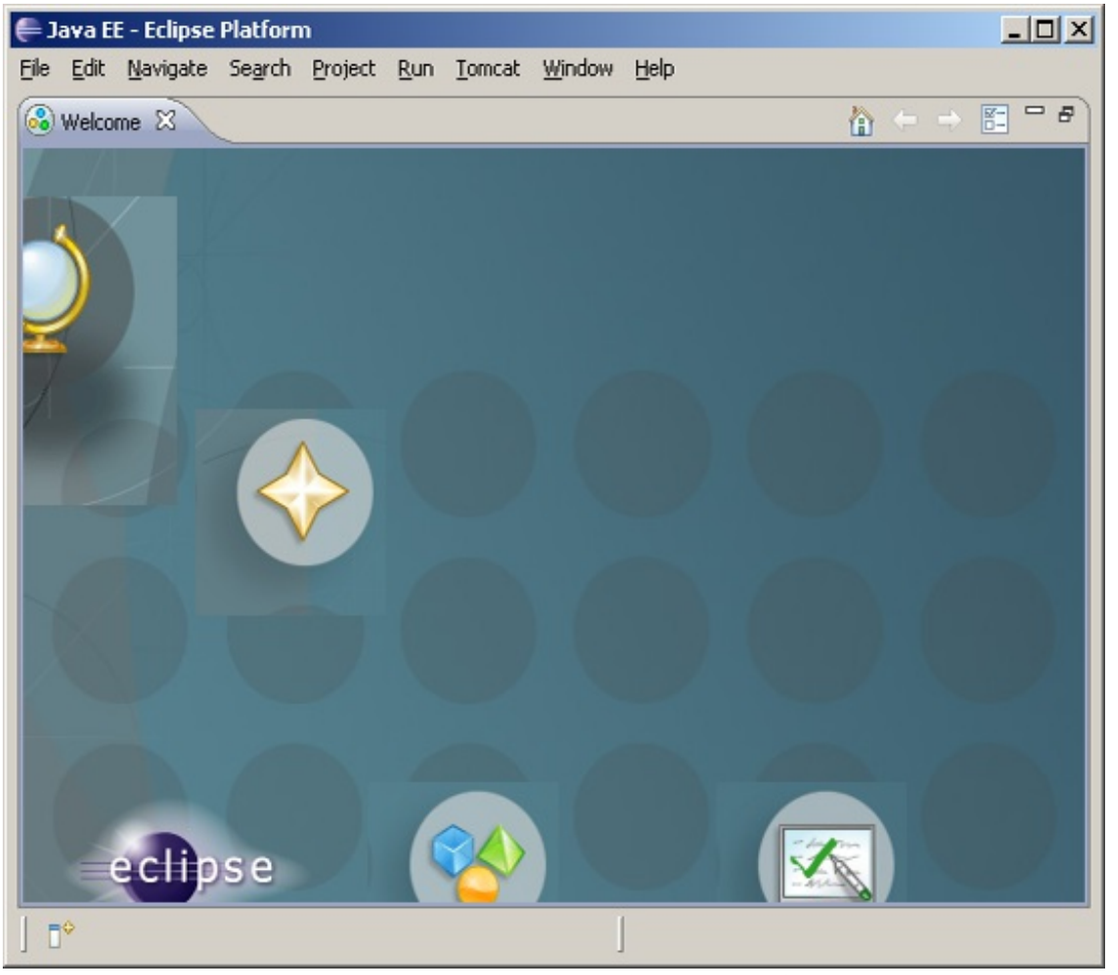
Resim 10.1 Yeni bir çalışma alanı oluşturulması



Resim 10.2 Çalışma alanı seçim paneli

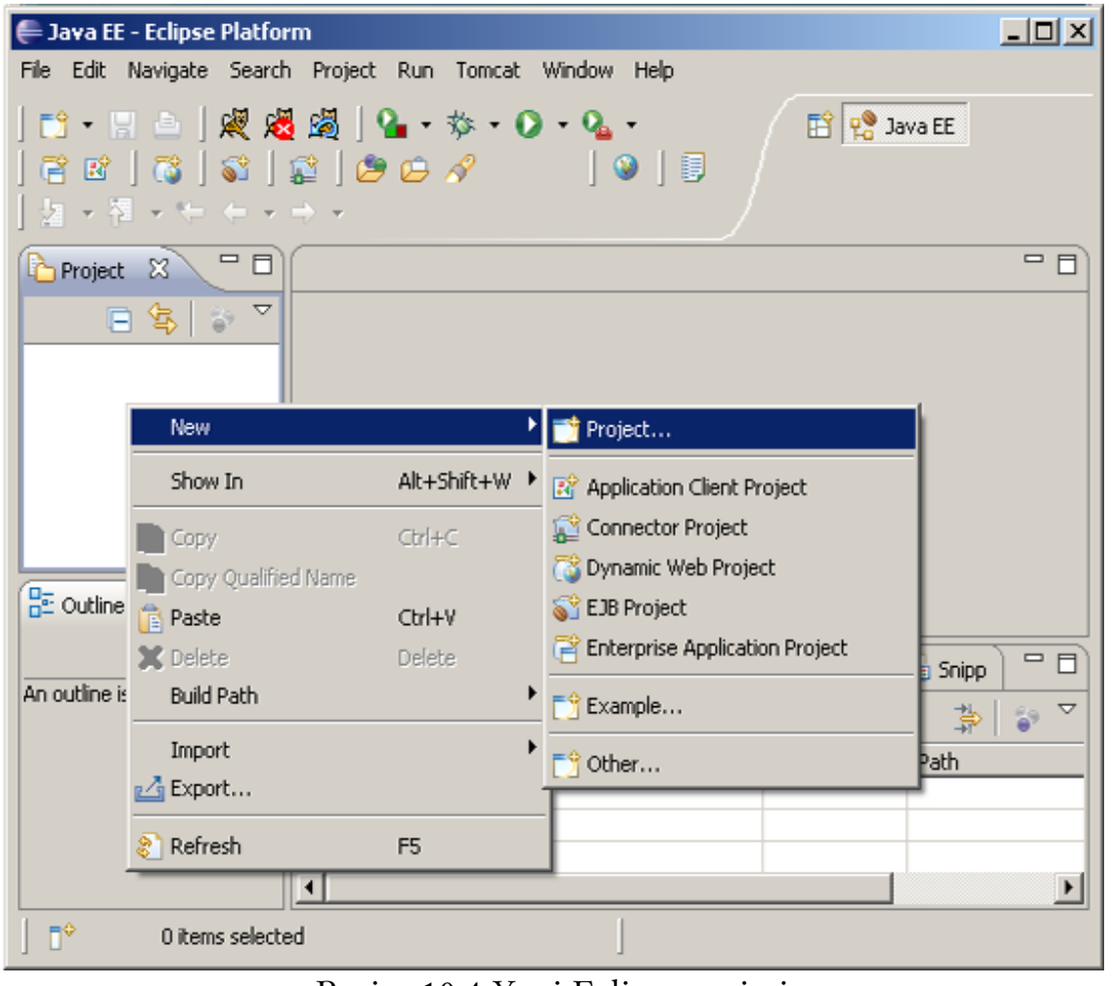
c:\workspace dizininde yeni bir çalışma alanını oluşturuyoruz. Eclipse altında oluşturduğumuz tüm projeler bu dizin içinde yer alacaktır.

Bu işlemin ardından Eclipse bizi resim 10.3 deki haliyle karşılar.



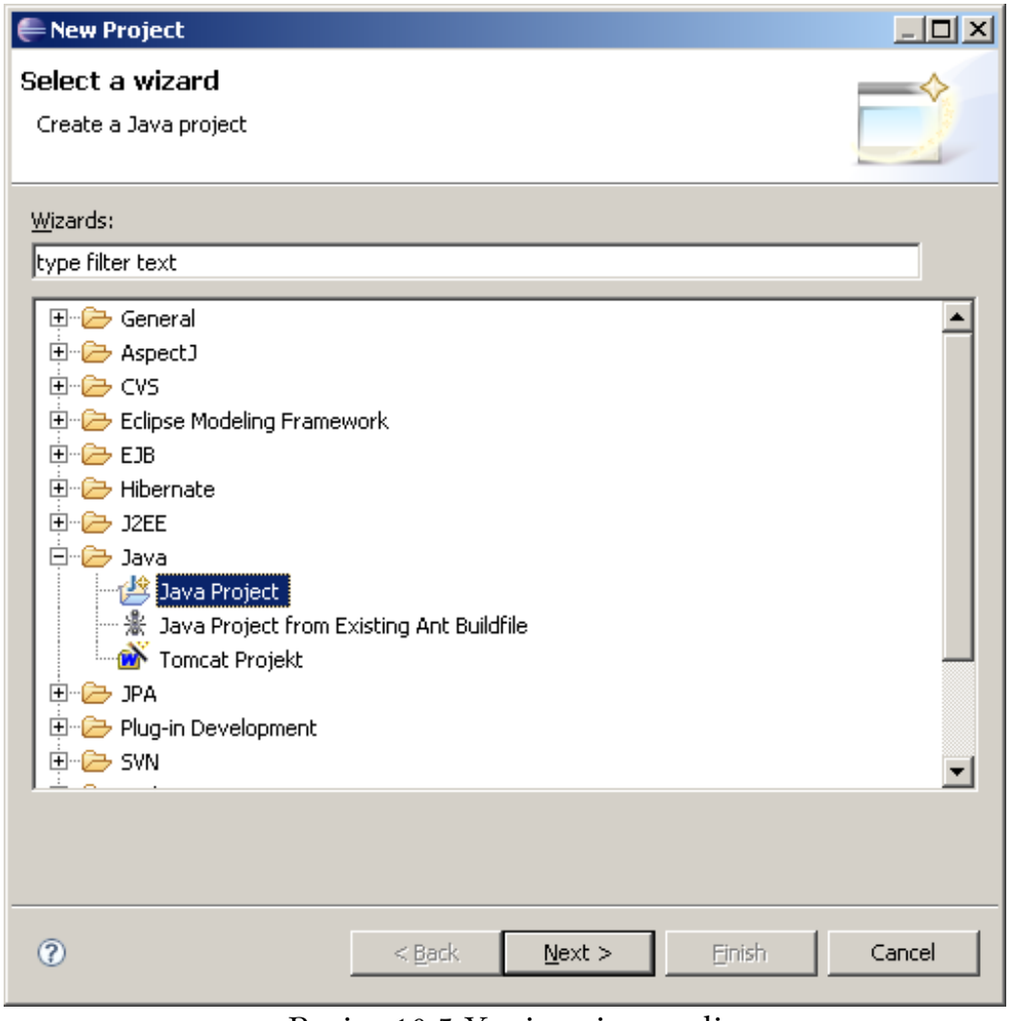
Resim 10.3 Eclipse Welcome sayfası

Yeni bir proje oluşturmak için Welcome sayfasını kapatıyoruz.



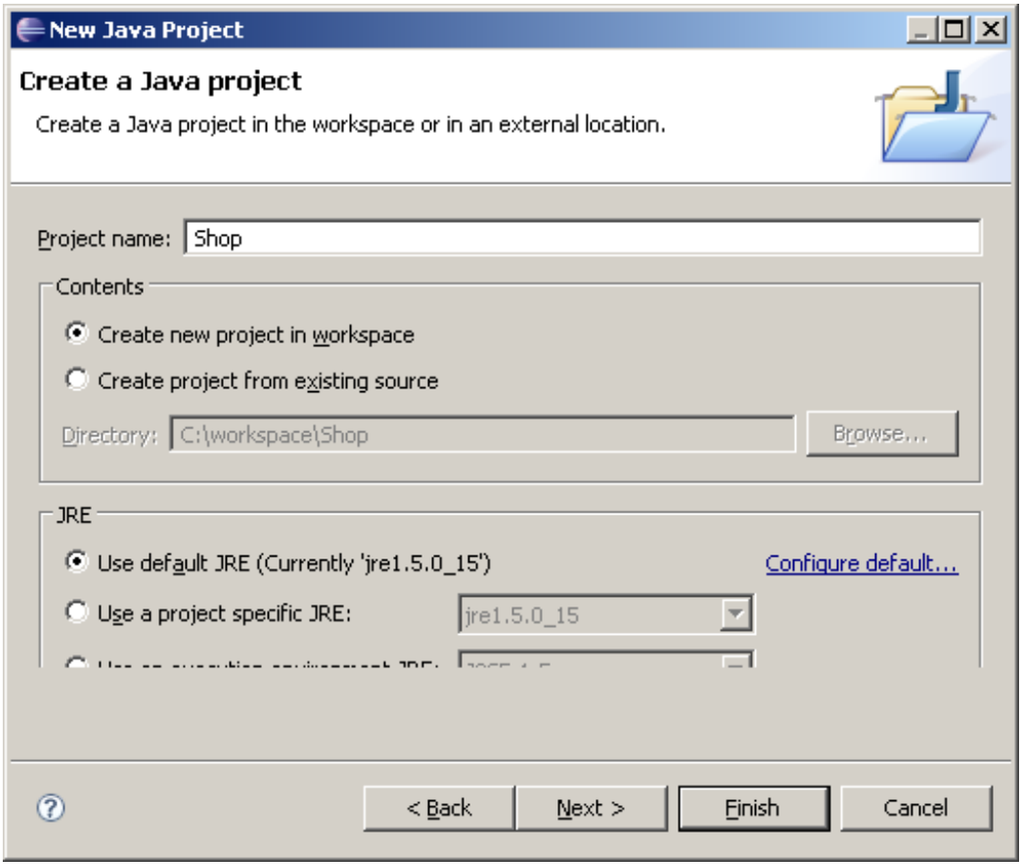
Resim 10.4 Yeni Eclipse projesi

Sol bölümde yer alan Projects paneli içinde sağ tuşa tıkladıktan sonra resim 10.4 yer alan tablo ile karşılaşırız. New > Project... menüsü üzerinde yeni bir Eclipse projesi oluşturma paneline erişiriz.



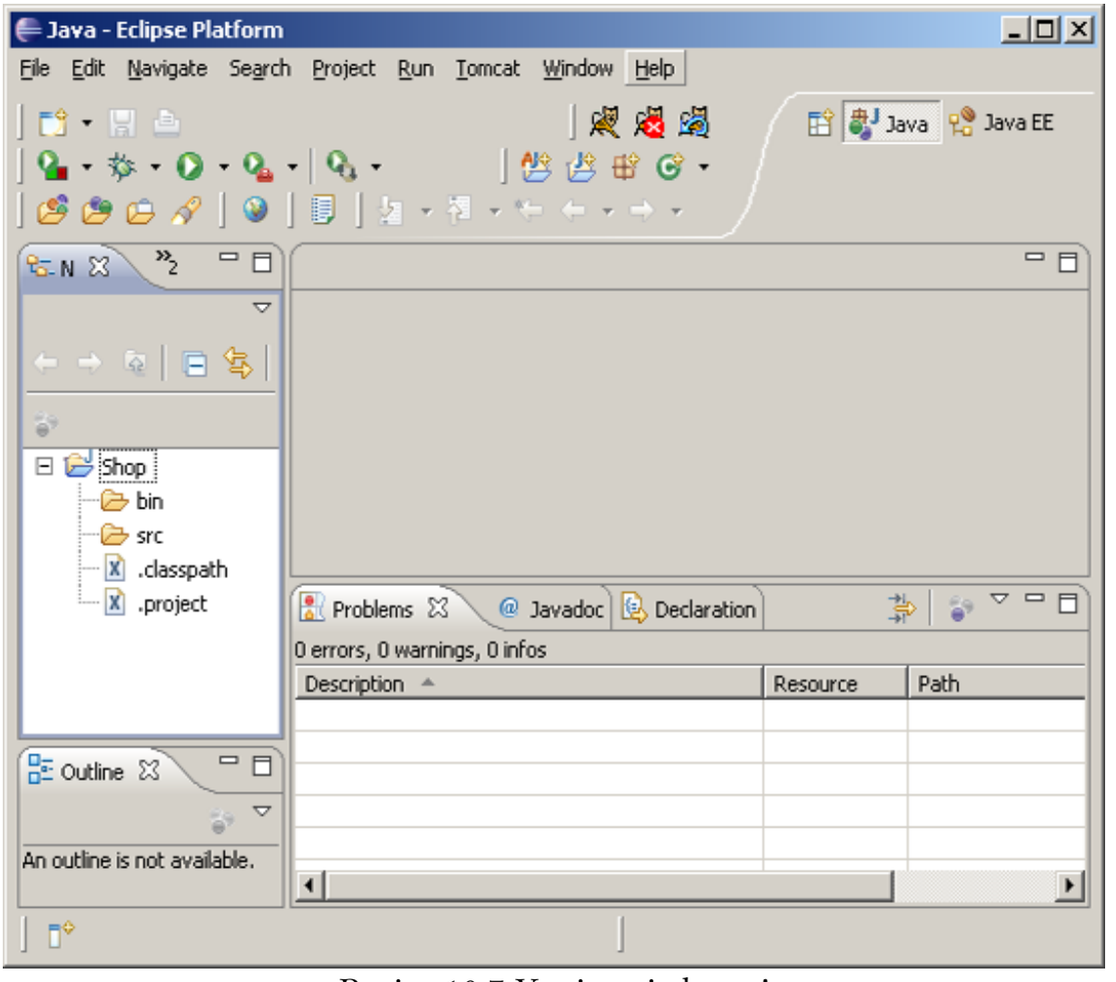
Resim 10.5 Yeni proje paneli

Web tabanlı bir Java projesi oluşturacağımız için Java > Java Project opsiyonunu seçiyoruz.



Resim 10.6 Yeni proje ayar paneli

Bu işlemlerin ardından Shop isminde yeni bir proje oluşturma işlemi tamamlandı.



Resim 10.7 Yeni proje hazır!

Şimdi ne yapıyoruz? Bir düşünelim! İterasyon planımız hazır. O halde neden ilk iterasyon listemizden bir kullanıcı hikayesi seçerek işe başlamıyoruz? Ben iki numaralı kullanıcı hikayesinin implementasyonu ile başlamak istiyorum. Listeye baktığımızda tüm kullanıcı hikayelerinin aynı öncelik derecesine (5) sahip olduğunu görüyoruz. O yüzden herhangi birisini seçerek, implementasyona başlayabiliriz.

TDD Top-Down

Shop sisteminin implementasyonunu dokuzuncu bölümde tanıştığımız TDD tarzı yapacağız. TDD kurallarına göre önce test hazırlayarak yazılıma başlamamız gerekiyor. Bunun için kullanıcı hikayeleri baz alınır. İmplemente etmek istediğimiz kullanıcı hikayesi şöyle:

Kullanıcı hikayesi 2: Kullanıcı email adresi ve şifreni kullanarak sisteme login yapar.

Günümüz projeleri zengin arayüzlerine sahip programlar doğurmaktadır. Bunlar çoğunlukla Shop projemizde olduğu gibi HTML cinsinden web arayüzleridir. Bunun yanı sıra web tabanlı programların çoğunluğu veri tabanında kullanıcı verilerini depolamaktadır. Yani görülen odur ki teknik mimari açıdan bakıldığında böyle projelerin implementasyonu kolay bir iş değildir. Ne yazık ki bu tür programların kitabın dokuzuncu bölümünde bulunan TDD metotları ile oluşturulması kolay değildir.

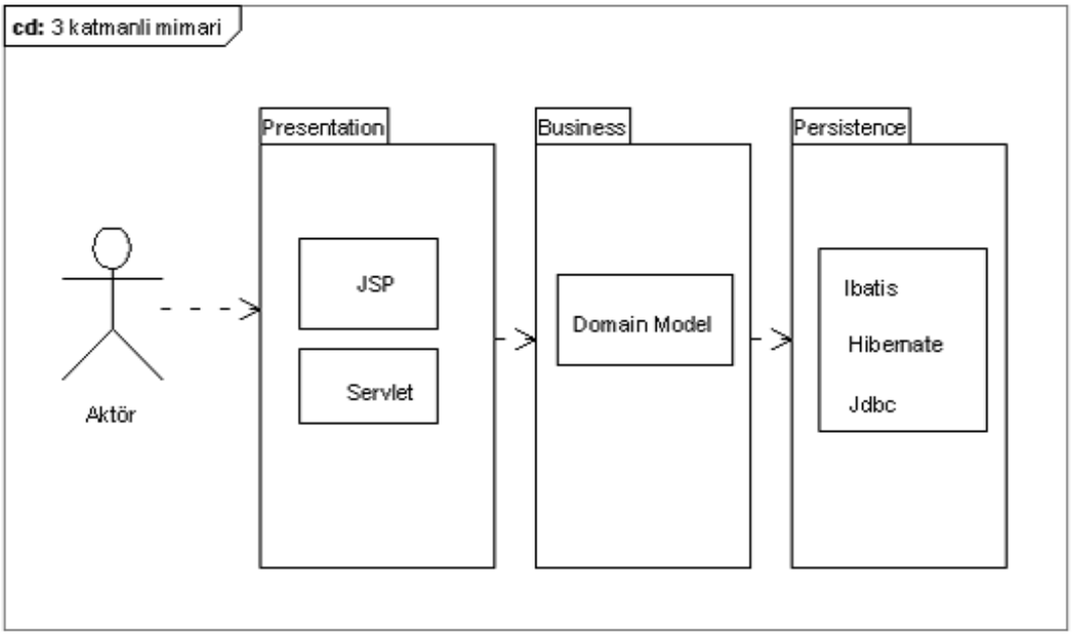
Bu tür karmaşık projeler için değişik türde testlerin hazırlanması gerekmektedir. Sadece komponentlerin (sınıfların) test edildiği birim testleri yeterli değildir. Birim testleri yanı sıra sekizinci bölümde tanıştığımız entegrasyon ve onay/kabul testlerinin de yapılması gerekmektedir.

Dokuzuncu bölümde tanışmış olduğunuz TDD programın temelini oluşturan sınıf ve metotlardan yola çıkmıştı. Hazırladığımız testler önce gerekli sınıfların ve daha sonra sahip oldukları metotların oluşturulmasını sağlamıştı. Bu şekilde programın aşağıdan (temelden) yukarıya doğru oluşturulması için bottom-up (aşağıdan yukarıya doğru) terimi kullanılmaktadır. Dokuzuncu bölümde yer alan Dvd örneğini bottom-up TDD şeklinde implemente etmiş olduk. Şimdi yeni bir TDD türüyle tanışacağız: Top-down (yukardan aşağıya) TDD.

Daha öncede belirttiğim gibi Shop gibi bir programı bottom-up TDD tarzı implemente etmek bizi çok zorlar. En alttan başlamak, sistemin gereklerini gözden kaybetmemize yol açabilir. Yukardan bir kroki şeklinde sisteme bakıldığında ne yapılması ve nasıl bir yol izlenmesi gerektiği daha kolayca anlaşılabilir. Top-down TDD nin ne olduğunu ve nasıl uygulandığını görmeden önce programın mimari yapısına tekrar bir göz atmamız gerekiyor. Sadece mimari yapıyı iyi anladığımız taktirde, top-down TDD nin nasıl uygulanabileceğini daha iyi anlayabiliriz.

3 Katmanlı Mimari

Günümüzde yapılan kurumsal projelerin temelinde üç ya da daha fazla katmanlı mimariler yatmaktadır.



Resim 10.8 Üç katmalı mimari

Çoğu ugyulamanın amacı veri oluşturmak, bu verileri depolamak, istendiği zaman depolanmış verileri elde edip, değerlendirmek ve belirli sonuçlara ulaşmaktır. Buradaki sihirli kelime „veri“ dir ve bilgisayarın ve internetin icat edilmesinde baş rolü oynamıştır.

Bir firmanın günlük faaliyetlerinde her gün birçok veri oluşur. Bu veriler değerlendirilir ve firmanın bir veya birden fazla veri tabanında depolanır. Her firmanın stratejik faaliyetlerinden birisi, bu veri oluşumunun kontrollü bir şekilde yapılmasını sağlamak olmalıdır. Veri kaybı aynı zamanda firmanın kazanç kaybı anlamına gelebileceği için birçok firma sahip oldukları verilere bilgisayar ve bu bilgisayarlarda çalışan bir takım programlar aracılığı ile sahip çıkmaktadırlar.

Uygulamaların ana amaçlarının veriler üzerinde işlem yapmak olduğunun altını çizdik. Peki program yazma kriterleri nelerdir? Her programcı istediği şekilde gelen istekler doğrultusunda program yazabilir mi? Evet yazabilir ama programcının profesyonelliği oluşan kodun kalitesi ile direk orantılıdır. Edindiğim tecrübeler doğrultusunda bir programın bakımı ve geliştirilmesinin yazılımından daha pahalı olduğunu söyleyebilirim. Tasarım şablonlarının (design patterns) kullanılmadığı ve TDD tarzı yazılım yapılmayan programların bakımı imkansız ya da çok zordur. Bir firma için sahip olduğu veriler ne kadar önemli ise, verileri işlemek için kullandığı programlar ve bu programların bakımı ve geliştirilmesi de bir o kadar önemlidir.

Günümüzde firmalar sahip oldukları verileri kullanmak, saklamak ve işlemek için web tabanlı programlar kullanmaktadırlar. Web tabanlı programların bakımı ve geliştirilmesi masaüstü programlardan daha kolay ve ulaşılan kullanıcı kitlesi daha büyük olduğu için (kullanıcılar genelde bir web tarayıcı – browser ile çalışabilirler) tercih edilmektedirler.

Web projelerde uygulanan ilk katman gösterim (presentation) katmanıdır. Bu katmanda veriler üzerinde işlem yapılmaz. Veriler üzerinde işlem diğer katmanlar tarafından gerçekleştirilir. Gösterim katmanı başka bir katmanda hazırlanmış olan verilerin kullanıcıya gösterimi için kullanılır. Bu katmanda JSP ve Servlet gibi gösterim teknolojileri kullanılarak edinilen veriler sunulur.

Gösterim katmanına veriler işletme katmanı (business) tarafından sağlanır. İşletme katmanında veriler üzerinde yapılacak işlemler tanımlanır. Bunlar Java sınıflarında oluşturulan metotlardır. Business metotları olarak bilinen bu birimlerde firmanın veriler üzerinde yapmak istediği işlemler implemente edilerek, istenilen neticeler elde edilir.

Gösterim katmanı için gerekli veriler veri depolama / edinme (persistence) katmanı tarafından sağlanır. Bu katmanın görevi JDBC ya da Hibernate gibi teknoloji ile veri tabanında yer alan verileri edinmek ve istenilen verileri veri tabanında depolamaktır.

Katmanlar arası iletişim tanımlanmış interface sınıflar üzerinden gerçekleşir. Örneğin gösterim katmanı işletme katmanında bulunan bir Facade interface üzerinden istediği verileri elde edebilir. Gösterim katmanı, işletme katmanı sadece bir interface sınıfından oluşuyormuş gibi düşünülerek, bu interface sınıfına karşı programlandığı taktirde, iki katman arasında esnek bir bağ oluşur. Bu durumda işletme katmanı dış dünyaya sunduğu Facade interface sınıfını istekleri doğrultusunda implemente ederek gösterim katmanını etkilemeden çalışma tarzını tanımlayabilir. Facade interface sınıfında tanımlanmış metotlar değişmediği sürece, işletme katmanında yapılacak değişiklikler gösterim katmanını etkilemez. Sadece bu şekilde hazırlanmış bir program gelecekte meydana gelen değişikliklere ayak uydurabilir yapıda olabilir. Tüm kodun bir katman içinde implemente edilmesi, kullanılan sınıflar arasındaki bağı yükselteceği gibi, bu kodun ilerideki bakımını güçleştirir.

Tekrar TDD Top-Down

Eğer ilerde bakımı ve geliştirilmesi kolay bir program ortaya koymak istiyorsak,

Shop sistemini üç katmanlı bir mimari ile oluşturmakta fayda vardır. Ne yazık ki böyle karmaşık bir yapının bottom-up tarzı implemente edilmesi kolay değildir. Bu yüzden en tepeden başlayarak aşağıya doğru inmemiz gerekiyor. En tepe ile gösterim katmanını kastediyorum. Bu katman bünyesinde kullanıcının gördüğü arayüzleri barındırmaktadır. Bu katmanı test ederek işe başlarsak, programın arkası çorap söküğü gibi gelecektir, çünkü gösterim katmanından sonra işletme ve ondan sonra veri katmanı gelmektedir. Bu tarz TDD uygulanmasına top-down adı verilmektedir.

Gösterim katmanından işe başlayabilmek için ilk önce testler oluşturmamız gerekiyor. Bunlar doğal olarak birim testleri olamaz, çünkü test etmek istediğimiz HTML arayüzleridir. Onay/kabul testleri ile işe başlamamız gerekiyor.

Onay/Kabul Testleri

Sekizinci bölümde onay/kabul testlerini şu şekilde tanımlamıştık:

Onay/kabul testleri ile sistemin bütünü kullanıcı gözüyle test edilir. Bu tür testlerde sistem kara kutu olarak düşünülür. Bu yüzden onay/kabul testlerinin diğer bir ismi kara kutu testleridir (black box testing). Kullanıcının sistemin içinde ne olup bittiğine dair bir bilgisi yoktur. Onun sistemden belirli beklentileri vardır. Bu amaçla sistem ile interaksiyona girer. Onay/kabul testlerinde sistemden beklenen geri bildirim test edilir.

Onay/kabul testleri müşteriye sistemin ne zaman çalışır durumda olduğunu ve programcıya ne implemente etmesi gerektiğini söyler. XP projelerinde onay/kabul testlerini müşteri ve sistem kullanıcıları tanımlar ve bu testler programcılar ya da testçiler tarafından implemente edilir.

XP de geri bildirim (feedback) önemlidir. Onay/kabul testleri müşteri ve proje ekibi için iyi bir geri bildirim mekanizmasıdır. Sistemin çalışır ya da çalışmaz durumda olduğu onay/kabul testleri aracılığıyla öğrenebiliriz. Bunun yanı sıra onay/kabul testleri ile projede ilerleme ölçülebilir.

Müşteri kendisine “neyi test etmem gerekiyor?” sorusunu sorabilir. XP bu soruyu şöyle cevaplar: “sadece çalışmasını istediğin şeyleri test et” Bu implementasyonu tamamlanmış yeni bir program modülü ya da login gibi sistemin temel işlevlerinden birisi olabilir.

Onay/kabul testlerinin JUnit testleri gibi tekrarlanabilir olması gerekmektedir. Genel olarak bir XP projesine oluşturulan tüm testlerin otomatik olarak çalıştırılabilir olmaları gerekir. Projenin herhangi bir safhasında müşteri gereksinimleri değişikliğe uğrayabileceği için mevcut sistem üzerinde değişiklik yapılması kaçınılmaz olabilir. Bu durumda değişikliklerin yan etkilerini test edebilmek için otomatik çalıştırabileceğimiz testlerin mevcut olması gerekmektedir.

Top-down TDD tarzı testlerin çıkış noktası kullanıcı hikayeleridir. Eğer kullanıcı hikayelerini testler ile ifade edebilirsek, kullanıcı hikayelerini ve müşteri gereksinimlerini daha iyi anlayabiliriz. Kullanıcı hikayelerinden yola çıkarak onay/kabul testlerini oluşturmamız gerekiyor, daha doğrusu bunu müşteri yapıyor, implementasyonu programcı olarak biz üstleniyoruz.

İlk onay/kabul testini oluşturduktan sonra çalıştırarak, testin olumsuz sonuç verdiğini görmemiz gerekiyor, çünkü program henüz boş bir kutudur ve fonksiyonel bir işlevi yoktur. Bu noktadan itibaren JUnit testleri ile devam ederek, sistemin bütününü oluşturacak olan sınıf ve metotları implemente edeceğiz. Bu testler tamamlandıktan sonra tekrar onay/kabul testine geri dönerek, ne durumda olduğunu kontrol edeceğiz. Eğer onay/kabul testi çalışır durumda ise, onay/kabul kriterleri tatmin edilmiştir, yani kullanıcının programdan (program arayüzünden) olan beklentisi tatmin edilmiştir. Bu durumda bir sonraki onay/kabul testine geçerek, test sürecini tekrarlayabiliriz. Bu işlem tüm onay/kabul testleri tamamlanana kadar devam eder.

Kullanıcı hikayemizi tekrar gözden geçirerek, onay/kabul testlerini oluşturuyoruz.

Kullanıcı hikayesi 2: Kullanıcı email adresi ve şifreni kullanarak sisteme login yapar.

- ***1. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifre alanlarını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi ve şifrenizi giriniz!" hata mesajı gösterilir.
- ***2. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adresini girer ve şifre alanını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen şifrenizi giriniz!" hata mesajı gösterilir.
- ***3. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adres alanını boş bırakarak şifresini girer ve login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi giriniz!" hata mesajı gösterilir.

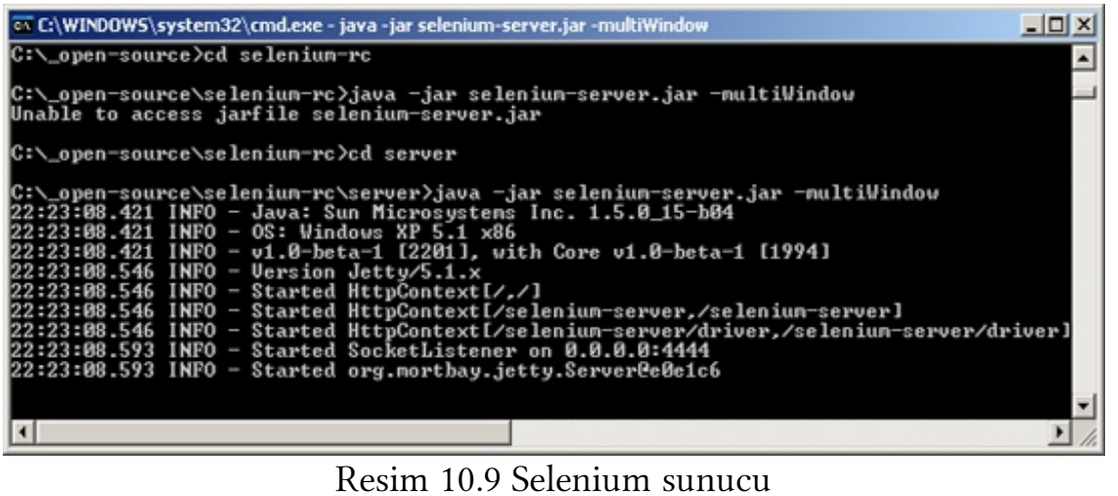
- ***4. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi ve şifre doğrudur. Login işlemi gerçekleşir. Üye hoş geldiniz sayfasına yönlendirilir.
- ***5. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçersizdir. Kullanıcıya "Email adresiniz geçersizdir, lütfen tekrar ediniz!" hata mesajı gösterilir.
- ***6. Onay/kabul Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçerli olmasına rağmen şifre hatalıdır. Kullanıcıya "Şifre hatalı, lütfen tekrar deneyiniz!" hata mesajı gösterilir.

Tanımladığımız bu altı onay/kabul testinin olumlu sonuç vermesi seçtiğimiz kullanıcı hikayesinin eksiksiz olarak implemente edildiği anlamına gelmektedir. Bu onay/kabul testlerini oluşturmak için Selenium programından faydalanacağız. Selenium hakkında geniş bilgiyi on birinci bölümden edinebilirsiniz.

Selenium İle İlk Onay/kabul Testi

Onay/kabul testlerini JUnit benzeri Java sınıfları olarak hazırlayacağız. Testleri çalıştırabilmek için Selenium sunucusuna (Selenium RC) ihtiyacımız var. Bu programı Selenium websayfasından adresinden temin edebilirsiniz. Kurulumu herhangi bir dizine gerçekleştirdikten sonra Selenium sunucuyu bir sonraki komut ile çalıştırıyoruz:

```
java -jar selenium-server.jar
```



Resim 10.9 Selenium sunucu

Selenium sunucusu localhost 4444 nolu portta çalışır hale geldikten sonra ilk

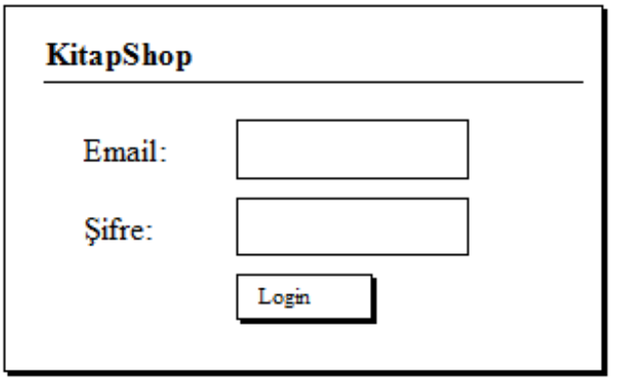
onay/kabul testini oluşturabiliriz.

Onay/kabul testinin müşteri tarafından nasıl tanımlandığına tekrar bir göz atalım:

1. Onay/kabul Testi

Kullanıcı login sayfasına gider. Email adresi ve şifre alanlarını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi ve şifrenizi giriniz!" hata mesajı gösterilir.

Şimdi gözümüzde Shop sistemine login yapmak için kullanılan bir login sayfası canlandıralım. Aslında bunu yapmamıza gerek yok, çünkü beşinci bölümdeki projenin keşif safhasında kullanıcı arayüz prototiplerini oluşturmuştuk. Login sayfası arayüzü şu şekilde prototip edilmiştir:



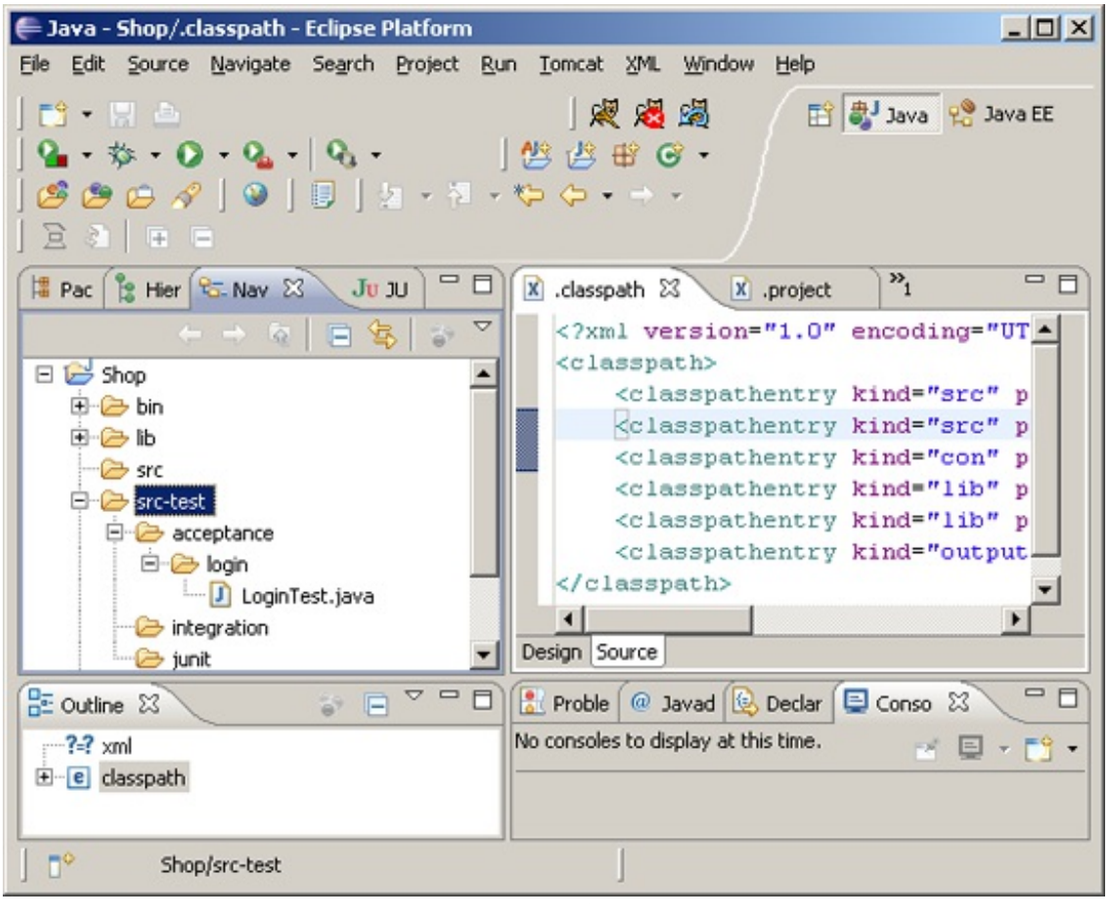
Resim 10.10 Login yapma sayfası

Bu prototip en basit haliyle bir login sayfasını gösteriyor. Bu tip prototipler programcılar tarafından keşif safhasında kağıt ya da yazı panolarında oluşturulur. Müşteri isteklerinin kavranması açısından prototipler büyük önem taşımaktadır.

HTML arayüz login işlemini yapabilmek için email adresinin ve kullanıcı şifresinin girilebileceği iki alan ve birde Login butonunu ihtiva etmektedir. İlk onay/kabul testimizi oluştururken böyle bir sayfanın daha önceden programlandığını farz ederek testi oluşturacağız. Birinci onay/kabul testinin amacı email ve şifre alanlarına kullanıcı tarafından herhangi bir veri girilmeden login butonuna tıklanması durumunda sistemin nasıl bir reaksiyon göstereceğini test etmektir. Kullanıcı beklentisi eğer email ve şifre alanları boş bırakılırsa sistem tarafından "Lütfen email adresinizi ve şifrenizi giriniz!" şeklinde bir hata mesajının ekranda gösterilmesi şeklindedir. Bu tanımlanan davranış haricinde sistemin değişik bir davranış göstermesi, onay/kabul testinin

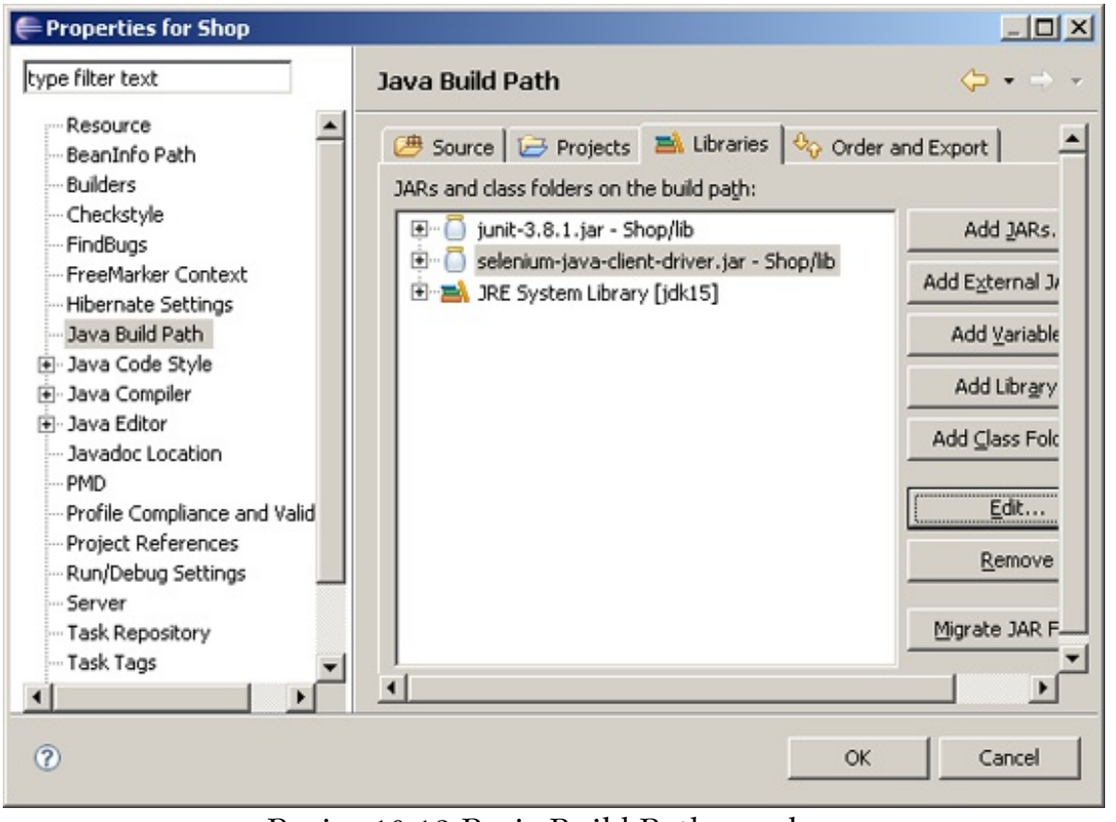
olumsuz sonuç verdiğinin ve kullanıcı hikayesinin bir bölümünün yanlış implemente edildiğinin göstergesidir.

Program kodları src ve test kodları src-test isimli dizinlerde tutulmalıdır. Ayrıca değişik türdeki testleri birbirinden ayırmak için src-test dizini altında acceptance, junit ve integration isminde dizinler oluşturuyoruz. Bu işlemlerin ardından Eclipse proje yapısı resim 10.11 de görüldüğü gibi olmalıdır.



Resim 10.11 Yeni Eclipse proje yapısı

İlk onay/kabul testi için src-test/acceptance/login dizininde LoginTest.java isminde bir Java sınıfı oluşturuyoruz. Selenium sunucusu ile bağlantı kurabilmek için junit.jar ve selenium-java-client-driver.jar dosyalarının Build Path ına eklenmesi gerekiyor.



Resim 10.12 Proje Build Path ayarları

Selenium ile hazırladığımız ilk onay/kabul testimiz kod 10.1 de yer almaktadır.

Kod 10.1 İlk akseptans test sınıfı

```
package acceptance.login;

import com.thoughtworks.selenium.*;

public class LoginTest extends SeleneseTestCase
{
    public void setUp() throws Exception
    {
        setUp("http://localhost/", "*chrome");
    }


    public void testEmailAndPasswordEmpty() throws Exception
    {
        selenium.open("/");
        selenium.click("link=Login");
        selenium.waitForPageToLoad("30000");
        selenium.click("Login");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Lütfen email adresinizi " +
                "ve şifrenizi giriniz!"));
```

```
    }
}
```

LoginTest sınıfı SeleneseTestCase sınıfını genişletmektedir. SeleneseTestCase sınıfı junit.framework.TestCase sınıfını genişlettiği için LoginTest sınıfı gerçek bir JUnit test sınıfı haline gelmektedir. O halde Selenium onay/kabul testlerini JUnit testleri gibi yapılandırabiliriz. Bunun için değişik test metotları oluşturarak, bir sınıf bünyesinde birden fazla onay/kabul test metodunun barınmasını sağlayabiliriz.

İlk test metodunun ismini testEmailAndPasswordEmpty() olarak seçiyorum. Bu metot ile birinci onay/kabul testini oluşturuyor olacağız. Şimdi bu metodun gövdesini oluşturan komutları yakından inceleyelim:

```
selenium.open("/");
```

Bu komut ile http://localhost/ adresine erişiyoruz.

```
selenium.click("link=Login");
```

Yüklenen ilk sayfada bulunan Login isimli linke tıklıyoruz.

```
selenium.waitForPageToLoad("30000");
```

Otuz saniye kadar sayfanın yüklenmesi için beklenebilir.

```
selenium.click("Login");
```

Yüklenen sayfada bulunan Login butonuna tıklıyoruz.

```
selenium.waitForPageToLoad("30000");
```

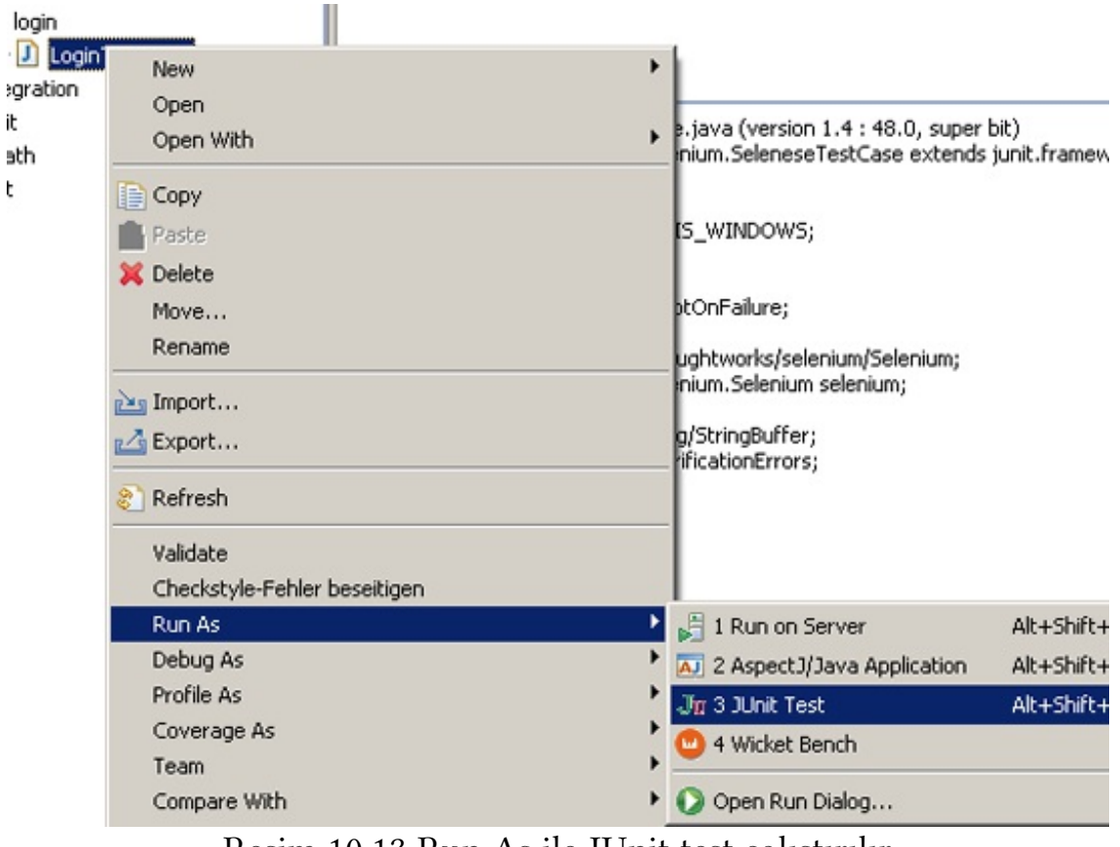
Sayfanın yüklenmesi için otuz saniyeye kadar beklenebiliyor.

```
verifyTrue(selenium.isTextPresent("Lütfen email adresinizi " +
                "ve şifrenizi giriniz!"));
```

Yüklenen sayfada "Lütfen email adresinizi ve şifrenizi giriniz!" şeklinde bir mesajın olup, olmadığını kontrol ediyoruz. Böyle bir mesajın varlığı durumunda, testimiz başarıyla tamamlanıyor, aksi taktirde test bir hata ile son buluyor.

İlk Selenium testimizi gözden geçirdiğimizde, testin aslında bir kullanıcının login işlemi için atması gereken adımları simüle ettiğini görmekteyiz. Burada bir kullanıcı (şahıs) yerine bir program yardımıyla sistemi HTML arayüzünden test etmiş oluyoruz. Bu sebepten dolayı testimiz bir onay/kabul testi haline gelmektedir, çünkü burada test edilen sistemin dış yüzü, HTML arayüzüdür. Shop sisteminin parçası olan herhangi bir sınıfı ya da komponenti değil, login modülünün tümünü dış dünyadan, sınıfların içine girmeden test ediyoruz.

Bu testi Eclipse altında sağ tuş ile erişeceğimiz menüden Run As > JUnit Test şeklinde çalıştırabiliriz. Bunun nasıl yapıldığını resim 10.13 de görmekteyiz.



Resim 10.13 Run As ile JUnit test çalıştırılır

Testin çalışabilmesi için ayrıca Selenium sunucusunun resim 10.9 da görüldüğü gibi çalışıyor durumda olması gerekmektedir. Test metodunda yer alan komutlar Selenium sunucusu aracılığıyla Selenium sunucusunun çalıştırdığı web tarayıcısına iletilecektir. Selenium sunucusu test ile web tarayıcısı arasında komutları ve sonuçları taşıyan bir proxy vazifesi görmektedir. Selenium sunucusu hakkında detaylı bilgiyi bir sonraki bölümden edinebilirsiniz.

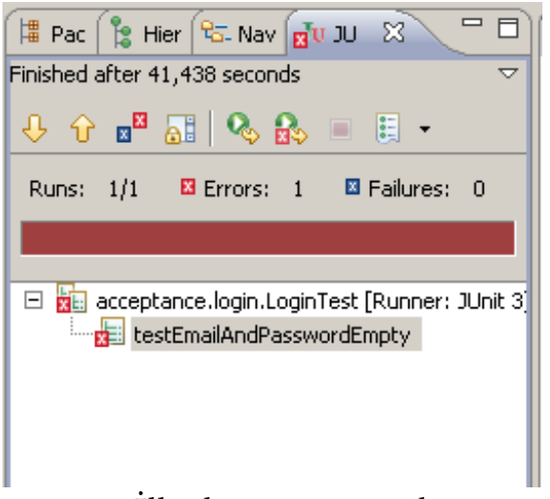
Test koşmaya başladıktan sonra Selenium sunucusu ile 4444 nolu port üzerinden bağlantı kurar. Selenium sunucusunun yaptığı işlemler kod 10.2 yer almaktadır.

```
INFO - Java: Sun Microsystems Inc. 1.5.0_15-b04
INFO - OS: Windows XP 5.1 x86
INFO - v1.0-beta-1 [2201], with Core v1.0-beta-1 [1994]
INFO - Version Jetty/5.1.x
INFO - Started HttpContext[/,/]
INFO - Started HttpContext[/selenium-server,/selenium-server]
INFO - Started HttpContext[/selenium-server/driver,
                           /selenium-server/driver]
INFO - Started SocketListener on 0.0.0.0:4444
INFO - Started org.mortbay.jetty.Server@e0e1c6
INFO - Checking Resource aliases
INFO - Command request: getNewBrowserSession[*chrome,
                        http://localhost/] on session null
INFO - creating new remote session
INFO - Allocated session f9073cd123454bb98dc0840b0df61aff
                        for http://localhost/, launching...
INFO - Preparing Firefox profile...
INFO - Launching Firefox...
INFO - Got result: OK,f9073cd123454bb98dc0840b0df61aff on
                        session f9073cd123454bb98dc0840b0df61aff
INFO - Command request: setContext[LoginTest.
                        testEmailAndPasswordEmpty
                        on session f9073cd123454bb98dc0840b0df61aff
INFO - Got result: OK on session
                        f9073cd123454bb98dc0840b0df61aff
INFO - Command request: open[/, ] on session
                        f9073cd123454bb98dc0840b0df61aff
INFO - Got result: Timed out after 30000ms on session
                        f9073cd123454bb98dc0840b0df61aff
INFO - Command request: testComplete[, ] on session
                        f9073cd123454bb98dc0840b0df61aff
INFO - Killing Firefox...
INFO - Got result: OK on session f9073cd123454bb98dc0840b0df61aff
```

On birinci satırdan itibaren Selenium sunucusu çalıştırdığımız teste cevap vermeye başlamıştır. Selenium sunucusu ilk iş olarak sistemde bulunan Firefox tarayıcısını faal hale getiriyor. On altıncı satırda Firefox web tarayıcısının çalışır duruma geldiğini görüyoruz. On yedinci satırdan itibaren LoginTest.testEmailAndPasswordEmpty() metodunda bulunan komutlar işlem görmeye başlıyor. On dokuzuncu satırda selenium.open("/"); komutunu Selenium sunucu tarafından web tarayıcısına gönderildiğini görüyoruz.

http://localhost, yani kendi bilgisayarımızda bir web sunucusu çalışır durumda olmadığı için test otuz saniye sonra aşağıda yer alan hata mesajıyla son bulur.

```
com.thoughtworks.selenium.SeleniumException: Timed out after 30000ms
```



Resim 10.14 İlk akseptans testi hata veriyor

http://localhost adresinde bir web sunucu çalışıyor olsaydı bile bu test olumlu sonuç vermezdi, çünkü test etmek istediğimiz login modülü henüz implemente edilmedi. Ama biz ilk adımı atarak, yani ilk onay/kabul testini oluşturarak, kullanıcı açısından sistem beklentilerini ifade ettik. Bu noktadan itibaren boş bir kutu olan programı login modülünü implemente etmeye başlayarak, dolduracağız. Login modülünü implemente ettikten sonra tekrar bu onay/kabul testine geri dönerek, testin ne durumda olduğunu kontrol edeceğiz. Login modülünün implemente edildiğinin tek ispatı, bu ve oluşturacağımız diğer onay/kabul testlerinin çalışır durumda olmasıdır.

Şimdi login modülünü implemente etmek için bu testi bu halde bırakıp, birim testlerine geçiyoruz. TDD de bir testi yarıda bırakarak, başka bir teste geçip, bir önceki test gereksinimlerini tatmin edecek şekilde gerekli sınıfları oluşturmak normaldir. Bir onay/kabul testini çalışır hale getirmek için de birim testlerine geçilmesi de doğaldır.

Tasarım Oturumu (Design Session)

Bir kullanıcı hikayesinden yola çıkarak onay/kabul testleri oluşturduğumuz zaman onay/kabul testleri bize kullanıcı hikayesinin nasıl implemente edilmesi gerektiği konusunda fikir verir. Bu testlerden yola çıkarak test güdümlü implementasyonu gerçekleştirebiliriz.

Bazı durumlarda onay/kabul ve diğer tür testler nasıl bir tasarımın (design) oluşturulması gerektiği hakkında bize ip uçları vermeyebilir. Bu durumda

ekipten birkaç arkadaş ile bir araya gelip, kısa bir tasarım oturumu gerçekleştirmemizde fayda vardır. Bu oturum yarım saati geçmeyecek şekilde fikir alışverişine dayanan bir yapıda olmalıdır. UML diyagramları çizilerek tasarım hakkında görüş alışverişinde bulunulabilir. Buradaki ana amaç tasarım içinde yer alan sınıfları keşfetmek ve nasıl implemente edilebilecekleri hakkında fikir oluşturmaktır.

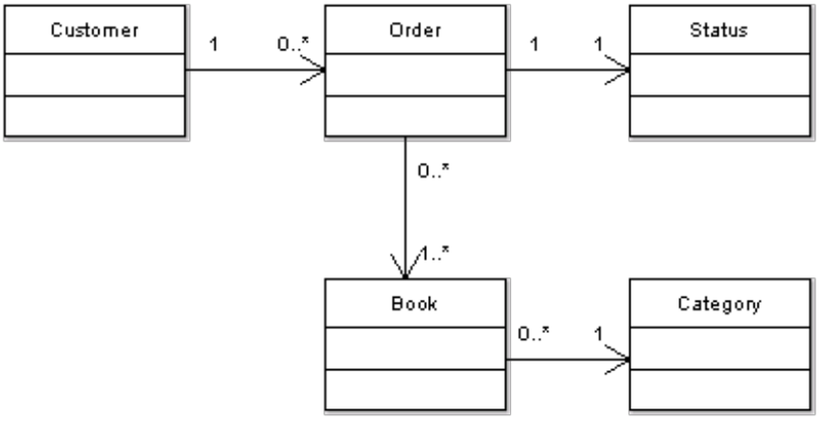
Tasarım oturumunda fazla zaman kaybedilmemelidir. Tasarım için gerekli sınıflar bir alan (domain) modelinde bir araya getirildikten sonra implementasyon için hemen bilgisayar başına geçilmeli ve test güdümlü implementasyon gerçekleştirilmelidir. Eğer bu mümkün değilse oluşturmak istediğimiz program hakkında bir fikir edinebilmek için alan modelinde yer alan sınıfları prototip olarak implemente ederek başlayabiliriz.

Böyle bir tasarım oturumu ne yapılmasının keşfini sağladığı için psikolojik olarak bizi (programcıyı) rahatlatacaktır. Ekip arkadaşlarımızla yaptığımız fikir alışverişi, implementasyonun nasıl yapılması gerektiği konusunda ip uçları verecektir.

Oturum sonunda implementasyon için birden fazla alternatif oluşmuş olabilir. Bunların içinden en basit olanı seçilerek, implementasyon gerçekleştirilmelidir.

Alan (Domain) Modeli ve Tasarım

Kitabın beşinci bölümünde keşif safhası kapsamında Shop sistemi için bir alan modeli oluşturmuştuk.



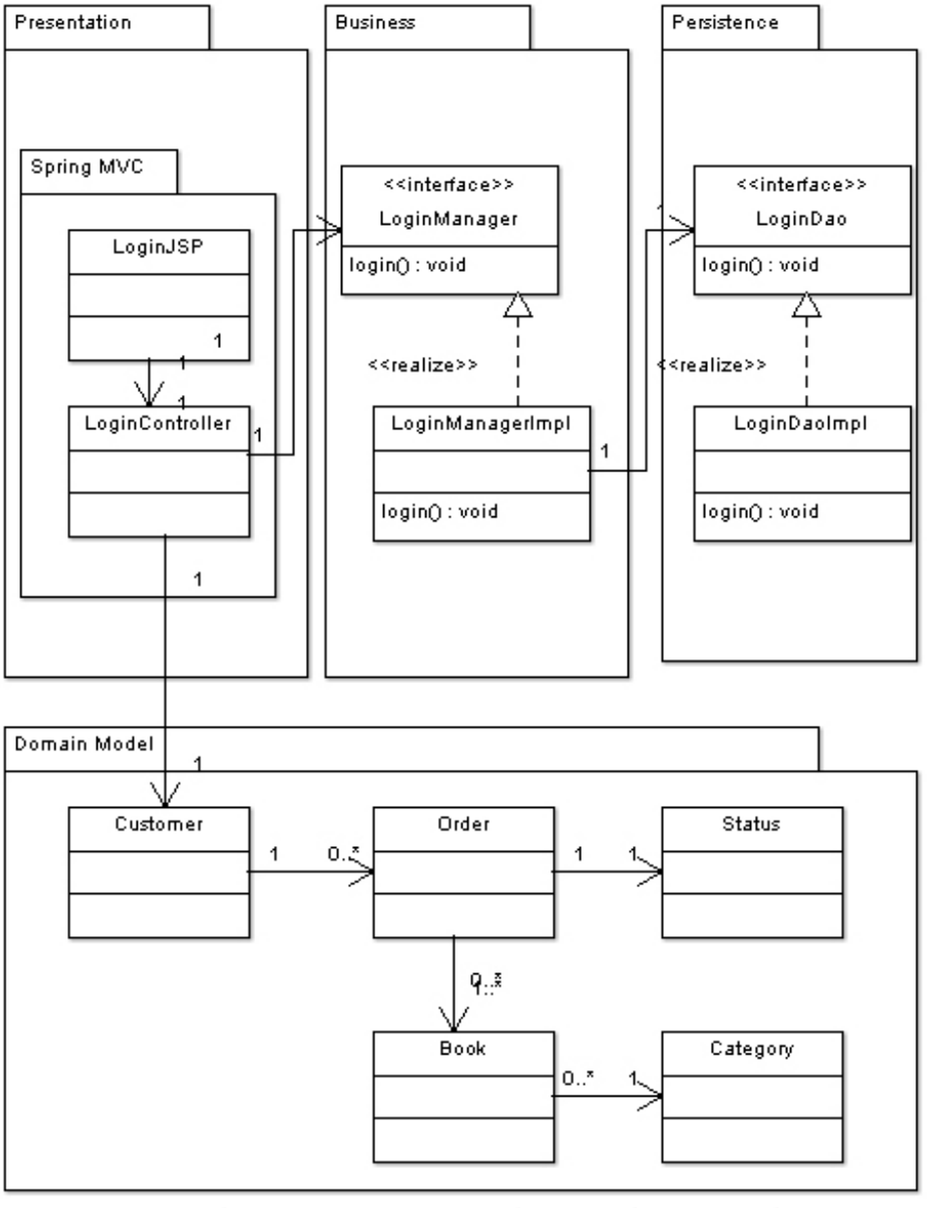
Resim 10.15 Shop sistemi alan modeli

Alan modelinde yer alan sınıfları tekrar hatırlayalım:

- ***Customer***: Bir müşteriyi modellemek için kullanılır.
- ***Order***: Müşterinin verdiği bir siparişi modellemek için kullanılır. Customer ve Order arasında 1 – 0..* ilişkisi vardır. Buna göre bir müşteri sıfır, bir ya da birden fazla sipariş verebilir. Her sipariş sadece bir müşteriye aittir.
- ***Status***: Her siparişin bir statüsü vardır.
- ***Book***: Shop sisteminde kitaplar satılacağı için Book isminde bir sınıf bulunmaktadır. Book ile Order arasında 1..* - 0..* şeklinde bir ilişki vardır. Buna göre kitap sıfır, bir ya da birden fazla sipariş içinde yer alabilir, bir sipariş içinde en az bir kitap bulunmak zorundadır.
- ***Category***: Shop sisteminde kitaplar değişik kategorilerde yer alır. Book ve Category sınıfları arasında 0..* - 1 ilişkisi vardır. Buna göre bir kitap sadece bir kategori içinde yer alabilir, bir kategori içinde sıfır, bir ya da birden fazla kitap bulunabilir.

Alan modeli bize hangi sınıfların birbirleriyle nasıl bir ilişki içinde olduğuna dair fikir veriyor, lakin implementasyonun nasıl yapılacağı hakkında bir bilgi ihtiva etmiyor.

İmplementasyon hakkında ekip arkadaşlarımızla fikir alışverişinde bulunabilmek için UML diyagramları oluşturabiliriz. Bunun bir örneğini resim 10.16 da görmekteyiz. Daha öncede belirttiğim gibi üç değişik ve sorumluluk alanları tanımlanmış katmandan oluşan bir tasarım oluşturmakta fayda vardır. Böyle bir tasarım gelecekte meydana gelecek olan değişiklikleri göğüsleyebilecek yapıda olacaktır. Her katman interface sınıfları aracılığıyla kullanıcı katman için erişilebilir hale getirilir. Bu bize kullanıcı katmanları etkilemeden interface sınıfının implementasyonunu değiştirme imkanını tanıyacak ve hatta gerekli durumlarda bir katmanının implementasyonunu tamamen değiştirebilme özgürlüğünü sunacaktır. Shop sistemi için düşündüğüm tasarım resim 10.16 da yer almaktadır.



Resim 10.16 Shop sistemi için üç katmanlı mimarik tasarım

Tasarım üç katman artı bu katmanlarda kullanılan alan model nesnelerinden oluşmaktadır. İlk katman gösterim (presentation) katmanıdır. Bu katman kullanıcıların gördüğü HTML arayüzleri ihtiva etmektedir. Bu katmanı Spring MVC frameworkü ile implemente edeceğiz. İkinci katman işletme (business) katmanıdır. Bu katmanda gösterim katmanı istekleri doğrultusunda gerekli işlemlerin implementasyonu yapılır. Örneğin gösterim katmanı gerekli verileri edinmek için direk veri tabanına bağlanmaz. İşletme katmanına başvurarak gerekli verilerin elde edilmesini sağlar. Bu katmanı Java sınıfları (POJO) kullanarak implemente edeceğiz. Üçüncü katman veri edinme / kaydetme (persistence) katmanıdır. Bu katman içinde işletme katmanının istekleri doğrultusunda veri tabanından veriler alınır ya da kaydedilir. Bu katmanı Hibernate teknolojisi ile implemente edeceğiz.

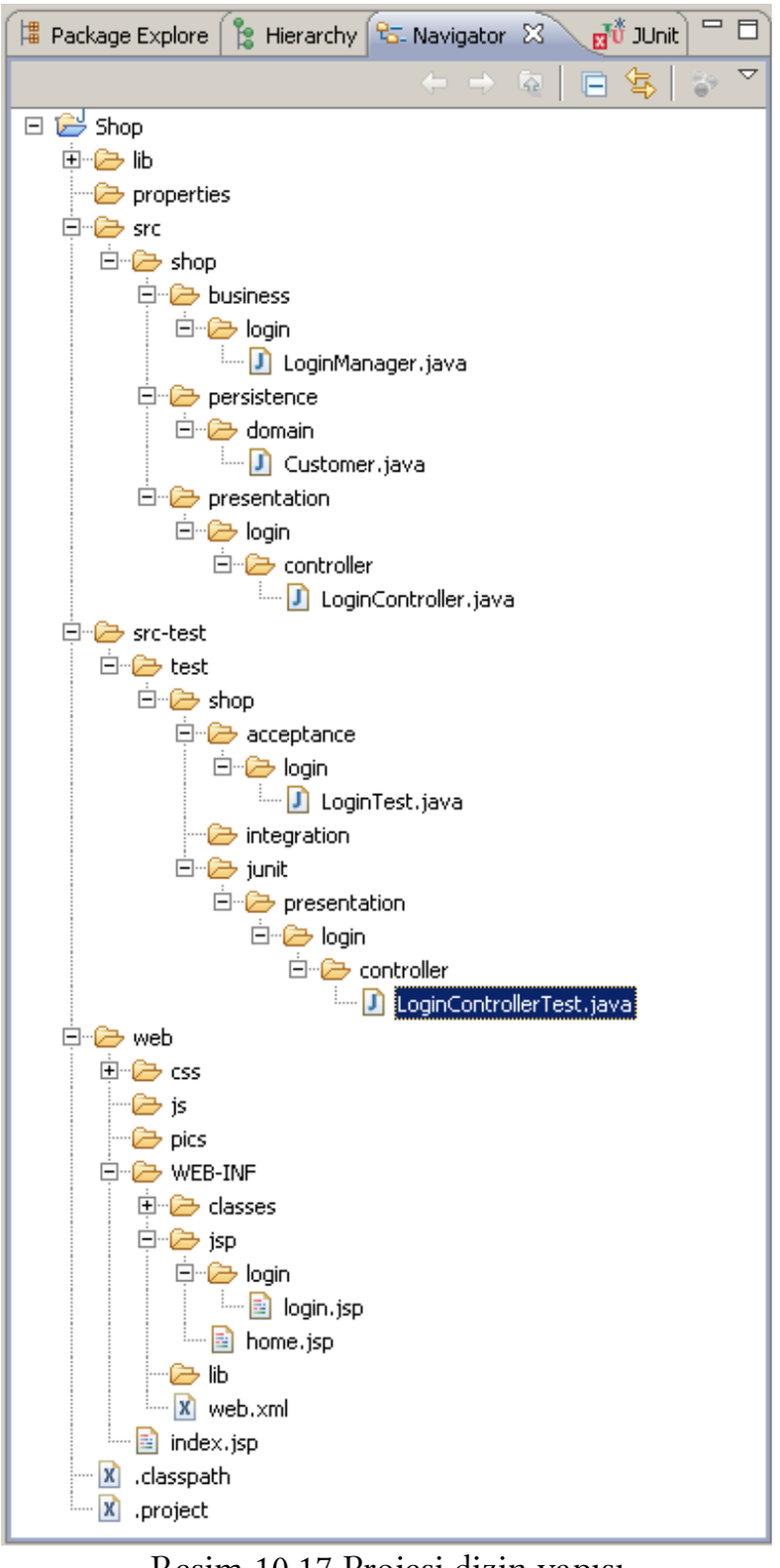
İlk önce gösterim katmanının implementasyonu ile başlayabiliriz.

Gösterim (Presentation) Katmanı

Oluşturmak istediğimiz Shop sistemi web tabanlı bir programdır. Bu yüzden web tabanlı bir uygulamayı implemente etmek için bir web çatısına ihtiyacımız var. Spring MVC ihtiyacımızı karşılayacak özelliklere sahip bir web çatısıdır. Spring MVC hakkında detaylı bilgiyi kitabın on üçüncü bölümünde edinebilirsiniz. Bu bölümü okumadan önce Spring ve Spring MVC hakkında bilgi edinmenizde fayda vardır.

Spring MVC ile implemente edeceğimiz gösterim katmanında login işlemini gerçekleştirmek için aşağıda yer alan sınıflara ve bir JSP sayfasına ihtiyacımız var:

1. LoginController.java: Login işlemini gerçekleştirmek için kullandığımız Spring MVC Controller sınıfıdır.
2. LoginManager.java: LoginController sınıfı tarafından login işlemini gerçekleştirmek için kullanılan ve işletme katmanında yer alan sınıftır.
3. login.jsp: Login işlemi için kullanılan HTML arayüzü ihtiva eden JSP sayfasıdır.
4. Customer.java: Bir müşteri hakkında bilgileri tutan model sınıfıdır.



Resim 10.17 Projesi dizin yapısı

Projenin dizin yapısını resim 10.17 de yer aldığı şekilde geliştiriyoruz. Hangi dizinde içinde hangi dosyaların yer aldığına bir göz atalım:

Tablo 10.3: Proje dizin yapısı

Dizin	Kullanım amacı
lib	Proje içinde kullanılan Jar dosyalarını barındırır.
properties	Konfigürasyon dosyalarının yer aldığı dizindir. Örneğin Spring applicationContext.xml bu dizinde yer alır.
src	Program kodu src dizininde yer alır. Her katman için özel bir paket oluşturulmuştur.
src-test	Akseptans, entegrasyon ve JUnit testleri bu dizide yer alır.
web	Shop sisteminin web tabanlı bölümü bu dizin içinde yer alır. Web root (web ana dizin) dizinidir.
web/css	Shop sistemi için Cascading Style Sheet (CSS) dosyalarını ihtiva eder.
web/js	Shop sistemi için hazırlanan Javascript dosyaları ihtiva eder.
web/pics	HTML arayüzlerinde kullanılan grafikler bu dizinde bulunur. /pics URI'sinden bu resimlere bir webserver altında erişilir.
web/WEB-INF	Web tabanlı bir Java programı için gerekli kofigürasyon dosyaları (örneğin web.xml) , Jar dosyaları ve JSP sayfaları bu ve alt dizinlerde yer alır.
WEB-INF/classes	Proje içinde derlenen Java sınıfları bu dizine kopyalanır (build output dir). Tomcat Shop sistemi için gerekli sınıfları bu dizin içinde arar.
WEB-INF/lib	Shop sistemi tarafından kullanılan Jar dosyaları bu dizinde yer alır, örneğin spring.jar, hibernate.jar.
WEB-INF/jsp	HTML arayüzlerini barındıran JSP sayfaları bu dizinde yer alır.

Gösterim katmanında yer alan LoginController sınıfını birim testleri oluşturarak test edebiliriz. Bu birim testlerini oluştururken Spring'in sundugu test çatısından faydalanabiliriz. Spring test çatısı Controller sınıflarının uygulamayı bir uygulama sunucusunda çalışır hale getirmek zorunda olmadan test edilmelerini mümkün kılmaktadır. Bu testler entegrasyon testlerini andırıyor olsalarda, entegrasyon değil, birim testleridir, çünkü controller sınıfı tarafından kullanılan ve servis katmanında yer alan sınıflar için mock nesneler kullanabiliriz.

İlk birim testini LoginController sınıfı test etmek için oluşturabiliriz. Henüz gerekli sınıfların hiç birisi oluşturulmadı. İlk test sınıfı src-

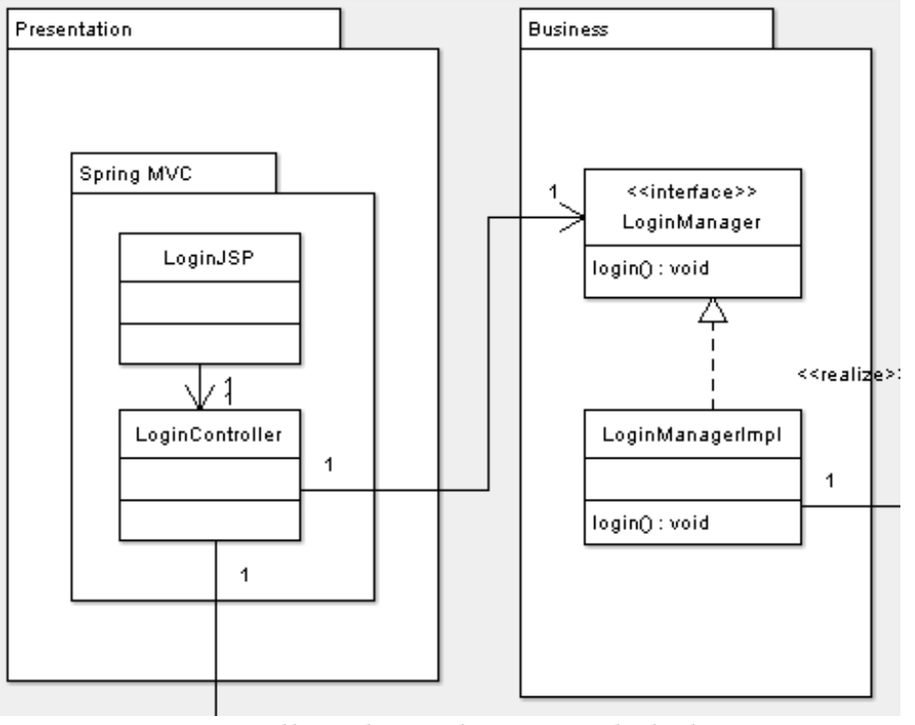
test/test/shop/unit/presentation/login/controller dizininde olacak ve LoginControllerTest ismini taşıyacak.

Oluşturmak istediğimiz testleri tekrar bir gözden geçirelim. LoginController sınıfı gösterim katmanının bir parçası olduğu için LoginControllerTest sınıfında oluşturacağımız testler daha önce belirlediğimiz onay/kabul testleriyle benzerlik taşıyabilir. Onay/kabul testlerini baz alarak birim test listesini oluşturuyoruz

Kullanıcı hikayesi 2: Kullanıcı isim ve şifreni kullanarak sisteme login yapar.

- ***1. Birim Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifre alanlarını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi ve şifrenizi giriniz!" hata mesajı gösterilir.
- ***2. Birim Testi*** - Kullanıcı login sayfasına gider. Email adresini girer ve şifre alanını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen şifrenizi giriniz!" hata mesajı gösterilir.
- ***3. Birim Testi*** - Kullanıcı login sayfasına gider. Email adres alanını boş bırakarak, şifresini girer ve login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi giriniz!" hata mesajı gösterilir.
- ***4. Birim Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi ve şifre doğrudur. Login işlemi gerçekleşir. Üye hoş geldiniz sayfasına yönlendirilir.
- ***5. Birim Testi*** - Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçersizdir. Kullanıcıya "Email adresiniz geçersizdir, lütfen tekrar ediniz!" hata mesajı gösterilir.
- ***6. Birim Testi***- Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçerli olmasana rağmen şifre hatalıdır. Kullanıcıya "Şifre hatalı, lütfen tekrar deneyiniz!" hata mesajı gösterilir.

İlk Testi oluşturmadan önce tekrar tasarımı gözden geçirmemizde fayda vardır. Bu bize testi ve implementasyonu nasıl yapılandırmamız gerektiği hakkında fikir verecektir.



Resim 10.18 LoginController işletme katmanında bulunan LoginManager interface sınıfını kullanmaktadır

LoginController sınıfının işlevini yerine getirebilmesi için işletme katmanında bulunan LoginManager interface sınıfı ile ortak çalışması gerekiyor, yani bir bağımlılık söz konusu. Bu şu anlama gelmekte: "LoginController sınıfı LoginManager sınıfı olmadan kendi görevini yerine getiremez. LoginController sınıfını test edebilmek için LoginManager sınıfının önceden oluşturulmuş olması gerekiyor". Test güdümlü implementasyonu gerçekleştirmek istiyoruz, lakin LoginController sınıfı için bir test oluşturarak işe başlamak yerine, LoginManager sınıfı ile işe başlamamız doğru olmaz. Bu noktayı açıklığa kavuşturmamız gerekiyor!

Test sınıfları oluşturmanın ana zorluklarının başında test edilen sınıfların ortak çalıştıkları diğer sınıfların mevcut olması gelmektedir. Yazılım sisteminde bulunan sınıfların çoğu tek başlarına değildirler. Görevlerini yerine getirmek için diğer sınıflarla ortak çalışırlar. Bu bağımlılıkları test esnasında göz önünde bulundurmamız gerekiyor.

Sınıfların diğer sınıflarla ortak çalışmaları, yani bağımlılıkların olması kötü bir durum değildir. Aksine nesnelerden oluşan bir sistemde sınıfları küçük tutabilmek ve test edilebilir olmalarını sağlayabilmek için bağımlılıkların olması gerekir. Lakin bu bağımlılıklar testleri zorlaştırır. Bunun sebepleri:

1. Top-Down TDD uygulayabilmek için önce test edilen sınıfların bağımlı

olduğu sınıfların oluşturulması gerekiyor. Bizim örneğimizde LoginController sınıfını test edebilmek için LoginManager sınıfını oluşturmamız gerekiyor, çünkü LoginController sınıfı görevini yerine getirebilmek için LoginManager sınıfına ihtiyaç duymaktadır. Bu durumda testi bırakarak, implementasyon detaylarına girmemiz gerekiyor. Bu durum bizi yapılması gereken testten uzaklaştırır.
2. Bağımlı olunan sınıflardan nesne oluşturulması zaman alıcı olabilir. Bu hem testin karmaşık bir yapıda olmasını hem de testlerin uzun sürmesini beraberinde getirir. Bu durum test yazılımını zorlaştırır.
3. Bağımlı olunan sınıfların kullanımı beraberinde istenmeyen diğer bağımlılıkların oluşmasını sağlar. Örneğin LoginManager sınıfı veri tabanı işlemlerini yerine getirmek için LoginDao interface sınıfına ihtiyaç duymaktadır. Bu LoginController sınıfını test edebilmek için LoginManager ve LoginDao sınıflarının test öncesi oluşturulması anlamına geliyor.

Bu sorunları aşmak için sekizinci bölümde tanıştığımız stub implementasyon nesnelerden faydalanabiliriz. Bir stub nesne test esnasında kullanılan ve gerçek olmayan olmayan, bir bağımlılığı temsil eden sahte bir implementasyondur. Stub nesneler kullanarak test içinde test edilen sınıfın bağımlılıklarını, onları implemente etmek zorunda kalmadan simüle edebiliriz. Bizim örneğimizde bu şu anlama gelmektedir: LoginManager sınıfını taklit eden bir stub nesne oluşturarak, bu stub nesneyi LoginControllerTest test sınıfında LoginManager implementasyonu olarak kullanabiliriz.

Bu açıklamanın ardından tekrar LoginController sınıfı için hazırlamak istediğimiz test sınıfına geri dönelim. İlk işlem olarak src-test/test/shop/unit/presentation/login/controller dizininde LoginControllerTest.java isminde bir test sınıfı oluşturuyoruz.

```
Kod 10.3  LoginControllerTest.java - 1. Birim Testi  implementasyonu

package test.shop.unit.presentation.login.controller;

public class LoginControllerTest
{
    @Test
    public void testLoginUsernameAndPasswordEmpty()
    {
    }
}
```

İlk test metodumuzu kullanıcının email adresi ve şifreni girmeden login

butonuna tıklamasıyla oluşması gereken hata mesajlarını test edecek şekilde yapılandırıyoruz.

Kod 10.4 LoginControllerTest.java - 1. Birim Testi implementasyonu

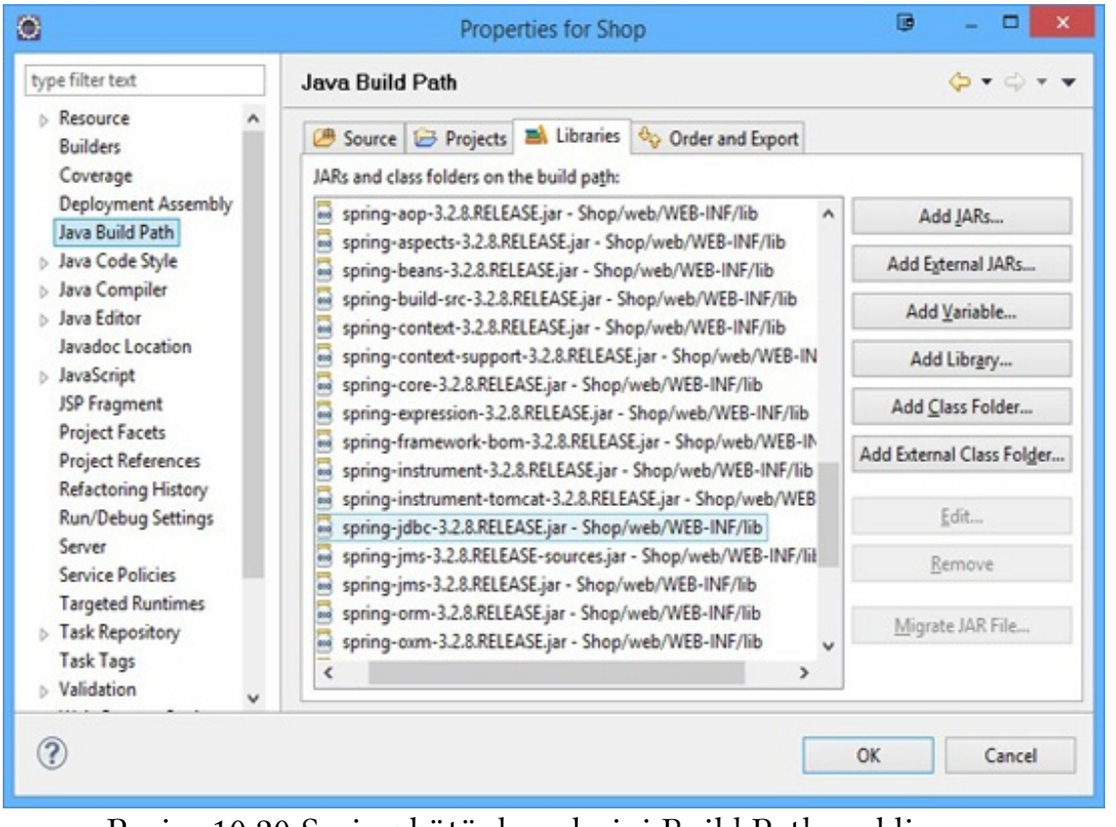
```
package test.shop.unit.presentation.login.controller;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:spring-servlet-test.xml" })
public class LoginControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(
                this.wac).build();
    }

    @Test
    public void testLoginUsernameAndPasswordEmpty()
            throws Exception {

        this.mockMvc
            .perform(post("/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("email", "")
            .param("password", ""))
            .andExpect(model().attributeHasFieldErrors("loginForm",
                        "email"))
            .andExpect( model().attributeHasFieldErrors("loginForm",
                        "password"))
            .andExpect(model().attributeErrorCount("loginForm", 2));

    }
}
```

Spring MVC uygulamalarını test etmenin en kolay yolu MockMvc sınıfını kullanmaktır. Bu sınıf bize kod 10.4 de görüldüğü gibi /login adresinde yer alan controller sınıfını Spring MVC uygulaması bünyesinde koşturma imkanı sunmaktadır. Bunun için Spring MVC uygulamasını Tomcat içinde çalısır hale getirmemiz gerekmemektedir. Daha ziyade Spring MVC uygulamasını

koşturabilmek için @ContextConfiguration anotasyonu ile kullandığımız Spring konfigürasyon dosyasını tanımlamamız yeterli olmaktadır.

testLoginUsernameAndPasswordEmpty() metodu bünyesinde yer alan birim testi bir kullanıcının email ve şifresini girmeden login butonuna tıklamasını simüle etmektedir. param() metotları bünyesinde form değişken isimleri tanımlanmakta, lakin bu değişkenlere bir değer atanmamaktadır. Bu şekilde uygulamadan beklediğimiz davranışı tetikleyerek, model().attributeHasFieldErrors() ile Spring MVC tarafından gerekli alanlar için hata mesajlarının oluşturulup, oluşturulmadıklarını kontrol edebiliriz.

Testin derlenebilmesi için Spring kütüphanelerinin (Jar dosyaları) projenin Build Path ına eklenmesi ve LoginController sınıfının oluşturulması gerekiyor. Gerekli Jar dosyalarını resim 10.20 görmekteyiz.



Resim 10.20 Spring kütüphanelerini Build Path a ekliyoruz

Testin derlenebilmesi için şimdilik LoginController sınıfını en basit haliyle oluşturuyoruz. LoginController sınıfını yapılandırmadan önce Spring konfigürasyon ayarlarını yapmamız gerekiyor, aksi taktirde testin ihtiyaç duyduğu altyapı çalışmaz halde olacaktır. LoginController sınıfının implemente edilmesi bile bu durumu değiştirmez!

Kod 10.5 LoginController.java

```
@Controller
@RequestMapping("/login")
public class LoginController {

    @RequestMapping(method = RequestMethod.POST)
    public String login() {
        return null;
    }
}
```

TDD kuralları gereği implementasyona geçmeden önce testi çalıştırıp, aslında çalışmaz durumda olduğunu kontrol ediyoruz. Çıkış noktamız her zaman çalışmayan bir testten yola çıkarak implementasyonu oluşturmak ve testi çalışır hale getirmektir.

Spring konfigürasyonu çok basit bir işlem değildir. Spring in bağımlı olduğu diğer kütüphanelerin lib dizinine eklenerek, classpath değişkeni üzerinden erişilebilir hale getirilmeleri gerekir. Bunun ilk örneğini LoginControllerTest test sınıfını çalıştırdığımızda görüyoruz:

```
java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory
```

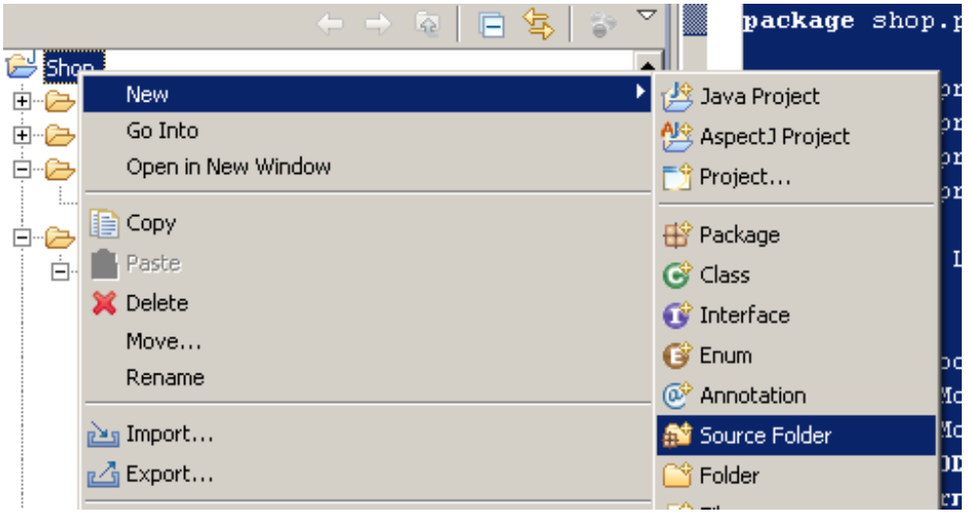
commons-logging-1.0.4.jar kütüphanesinin Build Path a eklenmesi gerekiyor. Bununla beraber bağımlı olduğumuz kütüphane listesi söyledir:

- junit-4.11.jar
- selenium-java-client-driver.jar
- jmock-junit4-2.6.0.jar
- jmock-2.6.0.jar
- commons-logging-1.0.4.jar
- spring-3.2.8.RELEASE.jar

Commons Logging kütüphanesini ekledikten sonra testi tekrar çalıştırıyoruz. Oluşan hata mesajı şu şekildedir:

```
org.springframework.beans.factory.BeanDefinitionStoreException:
        IOException parsing XML document from class path resource
                    [spring-servlet-test.xml];
        nested exception is java.io.FileNotFoundException:
        class path resource [spring-servlet-test.xml]
                    cannot be opened because it does not exist
```

Bu beklediğimiz bir hataydı, çünkü henüz spring-servlet-test.xml isminde bir konfigürasyon dosyası oluşturmadık. spring-servlet-test.xml isminde bir XML dosya oluşturup, projenin properties isimli dizinine yerleştiriyoruz. properties dizini bir kaynak kod dizini olduğu için bu dizin içinde bulunan tüm dosyalar Eclipse tarafından otomatik olarak Java sınıflarının derlendiği web/WEB-INF/classes dizinine kopyalanır. properties dizinini oluştururken bu özelliğe sahip olmasına dikkat ediyoruz.



Resim 10.21 Source folder tipindeki dizinlerin içerikleri otomatik olarak Java sınıflarının derlendiği dizine kopyalanır

Spring konfigürasyon dosyasını kod 10.6 daki gibi tanımlayarak başlayabiliriz. Spring ve Spring MVC hakkında detaylı bilgiyi kitabın Spring konulu bölümlerinde bulabilirsiniz.

Kod 10.6 spring-servlet-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <context:component-scan base-package="shop" />
```

```
    <mvc:annotation-driven />

    <bean
        class="org.springframework.web.servlet.view.
                                InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Kod 10.5 e yer alan LoginController sınıfı @Controller anotasyonunu taşımaktatadır. Tüm Spring MVC uygulamasının anotasyon bazlı olması kararını vermiş oluyoruz. Anotasyonları aktif hale getirmek için kod 10.6 da yer alan Spring konfigürasyon dosyasında mvc:annotation-driven elementini kullanıyoruz. InternalResourceViewResolver sınıfını kullanarak JSP sayfalarının lokasyonunu tayin etmiş oluyoruz.

Bu işlemlerin ardından kod 10.4 de yer alan testLoginUsernameAndPasswordEmpty isimli testimizi tekrar koşturuyoruz. Benim aldığım hata şu şekilde:

```
java.lang.AssertionError: No BindingResult for attribute: loginForm
at org.springframework.test.util.AssertionErrors.fail(
    AssertionErrors.java:39)
```

Kod 10.4 de yer alan testimizde uygulamadan olan beklentimizi

```
.andExpect(model().attributeHasFieldErrors("loginForm", "email"))
.andExpect( model().attributeHasFieldErrors("loginForm", "password"));
```

şeklinde ifade etmiştik. Burada beklentimiz kullanıcının HTML formda yer alan email ve password alanlarını boş bırakarak login butonuna tıklaması durumunda uygulamanın bu alanlar için gerekli hataları oluşturması yönündedir. Uygulamanın bu hataları oluşturabilmesi için loginForm isminde kullanıcının girmiş oldugu verileri bünyesinde taşıyan bir nesnenin var olması gerekiyor. O nesneyi controller bünyesinde şu şekilde oluşturabiliriz:

```
Kod 10.6.1 - RentalController

private static final String LOGIN_VIEW = "login";
private static final String LOGIN_FORM = "loginForm";
```

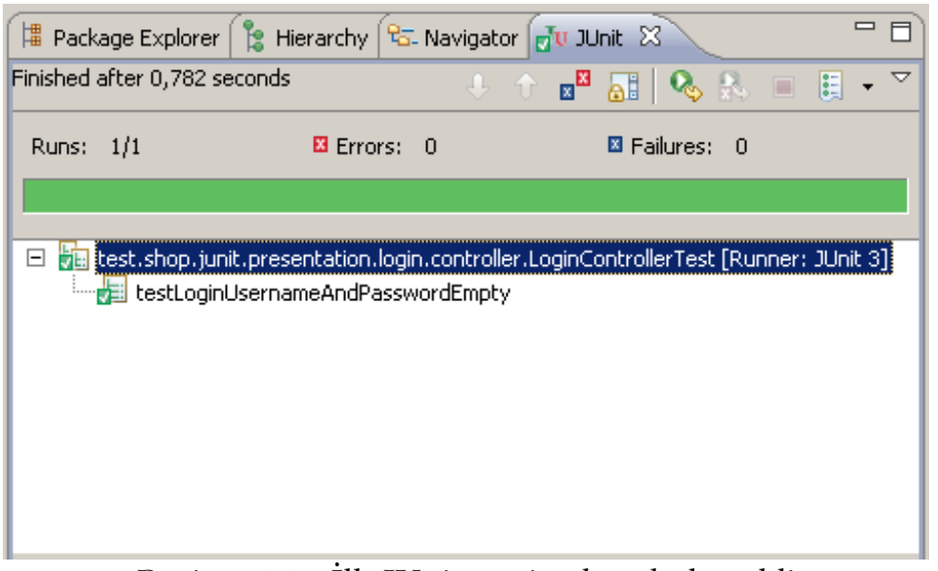
```
@RequestMapping(method = RequestMethod.POST)
public String login(
        final ModelMap model,
        @ModelAttribute(LOGIN_FORM) @Valid final Customer customer,
        final BindingResult result) throws IOException {
    if (result.hasErrors()) {
        return LOGIN_VIEW;
    } else {
        this.processLogin(customer);
        return "redirect:home";
    }
}
```

Spring böyle bir konfigürasyonda HttpServletRequest aracılığı ile kendisine ulaşan verilerden yola çıkarak Customer veri tipinde bir nesne oluşturabilmekte ve bu nesnenin login() metodunda kullanılmasını sağlayabilmektedir. Kod 10.6.1 de yer alan login() metodu ayrıca @Valid anotasyonu yardımı ile otomatik veri validasyonu işlemi yapılmasını sağlamaktadır. Bu amaçla Customer sınıfını kod 10.6.2 deki şekilde yapılandırmamız gerekiyor.

```
 Kod 10.6.2 - Customer

package shop.domain;

public class Customer
{
    @Size(min=1)
    private String email;

    @Size(min=1)
    private String password;
}
```

email ve password isimli değişkenler için @Size(min=1) anotasyonu yardımı ile otomatik veri uzunluğu kontrolü yapılmasını saglayabiliriz. min=1 kullanıcı login butonuna tıkladığında email ve password değişkenleri için bir değer ataması yapılması gerektiği anlamına gelmektedir. Eğer kullanıcı form üzerinde email ve şifre alanlarını boş bırakıp login butonuna tıkladı ise, otomatik validasyon email ve password degişkenlerine değer atanmadığını tespit edecek ve bu kullanıcıya bir hata olarak yansıyacaktır.

Bu implementasyonun ardından testimiz çalışır hale geliyor.



Resim 10.22 İlk JUnit testi çalışır hale geldi

İkinci testle devam edelim:

2. Birim Testi

Kullanıcı login sayfasına gider. Email adresini girer ve şifre alanını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen şifrenizi giriniz!" hata mesajı gösterilir.

Kod 10.9 LoginControllerTest.java - 2. Birim Testi implementasyonu

```
@Test
public void testLoginUsernameSuppliedPasswortEmpty() throws Exception{

    this.mockMvc
    .perform(post("/login")
    .contentType(MediaType.APPLICATION_FORM_URLENCODED)
    .param("email", "test_email")
    .param("password", ""))
    .andExpect( model().attributeHasFieldErrors("loginForm", "password"))
    .andExpect(model().attributeErrorCount("loginForm", 1));
}
```

Kod 10.9 da yer alan test metodu ikinci birim testi için oluşturduğumuz metottur. Yapı itibariyle birinci test ile benzerlik taşıyor. Bu test bünyesinde de beklentimiz hata adedinin bir olması şeklindedir, çünkü email alanı için test_email şeklinde bir değer girilmiştir. Password alanı boş bırakıldığı için "Lütfen şifrenizi giriniz!" hatası oluşmuştur.

Bu hatanın nereden geldiğini merak ediyorsanız, açıklık getireyim. Hata

mesajları için properties dizininde messages.properties ismini taşıyan bir dosya tanımladım. Bu dosyanın içeriği şu şekildedir:

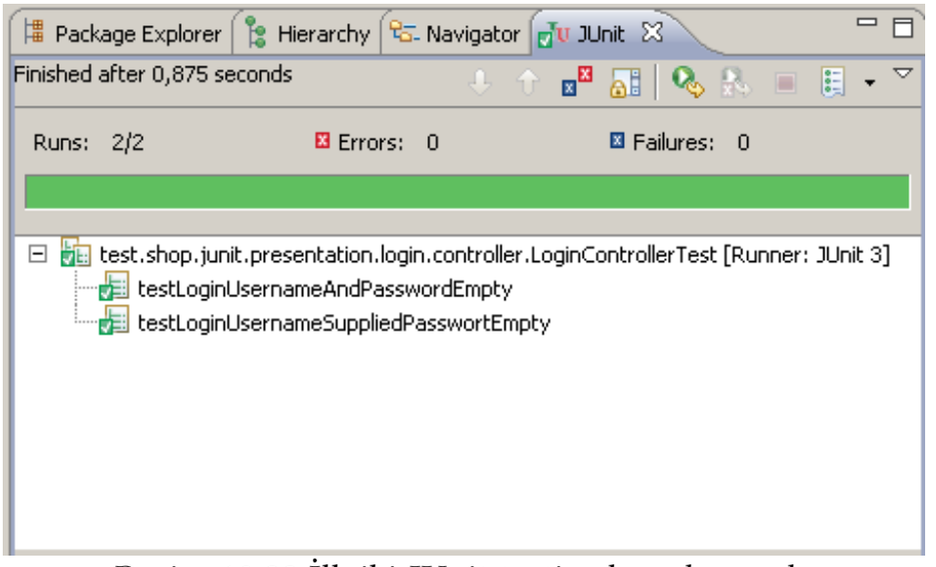
```
Kod 10.9.1 - messages.properties

Size.loginForm.email = Lütfen e-posta adresinizi giriniz!
Size.loginForm.password = Lütfen sifrenizi giriniz!
```

Email ya da password alanı boş bırakıldığında kod 10.6.1 yer alan login() metodunda kullanılan @Valid anotasyonu Customer sınıfı bünyesinde yer alan email ve password değişkenlerine atanan değerlerin kontrol edilmesini tetikler. Eğer bu alanlara bir değer atanmadı ise, Spring kullanılan anotasyonun ismi, form ismi ve alan isminden oluşan bir hata anahtarı oluşturur. Bu anahtarı kullanarak messages.properties dosyasında kullanıcıya gösterilen hata mesajlarını tanımlayabiliriz. Eğer kullanıcı email alanına veri girişi yapmadı ise, otomatik veri validasyonu sonucunda Size.loginForm.email isimli hata anahtarı oluşur. Aynı şey password alanı için de geçerlidir.

Spring MVC uygulamasının messages.properties dosyasını değerlendirebilmesi için konfigürasyon dosyasına aşağidaki eklentiyi yapmamız gerekmektedir:

```
Kod 10.9.1 - spring-servlet-test.xml

<bean class="org.springframework.context.support.
            ResourceBundleMessageSource" id="messageSource">
    <property name="basename" value="messages" />
</bean>
```

Bu değişiklikler ardından testimizi çalıştırıyoruz. Netice resim 10.23 de yer almaktadır.



Resim 10.23 İlk iki JUnit testi çalışır durumda

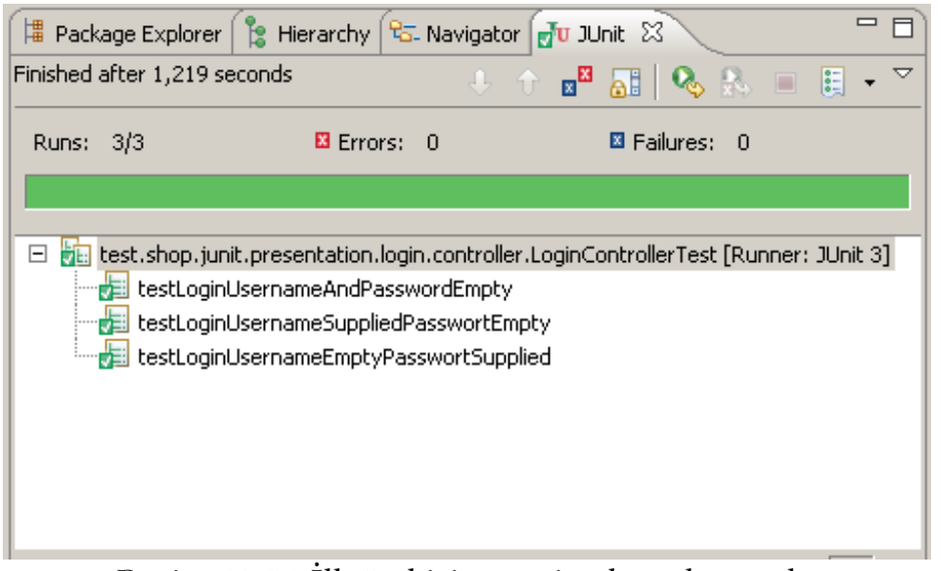
Bir sonraki teste geçebiliriz:

3. Birim Testi

Kullanıcı login sayfasına gider. Email adres alanını boş bırakarak, şifresini girer ve login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi giriniz!" hata mesajı gösterilir.

Kod 10.12 LoginControllerTest.java - 3. Birim testi

```
@Test
public void testLoginUsernameEmptyPasswortSupplied()throws Exception {
    this.mockMvc
    .perform(post("/login")
    .contentType(MediaType.APPLICATION_FORM_URLENCODED)
    .param("email", "")
    .param("password", "test_password"))
    .andExpect( model().attributeHasFieldErrors("loginForm", "email"))
    .andExpect(model().attributeErrorCount("loginForm", 1));
}
```



Resim 10.24 İlk üç birim testi çalışır durumda

Bir sonraki teste geçebiliriz:

4. Birim Testi

Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi ve şifre doğrudur. Login işlemi gerçekleşir. Üye hoş geldiniz sayfasına yönlendirilir.

Dördüncü test için mock ya da stub nesnelerden faydalanmamız gerekiyor, çünkü LoginController sınıfı login işlemini gerçekleştirmek için LoginManager sınıfından yardım almak zorunda. LoginManager sınıfı işletme (business) katmanında bulunan bir interface sınıftır. Değişik implementasyonlar sunabilmek ve gösterim katmanı ile işletme katmanı arasındaki bağı esnek tutabilmek için LoginManager sınıfını bir interface sınıf olarak tanımlıyoruz.

LoginManager interface sınıfını şu şekilde modelliyoruz:

Kod 10.14 LoginManager.java

```
package shop.business.login;

public interface LoginManager
{
    LoginResult login(String email,
            String password);
}
```

Dördüncü birim testini implemente edebilmek için LoginManager sınıfına ihtiyacımız var. Ama kullanabilmek için LoginManager sınıfını implemente

etmek zorunda değiliz. Bir stub implementasyon oluşturarak, LoginManager sınıfını simüle edeceğiz. Paket isminden de anlaşıldığı gibi bu sınıf işletme katmanındadır.

Önce test metodunu oluşturalım:

Kod 10.15 LoginControllerTest.java - dördüncü test metodu

```
@Autowired
ApplicationContext ctx;

@Test
public void testLoginSuccessfull() throws Exception{

    LoginController controller = ctx.getBean(
            LoginController.class);
    LoginManager manager = new LoginManagerDummyImpl(
            StatusCodes.LOGIN_SUCCESSFULL.getValue());
    controller.setService(manager);

    this.mockMvc
        .perform(post("/login")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("email", "test_username")
        .param("password", "test_password"))
        .andExpect(model().attributeErrorCount("loginForm", 0))
        .andExpect(forwardedUrl("/WEB-INF/jsp/home.jsp"));
}
```

Bir Spring nesnesi (bean) olan LoginController nesnesini LoginControllerTest sınıfına Spring tarafından enjekte edilen ctx nesnesinden edinebiliriz. Daha sonra LoginManagerDummyImpl sınıfından bir stub nesne oluşturarak, bu nesneyi controller sınıfına enjekte edebiliriz. LoginManager sınıfı için oluşturduğum stub implementasyonu kod 10.15.1 de yer almaktadır.

Kod 10.15.1 LoginManagerDummyImpl

```
package shop.business.login;

import org.springframework.stereotype.Component;

@Component
public class LoginManagerDummyImpl implements LoginManager {

    private int state;
```

```
    public LoginManagerDummyImpl() {
    }

    public LoginManagerDummyImpl(int state) {
        this.state = state;
    }

    public LoginResult login(String email, String password) {
        LoginResult result = new LoginResult();
        result.setStatus(this.state);
        return result;
    }
}
```

testLoginSuccessfull() metodu bünyesindeki beklentimiz email ve password değişkenlerine bir değer atandığında, bu değerlerin geçerli olması durumunda login işleminin başarıyla yapılmış olması yönündedir. Bunu forwardedUrl() metodunu kullanarak ifade ediyoruz. Bu test ardından LoginController sınıfında yer alan login() metodunu aşağıdaki şekilde genişletiyoruz.

Kod 10.16 LoginManagerDummyImpl

```
@RequestMapping(method = RequestMethod.POST)
public String login(
        final ModelMap model,
        @ModelAttribute(LOGIN_FORM) @Valid final Customer customer,
        final BindingResult result) throws IOException {
    if (result.hasErrors()) {
        return LOGIN_VIEW;
    } else {
        LoginResult loginResult = this.processLogin(customer);

        if(loginResult.getStatus() ==
            StatusCodes.LOGIN_SUCCESSFULL.getValue()){
            return "home";
        }
        return LOGIN_VIEW;
    }
}

private LoginResult processLogin(
        final Customer customer) {
    return this.manager.login(customer.getEmail(),
            customer.getPassword());
}
```

Kod 10.15 de yer alan LoginManagerDummyImpl stub nesnesini

StatusCodes.LOGIN_SUCCESSFULL değerini ihtiva edecek şekilde yapılandırdık. Bu nesneyi LoginController sınıfına enjekte ettik. login() metodu login işleminden sonra edindiği LoginResult nesnesi bünyesinde StatusCodes.LOGIN_SUCCESSFULL değerini bulduğu için "home" isimli sayfaya yönlendirme yapmaktadır. Bu login işleminin başarıyla tamamlandığının göstergesidir. Bu implementasyonla birlikte kod 10.15 de yer alan test çalışır hale gelmiştir. İmplemente ettiğim LoginResult sınıfı şu yapıdadır:

```
Kod 10.17   LoginResult.java

package shop.business.login;

import shop.domain.Customer;

public class LoginResult
{
    private int status;

    private Customer customer;

    public int getStatus()
      {
        return status;
    }

    public void setStatus(int status)
    {
        this.status = status;
    }

    public Customer getCustomer()
    {
        return customer;
    }

    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }
}
```

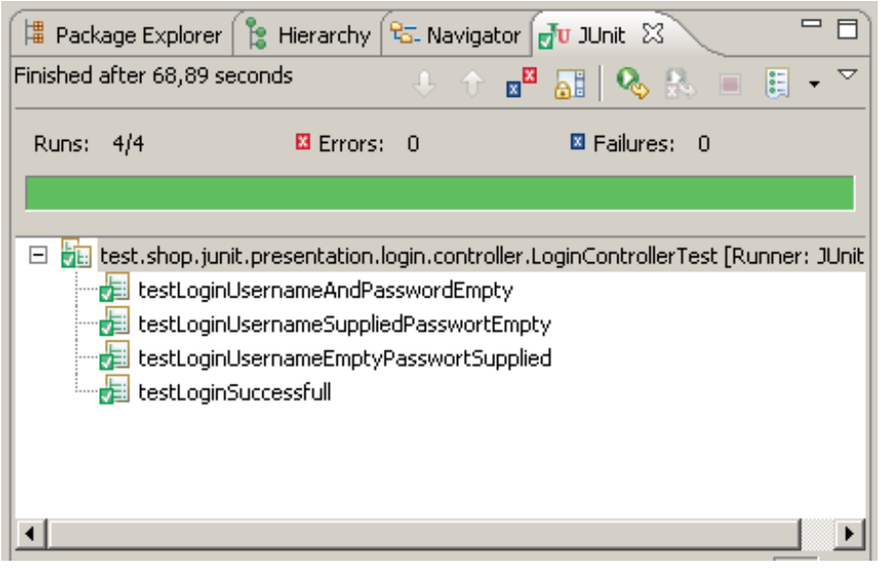
Çalıştığım çok katmanlı projelerde katmanlar arası bilgi alışverişini LoginResult gibi basit sınıflarla gerçekleştiriyorum. LoginResult sınıfı int tipinde ve status ismini taşıyan bir değişkene sahiptir. Değişik tipte statü kodları tanımlayarak,

işletme katmanında olup, bitenleri gösterme katmanına bildirebiliriz. Çoğu projede örneğin bir hesap (user account) veri tabanında bulunamadığı zaman AccountNotFoundException gibi Exception sınıfları kullanılarak, kullanıcı katman bir Exception aracılığıyla uyarılır. Aslında bu gereksizdir, çünkü Exception sınıfları aracılığıyla bilgi alışverişinde bulunmak hem zordur, hem de sisteme pahalıya mal olur. Eğer Exception sınıfları aracılığıyla başka bir katmana bilgi veriyorsak, bu katmanın öncelikle try, catch ile bu Exception nesnelerini yakalayıp, değerlendirmesi gerekir. Ayrıca Exception nesnelerinin mutlaka metot parametre listesinde bulunması ve try, catch ile işlenmiyorsa, throws ile bir üst katmana gönderilmesi gerekmektedir. Bu yazılımı çok karmaşık bir hale getirebilir. Birçok programcı tarafından henüz Java Exception handling mekanizmaları iyi anlaşılmadığı için katmanlar arası oluşturulan Exception nesneler bir try,catch blogunda işlem görmeden kaybolmakta ve sistemin yanlış davranışlar sergilemesine sebep vermektedir. Bu sebepten dolayı ben katmanlar arası statü kodları ile çalışılmasını daha uygun görüyorum. Örneğin bir hesap veri tabanında bulunamadı ise, bir Exception oluşturmak yerine kod 10.18 de tanımlanan Enum sınıfında bulunan statü kodları kullanılabilir.

```
Kod 10.18   StatusCodes.java

package shop.business.login;

public enum StatusCodes
{
    LOGIN_SUCCESSFULL(1),
    ACCOUNT_NOT_FOUND(2);

    private int value;

    private StatusCodes(int i)
    {
        this.value = i;
    }

    public int getValue()
    {
        return value;
    }

    public void setValue(int value)
    {
        this.value = value;
    }
```

```
}
```

İşletme katmanındaki LoginManager sınıfı eğer email adresi ve şifre doğru ise LOGIN_SUCCESSFULL statü koduyla durumu, yani login işlemininden olumlu sonuç alındığını gösterim katmanındaki LoginController sınıfına bildirecektir. LoginManager sınıfı bu statü kodunun taşınması için LoginResult sınıfını kullanır.



Resim 10.26 4. birim testi çalışır durumda

Bir sonraki teste geçebiliriz.

5. Birim Birim Testi

Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçersizdir. Kullanıcıya "Email adresiniz geçersizdir, lütfen tekrar ediniz!" hata mesajı gösterilir.

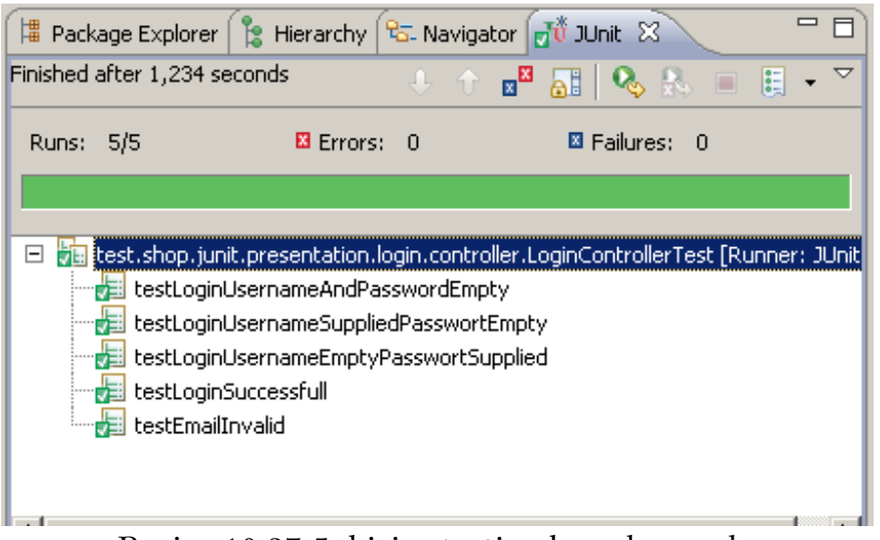
Kod 10.21 LoginControllerTest.java - 5. birim test metodu

```
@Test
public void testEmailInvalid() throws Exception{
    LoginController controller = ctx.getBean(
            LoginController.class);
    LoginManager manager = new LoginManagerDummyImpl(
            StatusCodes.EMAIL_INVALID.getValue());
    controller.setService(manager);

    this.mockMvc
        .perform(post("/login")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("email", "test_username")
```

```
        .param("password", "test_password"))
        .andExpect( model().attributeHasFieldErrors("loginForm",
                    "email"))
        .andExpect(model().attributeErrorCount("loginForm", 1));
}
```

Kod 10.22 LoginController.java

```
@RequestMapping(method = RequestMethod.POST)
public String login(
        final ModelMap model,
        @ModelAttribute(LOGIN_FORM) @Valid final Customer customer,
        final BindingResult result) throws IOException {
    if (result.hasErrors()) {
        return LOGIN_VIEW;
    } else {
        LoginResult loginResult = this.processLogin(customer);

        if(loginResult.getStatus() == StatusCodes.
                            LOGIN_SUCCESSFULL.getValue()){
            return "home";
        }
        else if(loginResult.getStatus() == StatusCodes.
                                EMAIL_INVALID.getValue()){
            result.rejectValue("email", "EMAIL_INVALID");
            return LOGIN_VIEW;
        }
        return LOGIN_VIEW;
    }
}
```



Resim 10.27 5. birim testi çalışır durumda

Bir sonraki teste geçebiliriz.

6. Birim Birim Testi

Kullanıcı login sayfasına gider. Email adresi ve şifreni girer ve login butonuna tıklar. Email adresi geçerli olmasına rağmen şifre hatalıdır. Kullanıcıya "Şifre hatalı, lütfen tekrar deneyiniz!" hata mesajı gösterilir.

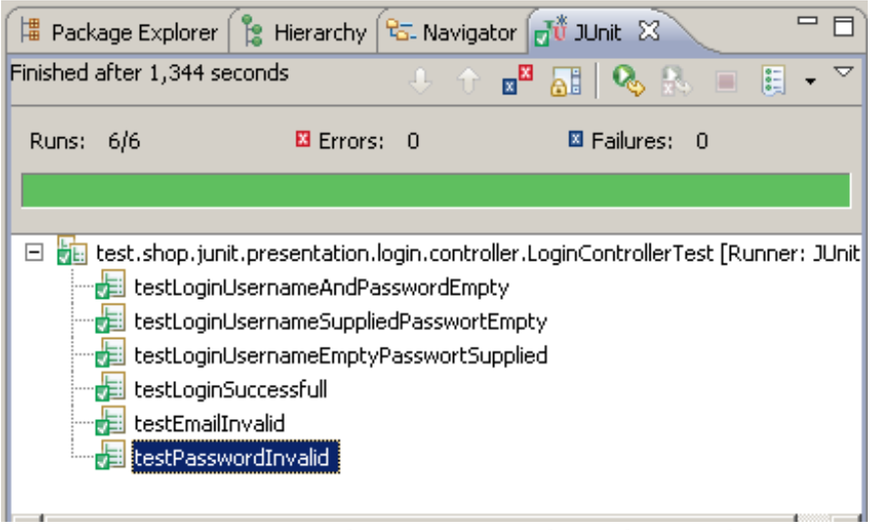
Kod 10.23 LoginControllerTest.java - 6. JUnit Test metodu

```
@Test
public void testPasswordInvalid() throws Exception{
    LoginController controller = ctx.getBean(
            LoginController.class);
    LoginManager manager = new LoginManagerDummyImpl(
            StatusCodes.PASSWORD_INVALID.getValue());
    controller.setService(manager);

    this.mockMvc
        .perform(post("/login")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("email", "test_username")
        .param("password", "test_password"))
        .andExpect( model().attributeHasFieldErrors("loginForm",
                                "password"))
        .andExpect(model().attributeErrorCount("loginForm", 1));
}
```

Kod 10.24 LoginController.java

```
@RequestMapping(method = RequestMethod.POST)
public String login(
        final ModelMap model,
        @ModelAttribute(LOGIN_FORM) @Valid final Customer customer,
        final BindingResult result) throws IOException {
    if (result.hasErrors()) {
        return LOGIN_VIEW;
    } else {
        LoginResult loginResult = this.processLogin(customer);

        if(loginResult.getStatus() == StatusCodes.
                            LOGIN_SUCCESSFULL.getValue()){
            return "home";
        }
        else if(loginResult.getStatus() ==
                        StatusCodes.EMAIL_INVALID.getValue()){
            result.rejectValue("email", "EMAIL_INVALID");
            return LOGIN_VIEW;
```

```
        }
        else if(loginResult.getStatus() == StatusCodes.
                            PASSWORD_INVALID.getValue()){
            result.rejectValue("password", "PASSWORD_INVALID");
            return LOGIN_VIEW;
        }
        return LOGIN_VIEW;
    }
}
```



Resim 10.28 6. birim testi çalışır durumda

Tanımlamış olduğumuz altı birim testini ve testler için gerekli sınıfları implemente ettik. Tüm testler çalışır durumda. Tekrar oluşturduğumuz bütün sınıfların kodunu gözden geçirelim.

Kod 10.24 LoginControllerTest.java - Son hali

```
package test.shop.unit.presentation.login.controller;

import static org.springframework.test.web.servlet.request.
                MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.
                MockMvcResultMatchers.forwardedUrl;
import static org.springframework.test.web.servlet.result.
                MockMvcResultMatchers.model;
import org.jmock.Mockery;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
```

```
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import shop.business.login.LoginManager;
import shop.business.login.LoginManagerDummyImpl;
import shop.business.login.StatusCodes;
import shop.presentation.login.controller.LoginController;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath:spring-servlet-test.xml" })
public class LoginControllerTest {

    Mockery context = new Mockery();

    @Autowired
    private WebApplicationContext wac;

    @Autowired
    ApplicationContext ctx;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(
                this.wac).build();
    }

    @Test
    public void testLoginUsernameAndPasswordEmpty()
            throws Exception {

        this.mockMvc
            .perform(post("/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("email", "")
            .param("password", ""))
            .andExpect(model().attributeHasFieldErrors("loginForm",
                        "email"))
            .andExpect( model().attributeHasFieldErrors("loginForm",
                        "password"))
            .andExpect(model().attributeErrorCount("loginForm", 2));
    }

    @Test
    public void testLoginUsernameSuppliedPasswortEmpty() throws
```

```
                                        Exception{

    this.mockMvc
    .perform(post("/login")
    .contentType(MediaType.APPLICATION_FORM_URLENCODED)
    .param("email", "test_email")
    .param("password", ""))
    .andExpect( model().attributeHasFieldErrors("loginForm",
                                        "password"))
    .andExpect(model().attributeErrorCount("loginForm", 1));
}


@Test
public void testLoginUsernameEmptyPasswortSupplied()throws
                Exception {
    this.mockMvc
        .perform(post("/login")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("email", "")
        .param("password", "test_password"))
        .andExpect( model().attributeHasFieldErrors("loginForm",
                    "email"))
        .andExpect(model().attributeErrorCount("loginForm", 1));
}

@Test
public void testLoginSuccessfull() throws Exception{

    LoginController controller = ctx.getBean(
            LoginController.class);
    LoginManager manager = new LoginManagerDummyImpl(
            StatusCodes.LOGIN_SUCCESSFULL.getValue());
    controller.setService(manager);

    this.mockMvc
        .perform(post("/login")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("email", "test_username")
        .param("password", "test_password"))
        .andExpect(model().attributeErrorCount("loginForm", 0))
        .andExpect(forwardedUrl("/WEB-INF/jsp/home.jsp"));
}


@Test
public void testEmailInvalid() throws Exception{
    LoginController controller = ctx.getBean(
            LoginController.class);
```

```
        LoginManager manager = new LoginManagerDummyImpl(
                StatusCodes.EMAIL_INVALID.getValue());
        controller.setService(manager);

        this.mockMvc
            .perform(post("/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("email", "test_username")
            .param("password", "test_password"))
            .andExpect( model().attributeHasFieldErrors("loginForm",
                            "email"))
            .andExpect(model().attributeErrorCount("loginForm", 1));
    }

    @Test
    public void testPasswordInvalid() throws Exception{
        LoginController controller = ctx.getBean(
                LoginController.class);
        LoginManager manager = new LoginManagerDummyImpl(
                StatusCodes.PASSWORD_INVALID.getValue());
        controller.setService(manager);

        this.mockMvc
            .perform(post("/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("email", "test_username")
            .param("password", "test_password"))
            .andExpect( model().attributeHasFieldErrors("loginForm",
                        "password"))
            .andExpect(model().attributeErrorCount("loginForm", 1));
    }
}
```

Kod 10.25 LoginController.java - Son hali

```
package shop.presentation.login.controller;

import java.io.IOException;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import shop.business.login.LoginManager;
import shop.business.login.LoginResult;
import shop.business.login.StatusCodes;
```

```
import shop.domain.Customer;

@Controller
@RequestMapping("/login")
public class LoginController {

    private static final String LOGIN_VIEW = "login";
    private static final String LOGIN_FORM = "loginForm";

    @Autowired
    private LoginManager manager;

    @RequestMapping(method = RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(LOGIN_FORM, new Customer());
        return LOGIN_VIEW;
    }

    @RequestMapping(method = RequestMethod.POST)
    public String login(
            final ModelMap model,
            @ModelAttribute(LOGIN_FORM) @Valid final Customer customer,
            final BindingResult result) throws IOException {
        if (result.hasErrors()) {
            return LOGIN_VIEW;
        } else {
            LoginResult loginResult = this
                    .processLogin(customer);

            if (loginResult.getStatus() ==
                        StatusCodes.LOGIN_SUCCESSFUL.getValue()) {
                return "home";
            } else if (loginResult.getStatus() ==
                            StatusCodes.EMAIL_INVALID
                    .getValue()) {
                result.rejectValue("email", "EMAIL_INVALID");
                return LOGIN_VIEW;
            } else if (loginResult.getStatus() ==
                            StatusCodes.PASSWORD_INVALID
                    .getValue()) {
                result.rejectValue("password",
                        "PASSWORD_INVALID");
                return LOGIN_VIEW;
            }
            return LOGIN_VIEW;
        }
    }
```

```
    private LoginResult processLogin(final Customer customer) {
        return this.manager.login(customer.getEmail(),
                customer.getPassword());
    }

    public LoginManager getManager() {
        return this.manager;
    }

    public void setService(final LoginManager manager) {
        this.manager = manager;
    }
}
```

Kod 10.26 StatusCodes.java - Son hali

```
package shop.business.login;

public enum StatusCodes
{
    LOGIN_SUCCESSFULL(1),
    EMAIL_INVALID(2),
    PASSWORD_INVALID(3);

    private int value;

    private StatusCodes(int i)
    {
        this.value = i;
    }

    public int getValue()
    {
        return value;
    }

    public void setValue(int value)
    {
        this.value = value;
    }
}
```

Kod 10.27 LoginManager.java - Son hali

```
package shop.business.login;

public interface LoginManager
```

```
{
    LoginResult login(String email,
            String password);
}
```

Kod 10.28 LoginResult.java - Son hali

```
package shop.business.login;

import shop.domain.Customer;

public class LoginResult
{
    private int status;
    private Customer customer;

    public int getStatus()
    {
        return status;
    }

    public void setStatus(int status)
    {
        this.status = status;
    }

    public Customer getCustomer()
    {
        return customer;
    }

    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }
}
```

Gösterim katmanında yer alan LoginController sınıfı için testleri tamamlamış olduk. LoginControllerTest ismini taşıyan bir test sınıfımız ve bu sınıf bünyesinde implemente ettiğimiz altı test metodu var.

Ant JUnit Entegrasyonu

Oluşturduğumuz testleri Eclipse altında otomatik olarak resim 10.28 de görüldüğü gibi çalıştırabiliyoruz. Testlerin çalıştırılmasını daha otomatize edebilmek ve ilerde sürekli entegrasyonu sağlayabilmek için testlerin Ant

üzerinden çalıştırılmasını sağlamamız gerekiyor. Bu amaçla aşağıda yer alan Ant build.xml dosyasını oluşturuyoruz.

Kod 10.29 Ant - build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Shop" default="run-junit">


    <property file="ant.properties" />


    <path id="compile.classpath">
        <fileset dir="${base.web.lib}">
            <include name="*.jar" />
        </fileset>
        <fileset dir="${base.lib}">
            <include name="*.jar" />
        </fileset>
    </path>



    <target name="compile" depends="clean, copy_properties">
        <javac srcdir="${base.src}"
            destdir="${build.dir.classes}"
            debug="on" verbose="off">
            <classpath>
                <path refid="compile.classpath" />
            </classpath>
        </javac>

        <javac srcdir="${base.src.test}"
            destdir="${build.dir.classes}" debug="on"
            verbose="off">
            <classpath>
                <path refid="compile.classpath" />
            </classpath>
        </javac>
    </target>



    <target name="copy_properties" depends="clean">
        <copy todir="${build.dir.classes}" overwrite="true">
            <fileset dir="${base.properties}">
                <include name="**/*.*" />
            </fileset>
```

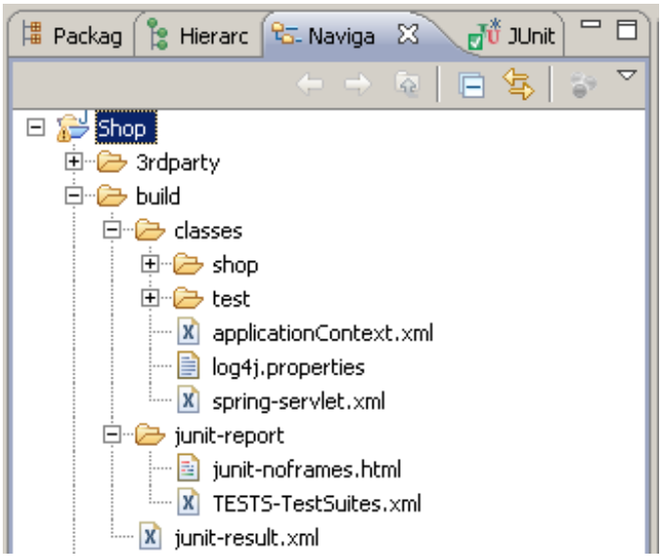
```
        </copy>
    </target>



    <target name="clean" >

        <delete quiet="true" includeemptydirs="true">
            <fileset dir="${build.dir}" />
        </delete>
        <mkdir dir="${build.dir}" />
        <mkdir dir="${build.dir.classes}" />
        <mkdir dir="${junit.report.dir}" />
    </target>



    <target name="run-junit" depends="compile">


        <junit fork="false" failureproperty="tests.failed"
            showoutput="true" printsummary="yes"
            haltonfailure="yes">
            <classpath refid="compile.classpath" />
            <classpath path="${build.dir.classes}" />

            <test name="${junit.testcase.class}"
                haltonfailure="yes" outfile="build/junit-result">
                <formatter type="xml" />
            </test>
        </junit>

        <fail if="tests.failed">
        ******************************************************
        ******************************************************
        JUnit testlerinde hata olustu. Lütfen kontrol et!
        ******************************************************
        ******************************************************
        </fail>

        <junitreport todir="${junit.report.dir}">
            <fileset dir="${build.dir}">
                <include name="junit-result.xml" />
            </fileset>
            <report format="noframes" todir="${junit.report.dir}" />
        </junitreport>
    </target>

</project>
```

build.xml bünyesinde değişik tipte hedefler (target) tanımlıyoruz. Bunlar:

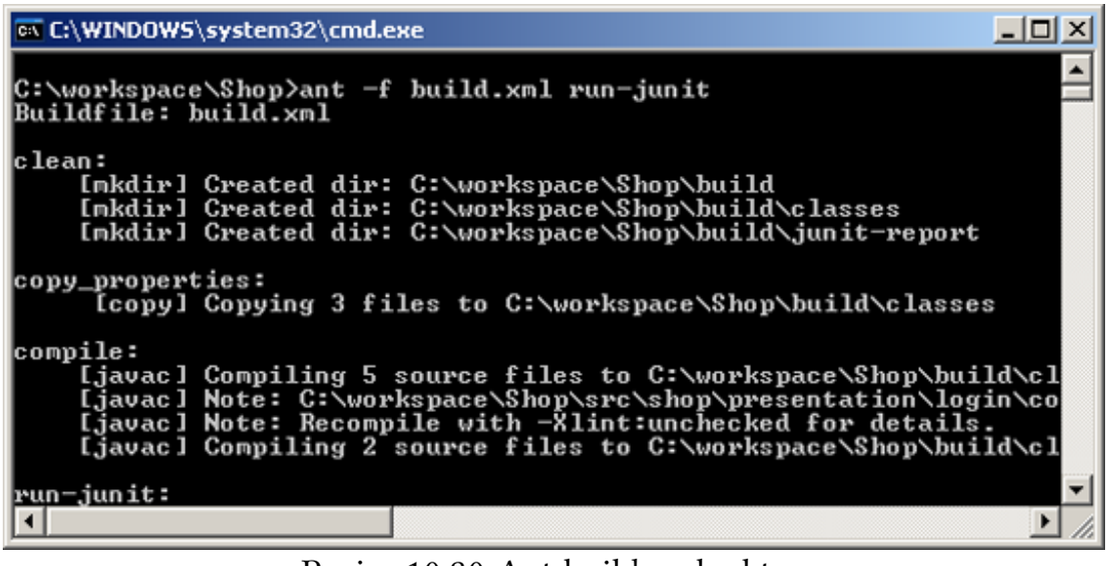
- ***compile*** - src ve src-test dizinlerinde bulunan tüm Java sınıflarını build/classes dizini derler. clean, copy_properties hedeflerine bağımlıdır.
- ***copy_properties*** - properties dizininde bulunun tüm dosyaları build/classes dizinine kopyalar. clean hedefine bağımlıdır.
- ***clean*** - ${build.dir} değişkeni ile build isminde bir dizin oluşturulur. Bu değişken ant.properties dosyasında tanımlanmıştır. Bu hedef bünyesinde build için gerekli dizin yapısı oluşturulur.
- ***run-junit*** - build/classes dizine derlenen LoginControllerTest sınıfı taskı ile çalıştırır. taskı ile HTML rapor oluşturulur. compile hedefine bağımlıdır.

ant.properties dosyasının içeriği kod 10.30 da yer almaktadır.

```
Kod 10.30   ant.properties

#base
base.dir=${basedir}
base.lib=${base.dir}/lib
base.src=${base.dir}/src
base.src.test=${base.dir}/src-test
base.properties=${base.dir}/properties


#build
build.dir=${base.dir}/build
build.dir.classes=${base.dir}/build/classes


#web
base.web=${base.dir}/web
base.web.web-inf=${base.web}/WEB-INF
base.web.lib=${base.web.web-inf}/lib
base.web.web-inf.classes=${base.web.web-inf}/classes

#junit
junit.testcase.class=test.shop.junit.presentation.login.
                controller.LoginControllerTest
junit.report.dir=${base.dir}/build/junit-report
```



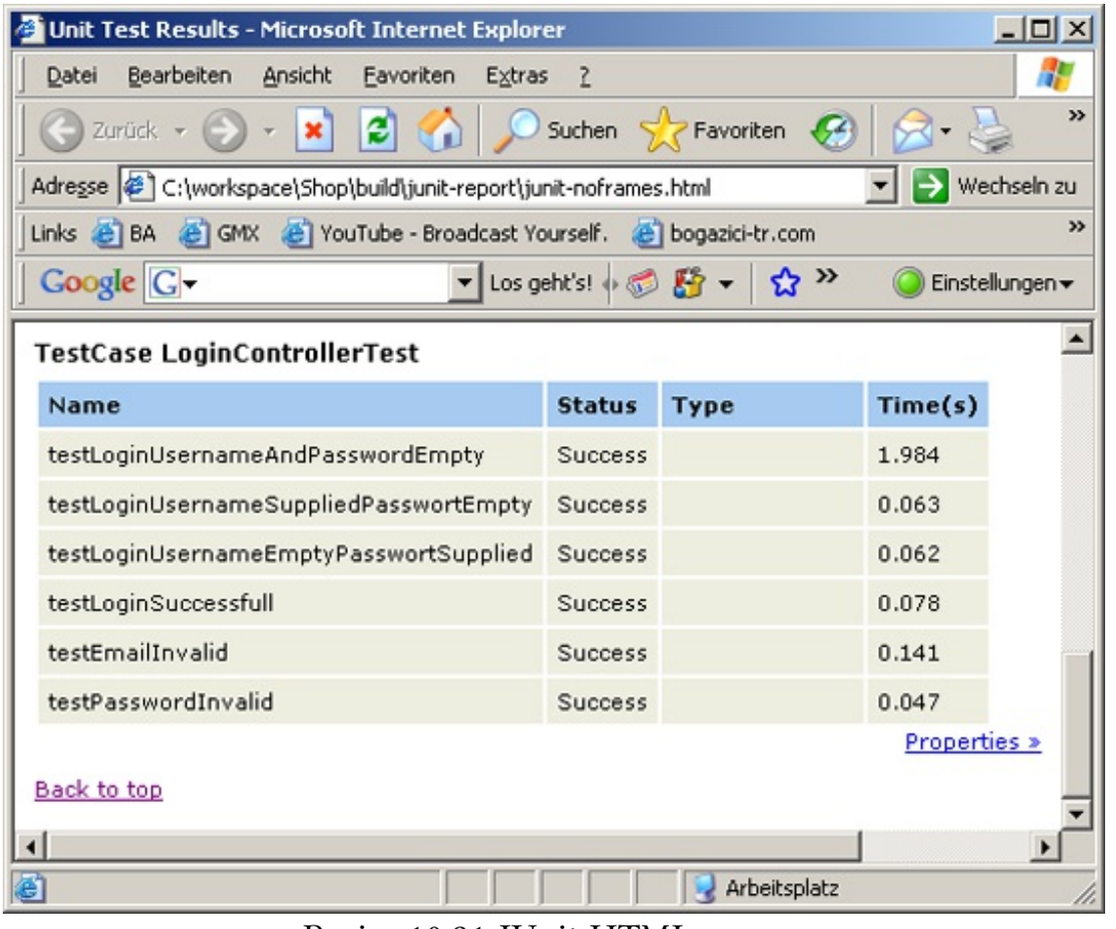
Resim 10.29 build dizin yapısı



Resim 10.30 Ant build.xml çıktısı

Oluşturduğumuz Ant skript ile yapılandırma (derleme, kopyalama) işlemlerini ve birim testlerinin çalıştırılmasını otomatize etmiş olduk. Bu sayede Eclipse platformuna bağımlılığımız ortadan kalkmış oldu. Resim 10.30 da görüldüğü gibi testleri bir console altında çalıştırabiliyoruz.

build.xml de dikkat çeken bir diğer element te junitreport elementidir. Bu task ile JUnit testleri çalıştıktan sonra, test sonuçları baz alınarak HTML türünde rapor hazırlanır. Bunun bir örneğini resim 10.31 de görmekteyiz.



Resim 10.31 JUnit HTML raporu

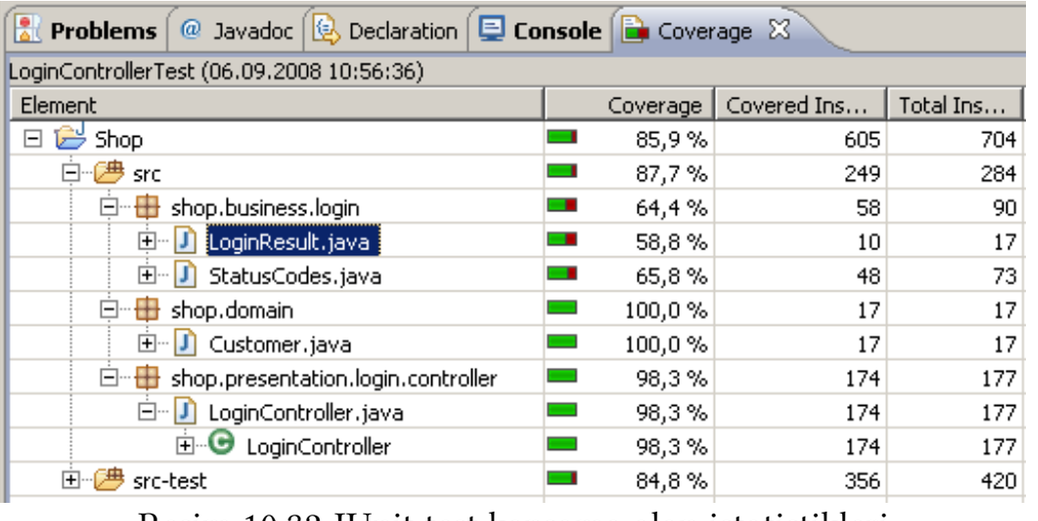
Kısa bir özet yaparak, şimdiye kadar neler yaptığımızı tekrar gözden geçirelim.

1. İmplementasyon için çıkış noktamız kullanıcı hikayelerinin (user story) 1. iterasyon için seçimi oldu. Tablo 10.2 de ilk iterasyon için seçtiğimiz kullanıcı hikayeleri yer almaktadır.
2. Dört iterasyondan oluşan bir sürüm planı hazırladık (tablo 10.1). Her iterasyon için bir haftalık zaman dilimi ayrıldı.
3. Sıfırıncı iterasyonda proje için gerekli çalışma ortamını oluşturduk
4. İki numaralı kullanıcı hikayesini seçerek, Selenium ile ilk onay/kabul testini oluşturduk. Bu ve diğer testler TDD tarzı implemente edildi. Bu testten yola çıkarak, bir tasarım oturumunda implemente etmek istediğimiz tasarımı oluşturduk (resim 10.16).
5. Gösterim katmanını Spring MVC ile implemente etmeye karar verdik. Gösterim katmanını implemente edebilmek için birim test listesi oluşturduk ve bu testleri tek tek oluşturarak, gösterim katmanında bulunan sınıfları implemente ettik.
6. İşletme katmanında bulunan LoginManager sınıfını simüle etmek için bir stub implementasyon oluşturduk.

7. Bir Ant skript (build.xml) ile testleri Ant üzerinden çalıştırılabilir hale getirdik. Her Ant skriptin çalıştırılmasında JUnit test sonuçlarının HTML raporu (resim 10.31) haline getirilmesini sağladık.

İki nolu kullanıcı hikayesini baz alarak başladığımız implementasyon sonucunda login işlemi için gösterim katmanında gerekli tüm sınıfları implemente etmiş olduk. Login işlemini gerçekleştirmek için gösterim katmanı işletme katmanında bulunan LoginManager interface sınıfını kullanmaktadır. Gösterim katmanı için gerekli implementasyonu bitirmeden işletme katmanına geçmek istemediğimiz ve topdown tarzı TDD uyguladığımız için LoginManager sınıfını simüle etmek için stub nesneden faydalandık.

Benim bu noktada merak ettiğim bir konu var! Acaba oluşturduğumuz test metotları ile implemente ettiğimiz sınıfların hangi bölümleri çalıştı? Test kapsama alanı (test coverage) nedir? Bunu tespit etmek için sekizinci bölümde tanıştığımız EclEmma programını kullanabiliriz. Bir Eclipse Plugin olan bu program aracılığıyla testlerin hangi kod alanlarına ulaştığını tespit edebilmektedir.



Resim 10.32 JUnit test kapsama alan istatistikleri

Proje genelinde test kapsama oranı % 85.9 dur. Bunun içine src ve src-test dizinlerinde yer alan tüm sınıflar dahildir. src-test dizininde bulunan sınıflar test sınıfları olduğu için bu dizindeki kapsama alanı bizi ilgilendirmiyor. Sadece src dizinini için tespit edilen test kapsama oranı önemlidir. src dizini için kapsama alanı %87.7 olarak tespit edilmiştir. Bu çok iyi bir orandır. Java projelerinde en fazla %85 ila %90 arası test kapsamı sağlanabilir. Java dilinin yapısal sorunlarından dolayı ve her set() ve get() metodunun test edilmesi gereksiz olduğu için %100 oranına erişmek, bu orana ulaşmak için harcanacak

emekle kıyaslandığında ekonomik değildir.

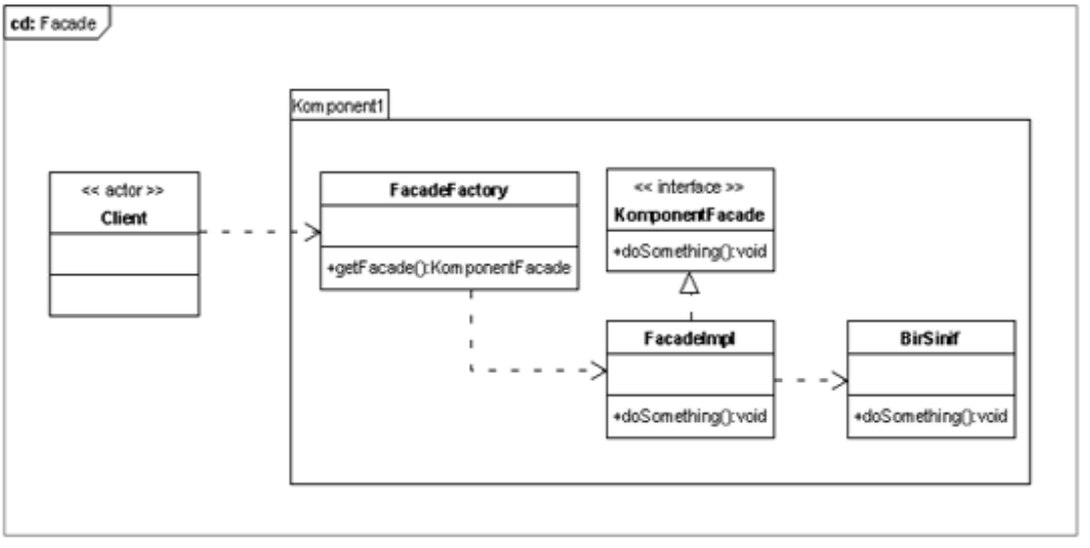
İşletme (Business) Katmanı

İşletme katmanı hazırlanan sistem içinde önemli bir rol oynamaktadır. Gösterim katmanının sadece veri gösterimi ve kullanıcı ile interaksiyonda kullanıldığını gördük. İşletim katmanında bulunan sınıflarda işletme mantığı (business logic) olarak isimlendirilen önemli metotlar yer alırlar. Programlama sürecinin büyük bir bölümünde işletme katmanı içinde yer alan sınıflar oluşturulur.

Bu katman içinde sunulan hizmetlere katmanın bir parçası olan LoginManager üzerinden erişilir. İşletme katmanının sunduğu hizmetlere erişimi bir noktada toplamak ve gösterim ve işletme katmanları arasındaki bağımlılığı azaltmak için Facade tasarım şablonu kullanılır.

Facade (Cephe) Tasarım Şablonu

İşletme ve veri katmanında Facade tasarım şablonunu uygulayacağız. Facade cephe anlamına gelmektedir. Profesyonel yazılım sistemleri birçok komponentin birleşiminden oluşur. Yazılım esnasında bir çok ekip birbirinden bağımsız, sistemin bütününü oluşturan değişik komponentler üzerinde çalışırlar. Bir komponent belirli bir işlevi yerine getirmek için hazırlanmış bir ya da birden fazla Java sınıfından oluşmaktadır.



Resim 10.33 Facade Tasarım Şablonu

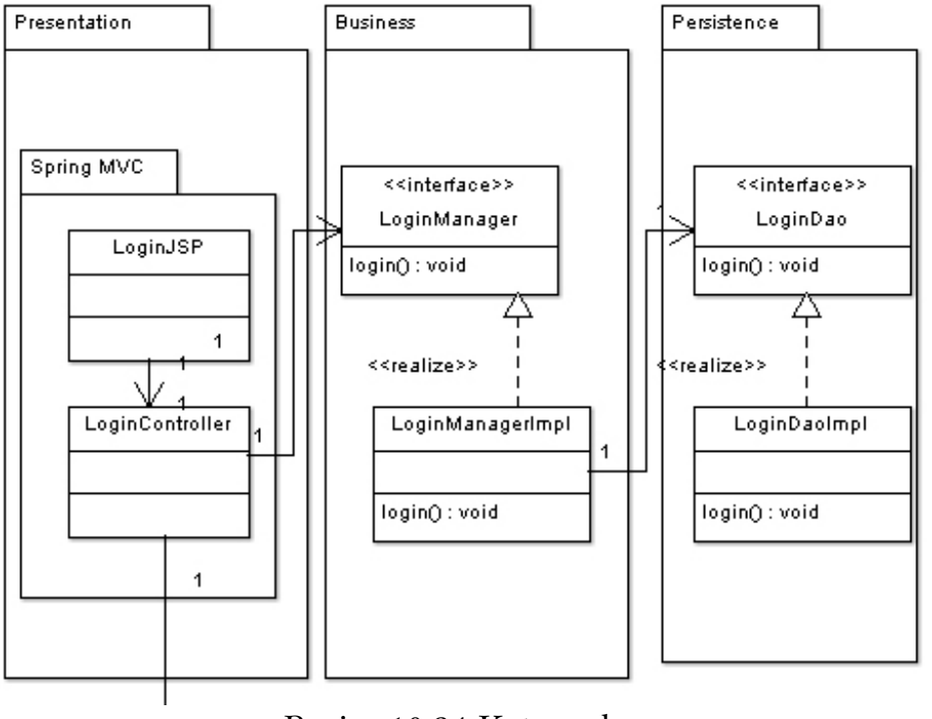
Bir komponentin sunmuş olduğu hizmetten yararlanabilmek için komponentin dış dünya için tanımlamış olduğu giriş/çıkış noktaları (input/output interface) kullanılır. Komponent sadece bu giriş/çıkış noktaları üzerinden dış dünya ile iletişim kurar ve iç dünyasını tamamen gizler. Bu iletişim noktaları genelde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

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

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

İşletme (Business) Katmanı Testleri

İşletme katmanının görevini yerine getirebilmek için veri katmanına katmanına ihtiyaç duymaktadır. Bu katman için testleri oluştururken bunu göz önünde bulundurmamız gerekiyor. Gösterim katmanı implementasyonunda olduğu gibi yine mock nesneler kullanarak, veri katmanına olan bağımlılığı simüle edeceğiz.



Resim 10.34 Katmanlar

İşletme katmanında bulunan LoginManager interface sınıfının implementasyonu için düşündüğümüz testler şu şekildedir.

Kullanıcı hikayesi 2: Kullanıcı isim ve şifreni kullanarak sisteme login yapar.

- ***1. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçersizdir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunamaz. Kullanıcıya "Email adresiniz geçersiz, lütfen tekrar ediniz!" hata mesajı gönderilir.
- ***2. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçersizdir. Kullanıcıya "Şifre hatalı, lütfen, lütfen tekrar deneyiniz" hata mesajı gönderilir.
- ***3. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçerlidir. Login işlemi gerçekleştirilir.

```
Kod 10.31   LoginManagerImplTest.java

package test.shop.unit.business.login;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Before;
```

```
import org.junit.Test;
import shop.business.login.LoginManagerImpl;
import shop.business.login.LoginResult;
import shop.business.login.StatusCodes;
import shop.persistence.login.LoginDao;
import shop.persistence.login.LoginDaoResult;

public class LoginManagerImplTest {

      Mockery context = new Mockery();
    1 private LoginManagerImpl manager = null;
    2 private LoginDao dao;
    3 private LoginDaoResult result = null;

    4 @Before
    5 public void setUp() {
    6   manager = new LoginManagerImpl();
    7   dao = context.mock(LoginDao.class);
    8   manager.setDao(dao);
    9   result = new LoginDaoResult();
    10 }

    11 @Test
    12 public void testEmailInvalid() {
    13 }
}
```

Test etmek istediğimiz sınıf LoginManager sınıfının implementasyonu olacağı için src-test/test/shop/junit/business/login dizininde LoginManagerImplTest ismini taşıyan yeni bir test sınıfı oluşturuyoruz. LoginManager interface sınıfını implemente eden sınıf LoginManagerImpl ismini taşıyacak.

Kod 10.31 da LoginManagerImplTest sınıfı yer almaktadır. Birinci satırda manager isminde bir değişken tanımlıyoruz. Bu değişken test etmek istediğimiz LoginManagerImpl sınıfı veri tipine sahiptir.

Resim 10.34 de görüldüğü gibi LoginManagerImpl sınıfı LoginDao interface sınıfını kullanmaktadır. Bu iki sınıf arasındaki bağlantıyı test edebilmek için LoginManagerImplTest sınıfının yedinci satırında dao isminde bir mock nesne tanımlıyoruz. Yedinci satırda LoginManager sınıfından olan nesnenin setDao() metodu aracılığı ile mock nesneyi manager nesnesine enjekte ediyoruz.

Kod 10.32 LoginManagerImpl.java

```
@Test
public void testEmailInvalid() {
```

```
    try {
        result.setStatus(StatusCodes.EMAIL_INVALID
                .getValue());

        final String email = "test_email";
        final String password = "test_password";

        context.checking(new Expectations() {
            {
                oneOf(dao).findUser(email, password);
                will(returnValue(result));
            }
        });

        LoginResult loginResult = manager.login(email,
                password);

        assertTrue(loginResult.getStatus() == StatusCodes.EMAIL_INVALID
                .getValue());

    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```

İlk birim testini testEmailInValid() isimli test metodunda implemente ediyoruz. LoginDao interface sınıfında findUser() isminde bir metot bulunduğunu farz ediyor ve mock nesneyi oluşturduğumuz LoginDaoResult tipindeki nesneyi geri verecek şekilde programlıyoruz.

manager.login() ile login işlemini gerçekleştiriyoruz. AssertTrue komutuyla login() metodundan elde ettiğimiz LoginResult tipi nesnede bulunan status değişkenin değerini kontrol ediyoruz.

Bu test sınıfı kullanılan bazı sınıfların eksikliğinden dolayı derlenmez durumda. İlk önce gerekli sınıfları en basıt halleriyle oluşturuyoruz.

Kod 10.33 LoginDao.java

```
package shop.persistence.login;

public interface LoginDao
{
    LoginDaoResult findUser(String email, String password);
```

```
}
```

Kod 10.34 LoginDaoResult.java

```
package shop.persistence.login;

import shop.domain.Customer;

public class LoginDaoResult
{
    private int status;
    private Customer customer;

    public int getStatus()
    {
        return status;
    }

    public void setStatus(int status)
    {
        this.status = status;
    }

    public Customer getCustomer()
    {
        return customer;
    }

    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }
}
```

Kod 10.35 LoginManagerImpl.java

```
package shop.business.login;

import shop.persistence.login.LoginDao;

public class LoginManagerImpl implements LoginManager
{

    private LoginDao dao;


    public LoginResult login(String email, String password)
    {
```

```
        return null;
    }


    public LoginDao getDao()
    {
        return dao;
    }

    public void setDao(LoginDao dao)
    {
        this.dao = dao;
    }
}
```

Test etmek istediğimiz LoginManagerImpl sınıfı kod 10.35 de yer almaktadır. Görüldüğü gibi login() metodunu null değerini verecek şekilde oluşturuyoruz. Bunun yanı sıra bu sınıf bünyesinde LoginDao tipinde ve dao isminde bir sınıf değişkeni tanımlıyoruz. LoginManageImpl sınıfı bu değişken üzerinde veri katmanını kullanacaktır.

Bu değişikliklerin ardından testimizi çalıştırıyoruz.

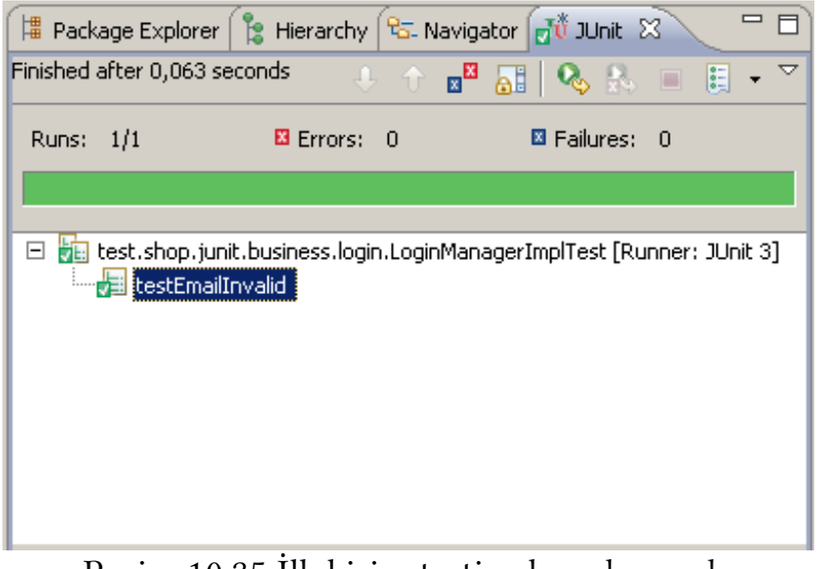
```
java.lang.NullPointerException at
    test.shop.junit.business.login.
        LoginManagerImplTest.testEmailInvalid(
                    LoginManagerImplTest.java:61)
```

Testimiz çalışır durumda değil, çünkü login() metodu null değerini veriyor ve bu durum NullPointerException oluşmasına sebep oluyor. Testin olumlu sonuç verebilmesi için login metodunu kod 10.36 de görüldüğü gibi değiştiriyoruz.

```
Kod 10.36   LoginManagerImpl.java

/**
 * Login metodu
 */
public LoginResult login(String email, String password)
{
    LoginResult result = new LoginResult();
    try
    {
        LoginDaoResult daoResult =
            dao.findUser(email, password);
        copyResult(daoResult, result);
    }
```

```
    catch (Exception e)
    {
        throw new LoginManagerException(e);
    }
    return result;
}

private void copyResult(LoginDaoResult daoResult,
        LoginResult result)
{
    if(daoResult.getCustomer() != null)
    {
        result.setCustomer(daoResult.getCustomer());
    }
    result.setStatus(daoResult.getStatus());
}
```

login() metodunda oluşabilecek tüm hataları catch bloğunda yakalıyor ve oluşan Exception nesnesini LoginManagerException nesnesine dönüştürerek, bir üst katmana gönderiyoruz. LoginManagerException RuntimeException tipinde (unchecked Exception) bir Exception olduğu için metot imzasında throws ile tanımlanması gerekmiyor. dao değişkeninden edindiğimiz verileri copyResult() metodunda LoginResult sınıfından oluşturduğumuz nesneye kopyalıyoruz.



Resim 10.35 İlk birim testi çalışır durumda

İkinci teste geçebiliriz:

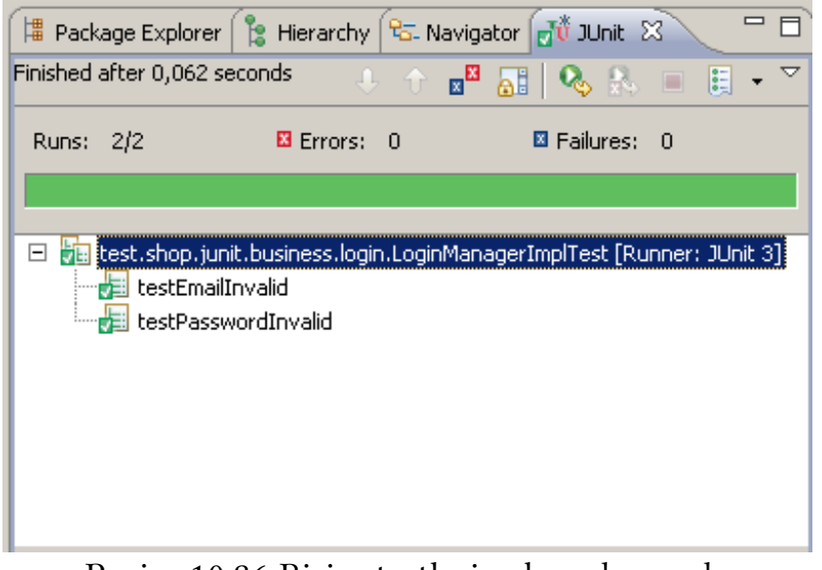
2. Birim Testi

Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçersizdir.

Kullanıcıya “Şifre hatalı, lütfen tekrar deneyiniz” hata mesajı gönderilir.

Kod 10.37 LoginManagerImplTest.java

```
@Test
public void testPasswordInvalid() {
    try {
        result.setStatus(StatusCodes.PASSWORD_INVALID
                .getValue());
        final String email = "test_email";
        final String password = "test_password";

        context.checking(new Expectations() {
            {
                oneOf(dao).findUser(email, password);
                will(returnValue(result));
            }
        });

        LoginResult loginResult = manager.login(email,
                password);

        assertTrue(loginResult.getStatus() ==
                        StatusCodes.PASSWORD_INVALID
                .getValue());

    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```



Resim 10.36 Birim testleri çalışır durumda

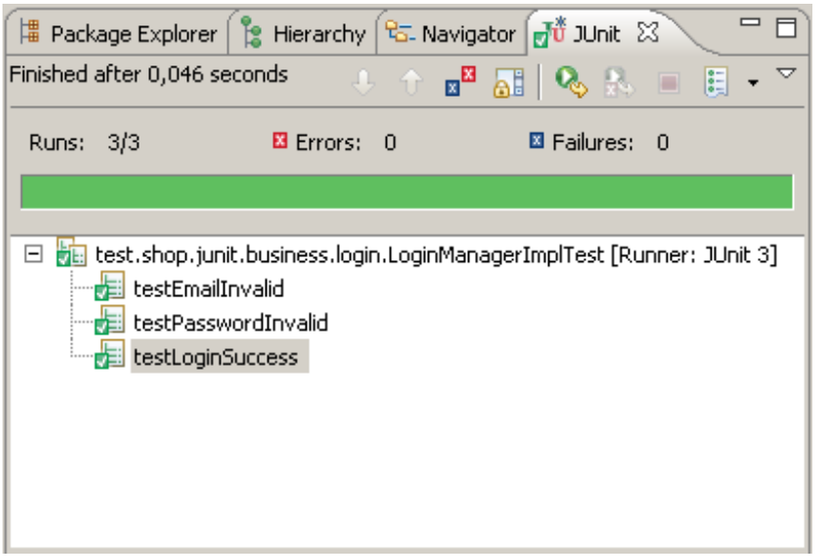
Üçüncü teste geçebiliriz:

3. Birim Testi

Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçerlidir. Login işlemi gerçekleştirilir.

Kod 10.38 LoginManagerImplTest.java

```
@Test
public void testLoginSuccess() {
    try {
        result.setStatus(StatusCodes.LOGIN_SUCCESSFULL
                .getValue());

        final String email = "test_email";
        final String password = "test_password";

        context.checking(new Expectations() {
            {
                oneOf(dao).findUser(email, password);
                will(returnValue(result));
            }
        });

        LoginResult loginResult = manager.login(email,
                password);

        assertTrue(loginResult.getStatus() ==
                        StatusCodes.LOGIN_SUCCESSFULL
                .getValue());
    } catch (Exception e) {
        e.printStackTrace();
        fail();

    }
}
```



Resim 10.37 Birim testleri çalışır durumda

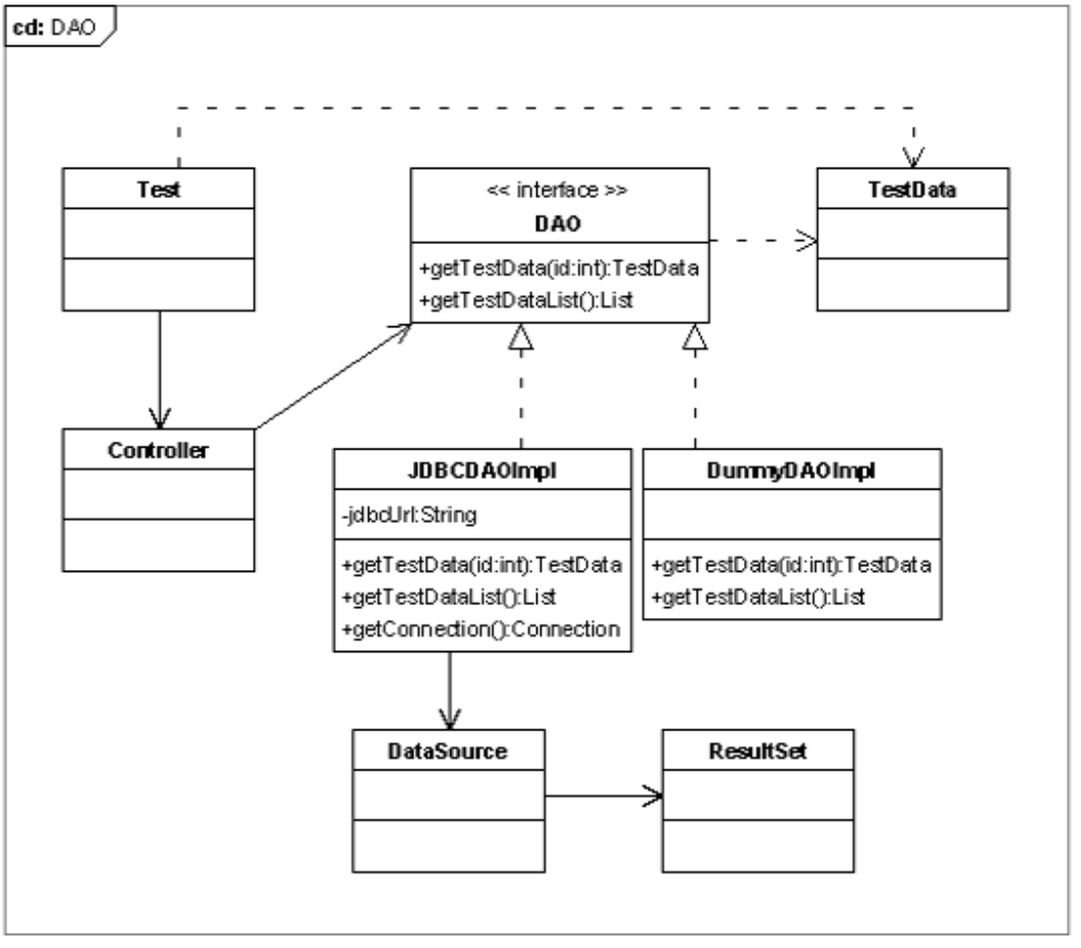
Veri (Persistence) Katmanı

Veri tabanından verileri okuma, değiştirme ve depolama işlemleri bu katmanda gerçekleşir. LoginDao interface sınıfı sistem için kullanılan veri deposu mekanizmasını diğer katmanlar için transparan hale getirir. Veri deposu olarak bir veri tabanı ya da veriler bir bir dosyada tutulabilir. Nasıl bir veri deposu kullanıldığını işletme katmanı LoginDao interface sınıfını kullandığı için bilmek zorunda değildir. Bu bize işletme katmanını etkilemeden sistem için kullanılan veri deposunu değiştirme imkanını tanır. DAO gibi tasarım şablonlarının neden kullanıldığını bu örnekte çok iyi görmüş oluyoruz. Amacımız kısa zamanda karmaşık bir yapıya ulaşan projelerde sistemin bütününü oluşturan modüleri birbirlerini etkilemeyecek şekilde modifike edebilir hale getirmektir. Tasarım şablonları yardımı ile her modül diğerlerine zarar vermeden başka bir implementasyon ile değiştirilebilir hale geldiği taktirde, yazılan kodun geliştirilmesi ve bakımı çok daha kolaylaşır.

DAO Tasarım Şablonu

Birçok programın var olma nedeni veriler üzerinde işlem yapmak, verileri veri tabanlarında depolamak ve bu verileri tekrar edinmektir. Bu böyle olunca verilerin program tarafından nasıl veri tabanlarına konulduğu ve tekrar edinildiği önem kazanmaktadır. Data Access Objects (DAO) tasarım şablonu ile kullanılan veri tabanına erişim ve veri depolama-edinme işlemi daha soyutlaştırılarak diğer katmanların veri tabanına olan bağımlılıkları azaltılır.

DAO ile diğer katmanlar etkilenmeden veri tabanı değiştirilebilir. Daha öncede belirttiğim gibi amacımız birbirini kullanan ama birbirine bağımlılıkları çok az olan katmanlar oluşturmak ve gerekli olduğu zaman bir katmanı diğer katmanlar etkilenmeden değiştirebilmek olmalıdır. Katmanlar arası bağımlılık interface sınıfları üzerinden olduğu sürece bu amacımıza her zaman ulaşabiliriz.



Resim 10.38 DAO tasarım şablonu

DAO tasarım şablonunun merkezinde bir interface sınıf vardır (DAO). Bu interface sınıfını değişik veri tabanı teknolojileri kullanarak implemente edebiliriz. Resim 10.38 de görüldüğü gibi DAO interface sınıfı JDBC teknolojisi kullanılarak implemente edilmiştir. Bunun yanı sıra testlerde kullanılmak üzere bir dummy implementasyon (DummyDAOImpl) mevcuttur.

Veri Katmanı Testleri

Bu katman veri tabanı ile direk bağlantısı olan tek katmandır. İşletme katmanı bu katman üzerinde veri tabanı işlemlerini gerçekleştirebilir. Bu katmanın

implementasyon tarzı nasıl bir veri deposunun kullanıldığını belirler. Bu katmanın implementasyonu için Hibernate ORM teknolojisinden faydalanacağız.

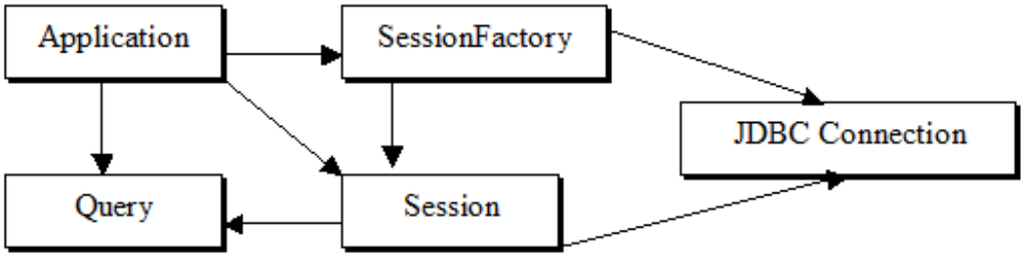
Veri katmanında bulunan LoginDao interface sınıfının implementasyonu için düşündüğümüz testler şu şekildedir.

Kullanıcı hikayesi 2: Kullanıcı email adresi ve şifreni kullanarak sisteme login yapar.

- ***1. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçersizdir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunamaz. EMAIL_INVALID statü kodu oluşturulur.
- ***2. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçersizdir. PASSWORD_INVALID statü kodu oluşturulur.
- ***3. Birim Testi*** - Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçerlidir. LOGIN_SUCCESSFULL statü kodu oluşturulur.

Hibernate ile Veri Katmanı İmplementasyonu

Hibernate in sunduğu API yi kullanarak JDBC teknolojisine bağımlı kalmadan veri tabanı üzerinde işlem yapabiliriz. Veri tabanı üzerinde işlem yapabilmek için Hibernate SessionFactory, Session ve Query gibi interface sınıflar sunmaktadır. Bu sınıflar arasındaki ilişki bir sonraki resimde görülmektedir.



Resim 10.39 SessionFactory, Session ve Query Hibernate API de merkezi rol oynamaktadır

Veri tabanı üzerinde işlem yapabilmek için SessionFactory aracılığıyla bir Session nesnesi edinmemiz gerekiyor. Session, JDBC den tanıdığımız Connection nesnelerine benzer. Session nesnesi ile veri tabanı üzerinde

istediğimiz işlemi gerçekleştirebiliriz. Query sınıfı ile SQL komutları oluşturabiliriz.

Spring ile SessionFactory nesnesini konfigüre edebilir ve veri tabanını kullanabiliriz. Ama LoginDao için gerekli implementasyonu veri tabanı kullanmak zorunda kalmadan gerçekleştirmek istiyoruz. Bu yüzden SessionFactory, Session ve Query sınıflarını mock nesneler kullanarak simüle edeceğiz. Daha sonra oluşturacağımız entegrasyon testlerinde Hibernate ve Spring ile veri tabanını nasıl sisteme entegre edebileceğimizi göreceğiz.

Kod 10.39 LoginDaoImplTest.java

```
package test.shop.unit.persistence.login;

import static org.hamcrest.CoreMatchers.notNullValue;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.classic.Session;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Before;
import org.junit.Test;

import shop.business.login.StatusCodes;
import shop.domain.Customer;
import shop.persistence.login.LoginDaoImpl;
import shop.persistence.login.LoginDaoResult;

public class LoginDaoImplTest {
    private Mockery context;
    private SessionFactory sessionFactory;
    private Session session;
    private Query query;
    private LoginDaoImpl dao;
    private Transaction tx;
    private String email = "test_email";
    private String password = "test_password";
    private final String hql = "from Customer c where c.email= :email";
    private LoginDaoResult result;

    /**
```

```
     * setUp() metodu
     */
    @Before
    public void setUp() {
        context = new Mockery();
        dao = new LoginDaoImpl();
        result = new LoginDaoResult();
        sessionFactory = context.mock(SessionFactory.class);
        session = context.mock(Session.class);
        query = context.mock(Query.class);
        tx = context.mock(Transaction.class);

        email = "test_email";
        password = "test_password";

        context.checking(new Expectations() {
            {
                oneOf(sessionFactory).getCurrentSession();
                will(returnValue(session));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(session).beginTransaction();
                will(returnValue(tx));
            }
        });

        dao.setSessionFactory(sessionFactory);

    }
}
```

Kod 10.39 de LoginDaoImplTest sınıfı yer almaktadır. Kullanmak istediğimiz Hibernate sınıfları için gerekli mock nesnelerini setUp() metodunda oluşturuyoruz.

```
    Kod 10.40   LoginDaoImplTest.java

    @Test
    public void testEmailInvalid() {
        try {

            context.checking(new Expectations() {
                {
                    oneOf(session).createQuery(hql);
```

```
                will(returnValue(query));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).setParameter("email",
                        email);
                will(returnValue(query));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).uniqueResult();
                will(returnValue(null));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(tx).commit();
            }
        });

        LoginDaoResult result = dao.findUser(email,
                password);

        assertTrue(result.getStatus() == StatusCodes.EMAIL_INVALID
                .getValue());

    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```

testEmailInvalid() ismini taşıyan test metodumuzun büyük bir bölümünde Hibernate sınıflarını simüle eden mock nesnelerini programlıyoruz. LoginDaoImpl sınıfında bulunan findUser() metoduyla kullanıcı hesabını veri tabanında arıyoruz.

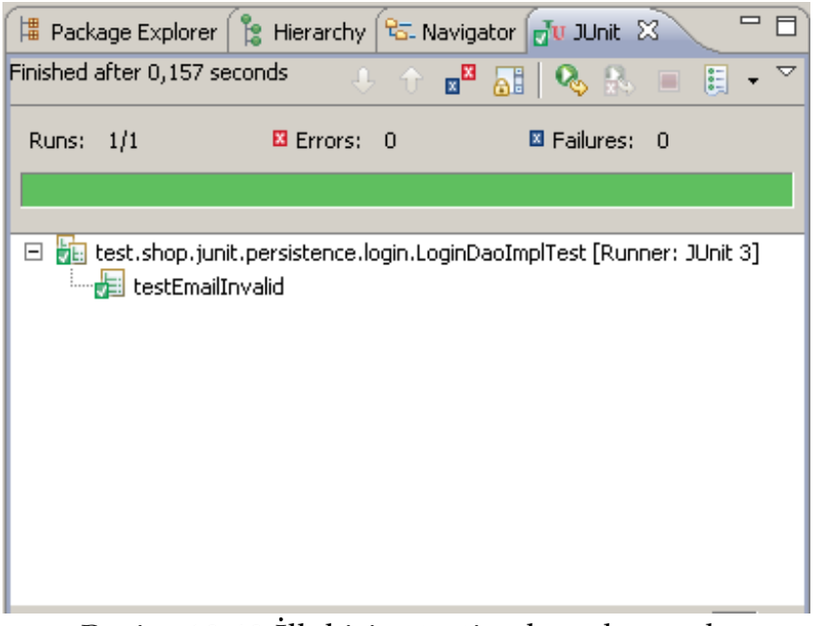
Bu test sınıfının derlenebilmesi için LoginDaoImpl sınıfını oluşturmamız gerekiyor.

Kod 10.41 LoginDaoImpl.java

```
package shop.persistence.login;

import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;

import shop.business.login.StatusCodes;
import shop.domain.Customer;

public class LoginDaoImpl implements LoginDao
{

    private SessionFactory factory;


    public LoginDaoResult findUser(String email, String password)
    {
        LoginDaoResult result = new LoginDaoResult();
        Session session = getSessionFactory().getCurrentSession();
        Transaction tx = session.beginTransaction();
        String hql ="from Customer c where c.email= :email";
        Query query = session.createQuery(hql);
        query.setParameter("email", email);

        Customer customer = (Customer)query.uniqueResult();
        tx.commit();

        if(customer == null)
        {
            result.setStatus(StatusCodes.EMAIL_INVALID.getValue());
        }

        return result;
    }

    public SessionFactory getFactory()
    {
        return factory;
    }


    public void setFactory(SessionFactory factory)
    {
        this.factory = factory;
    }
}
```

LoginDaoImpl sınıfında Hibernate API yi kullanabilmek için SessionFactory

veri tipinde factory ismini taşıyan bir sınıf değişkeni tanımlıyoruz. findUser() metodunu kod 10.41 görüldüğü gibi implemente ediyoruz. Bu implementasyon testin olumlu sonuç vermesi için yeterlidir.



Resim 10.40 İlk birim testi çalışır durumda

İkinci teste geçebiliriz:

2. Birim Testi

Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçersizdir. PASSWORD_INVALID statü kodu oluşturulur.

Kod 10.42 LoginDaoImplTest.java

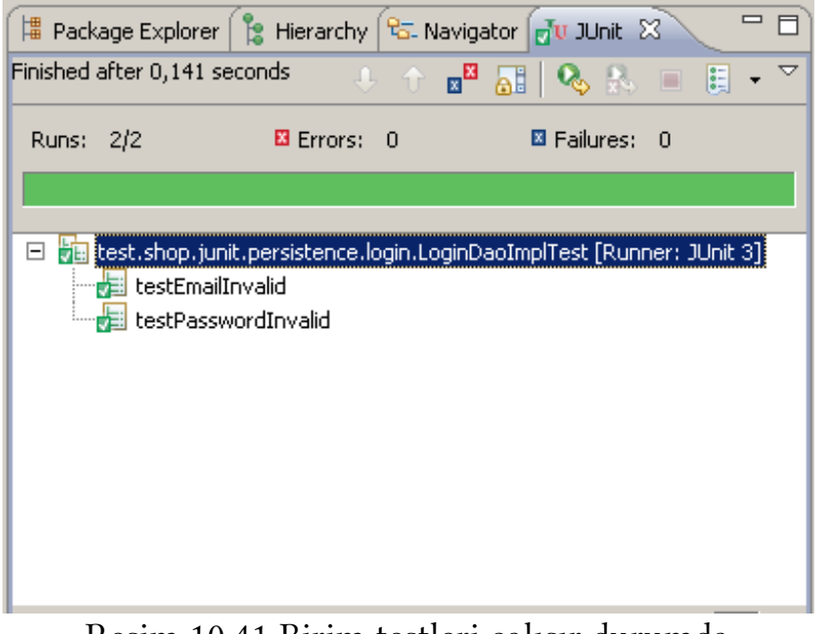
```
@Test
public void testPasswordInvalid() {
    try {

        result.setStatus(StatusCodes.PASSWORD_INVALID
                .getValue());

        final Customer customer = new Customer();
        customer.setEmail(email);
        customer.setPassword("");

        context.checking(new Expectations() {
            {
                oneOf(session).createQuery(hql);
                will(returnValue(query));
```

```
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).setParameter("email",
                        email);
                will(returnValue(query));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).uniqueResult();
                will(returnValue(customer));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(tx).commit();
            }
        });

        LoginDaoResult daoResult = dao.findUser(email,
                password);

        assertTrue(daoResult.getStatus() == StatusCodes.PASSWORD_INVALID
                .getValue());

    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```

Kod 10.43 LoginDaoImpl.java

```
/**
 * findUser()
 */
public LoginDaoResult findUser(String email, String password)
{
    LoginDaoResult result = new LoginDaoResult();
    Session session = getSessionFactory().getCurrentSession();
    Transaction tx = session.beginTransaction();
    String hql ="from Customer c where c.email= :email";
    Query query = session.createQuery(hql);
```

```
    query.setParameter("email", email);

    Customer customer = (Customer)query.uniqueResult();
    tx.commit();

    if(customer == null)
    {
        result.setStatus(
                StatusCodes.EMAIL_INVALID.getValue());
    }
    else
    {
        if(!customer.getPassword().equals(password))
        {
            result.setStatus(
                StatusCodes.PASSWORD_INVALID.getValue());
        }
    }
    return result;
}
```



Resim 10.41 Birim testleri çalışır durumda

Üçüncü teste geçebiliriz:

3. Birim Testi

Kullanıcı tarafından girilen email adresi geçerlidir. Veri tabanında bu email adresiyle ilişkili bir hesap bulunur. Kullanıcı tarafından girilen şifre geçerlidir. LOGIN_SUCCESSFULL statü kodu oluşturulur.

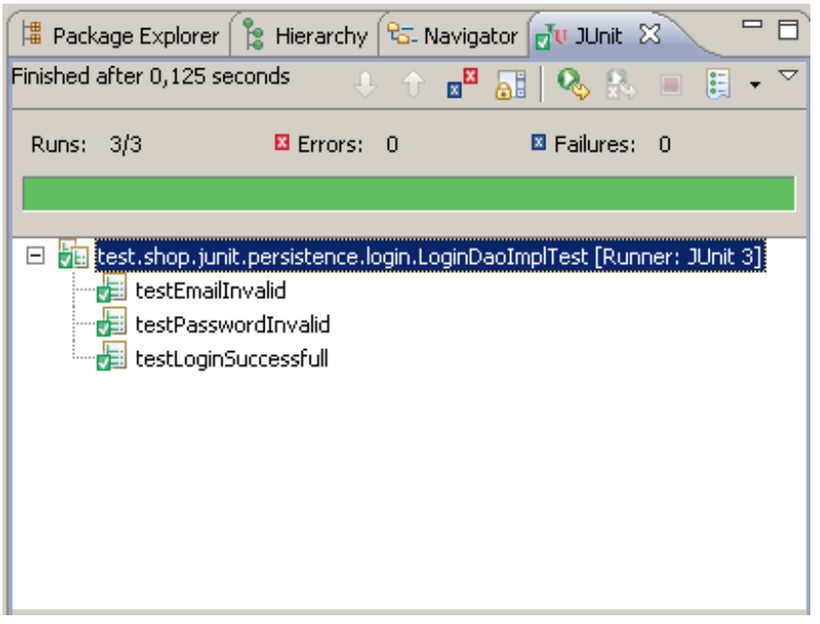
Kod 10.44 LoginDaoImplTest.java

```
@Test
public void testLoginSuccessfull() {
    try {

        final Customer customer = new Customer();
        customer.setEmail(email);
        customer.setPassword(password);

        context.checking(new Expectations() {
            {
                oneOf(session).createQuery(hql);
                will(returnValue(query));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).setParameter("email",
                        email);
                will(returnValue(query));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(query).uniqueResult();
                will(returnValue(customer));
            }
        });

        context.checking(new Expectations() {
            {
                oneOf(tx).commit();
            }
        });

        LoginDaoResult daoResult = dao.findUser(email,
                password);

        assertTrue(daoResult.getStatus() == StatusCodes.LOGIN_SUCCESSFULL
                .getValue());

        assertThat(daoResult.getCustomer(),
                notNullValue());

        assertSame(customer, daoResult.getCustomer());
```

```
    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```

Kod 10.45 LoginDaoImpl.java

```
/**
 * findUser()
 */
public LoginDaoResult findUser(String email, String password)
{
    LoginDaoResult result = new LoginDaoResult();
    Session session = getSessionFactory().getCurrentSession();
    Transaction tx = session.beginTransaction();
    String hql ="from Customer c where c.email= :email";
    Query query = session.createQuery(hql);
    query.setParameter("email", email);

    Customer customer = (Customer)query.uniqueResult();
    tx.commit();

    if(customer == null)
    {
        result.setStatus(
                StatusCodes.EMAIL_INVALID.getValue());
    }
    else
    {
        if(!customer.getPassword().equals(password))
        {
            result.setStatus(
                StatusCodes.PASSWORD_INVALID.getValue());
        }
        else
        {
            result.setCustomer(customer);
            result.setStatus(
                StatusCodes.LOGIN_SUCCESSFULL.getValue());
        }
    }
    return result;
}
```



Resim 10.42 Birim testleri çalışır durumda

Gösterim katmanı, işletme katmanı ve veri katmanı için tüm testleri üç sınıf ve on iki değişik test metodunda implemente ettik. Kod 10.46 da üç test sınıfından oluşan bir JUnit Test Suite sınıfı yer almaktadır. Bu test suite ile tüm testleri arka arkaya koşturabiliriz.

Kod 10.46 AllTests.java - JUnit Test Suite

```
package test.shop.junit;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

import test.shop.unit.business.login.LoginManagerImplTest;
import test.shop.unit.persistence.login.LoginDaoImplTest;
import test.shop.unit.presentation.login.controller.LoginControllerTest;

@RunWith(Suite.class)
@SuiteClasses({ LoginControllerTest.class,
        LoginManagerImplTest.class, LoginDaoImplTest.class })
public class AllTests {
}
```

Entegrasyon Testleri

Oluşturduğumuz birim testleri ile bir veri tabanına ihtiyaç duymadan mock ve stub nesneler yardımıyla değişik katmanlardaki sınıfları test ettik. Şöyle bir

geriye baktığımızda aslında birim testlerin bir simülasyondan ibaret olduğunu görebiliriz. Sınıfların ihtiyaç duydukları gerekli altyapıyı mock ve stub nesneler simüle etti. Mock ve stub nesneleri ihtiyaç duyduğumuz şekilde programlayarak, var olmayan bir altyapıyı varmış gibi gösterdik. Bu noktada haklı olarak şu soruyu sorabilirsiniz: "Oluşturduğumuz sınıflar gerçek bir veri tabanı ile çalışır durumda mı?". Bu sorunun cevabını entegrasyon testleri ile verebiliriz.

Sekizinci bölümü de entegrasyon testlerini nasıl tanımladığımıza tekrar bir göz atalım:

Birçok komponentten oluşan bir sistemde, komponentler arası entegrasyonu test etmek için entegrasyon testleri oluşturulur. Entegrasyon testlerinde mock nesneleri kullanılmaz. Entegrasyon testlerindeki ana amaç sistemin değişik bölümlerinin (subsystem) entegre ederek, işlevlerini kontrol etmektir. Entegrasyon testlerinde test edilen sınıflar için gerekli tüm altyapı (verı tabanı, email sunucusu vs.) çalışır duruma getirilir ve entegrasyon test edilir.

Özellikle veri katmanı için entegrasyon testleri büyük önem taşımaktadır. Veri katmanını gerçek koşullarda test edebilmek için bir veri tabanına sahip olmamız gerekiyor. Bu veri tabanı yazılımı ve testleri zorlaştırmayacak, Ant ile entegre olabilecek yapıda olmalıdır.

Oluşturmak istediğimiz entegrasyon testleri şu şekildedir:

- ***1. Entegrasyon Testi*** - Veri tabanına test_email email adresine sahip bir müşteri bulunmaktadır. Gerekli sınıflar aracılığıyla müşteri bilgileri edinilir.
- ***2. Entegrasyon Testi*** - İşletme katmanı veri katmanı aracılığıyla müşteri bilgilerini edinir. Aranan müşterinin email adresi: test_email

HSQLDB

Ben yazılım esnasında HSQLDB veri tabanını kullanmayı tavsiye ediyorum. HSQLDB tamamen Java dilinde yazılmış, Ant ile entegrasyonu kolay ve kurulum gerektirmeyen bir veri tabanıdır.

HSQLDB yi Ant ile entegre edebilmek için aşağıda yer alan Jar dosyalarının lib dizinine eklenmesi gerekiyor.

- bsh-2.0b4.jar

- hsqldb.jar

HSQLDB Ant entegrasyonu için build.xml içinde kod 10.47 yer alan hedefleri (target) oluşturuyoruz.

Kod 10.47 Ant build.xml

```
<target name="hsqldb-start">
    <java fork="true"
        classname="${hclass}"
        classpath="${hjar}"
        args="${hfile} -dbname.0 ${halias} -port ${hport}"
        spawn="true"
        newenvironment="true"
        failonerror="false" />
</target>


<target name="hsqldb-stop">
    <concat destfile="temp.bsh">
    <![CDATA[
      import java.sql.*;
      Class.forName("org.hsqldb.jdbcDriver");
      String url = "jdbc:hsqldb:hsql://127.0.0.1:9006/" + bsh.args[0];
      Connection con = DriverManager.getConnection(url, "sa", "");
      String sql = "SHUTDOWN";
      Statement stmt = con.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
    ]]>
    </concat>
    <java
        classname="bsh.Interpreter"
        fork="true"
        failonerror="false"
        resultproperty="hsql-stop-result">
        <classpath refid="compile.classpath" />
        <arg line="temp.bsh ${halias}" />
    </java>
</target>


<target name="hsqldb-client-start">
    <java fork="true"
        classpath="${hjar}"
        classname="org.hsqldb.util.DatabaseManagerSwing" />
</target>
```

Kod 10.48 ant.properties

```
#hsql
hjar=lib/hsqldb.jar
hclass=org.hsqldb.Server
hfile=-database.0 db/hsqldb-data/
halias=shop
hport=9006
```

DBUnit

Entegrasyon testleri için gerekli verilerin veri tabanında bulunması gerekmektedir. Test öncesi bu verilerin veri tabanında kullanılır hale getirilmeleri gerekiyor. Bu işlem için DBUnit den faydalanabiliriz. Kitabın altıncı bölümünde detaylı olarak incelediğimiz DBUnit ile test verileri test öncesi HSQLDB veri tabanına yerleştirilir.

Veri tabanına yerleştirmek istediğimiz verileri dbunit-dataset.xml isminde bir dosyaya yerleştiriyoruz:

Kod 10.49 dbunit-dataset.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

  <customer
    id="1"
    email="test_email"
    password="test_password"/>

  <customer
    id="2"
    email="test_email2"
    password="test_password2"/>

</dataset>
```

Veri tabanında customer isminde, müşteri bilgilerinin tutulduğu bir tablo bulunmaktadır. Bu tablonun email ve password isimlerinde iki kolonu vardır. DBUnit test öncesi dbunit-dataset.xml dosyasında tanımladığımız iki müşteriyi customer tablosuna ekleyecektir. Bu sayede entegrasyon testleri customer tablosundan gerek duydukları verileri elde edebilir hale gelirler.

DBUnitTestCase

Her test başlangıcında dbunit-dataset.xml dosyasında bulunan verilerin veri tabanına yüklenmesi gerekir. Bu amaçla DBUnitTestCase ismini taşıyan ve setUp() metodunda verilerin veri tabanına yükleme işleminin gerçekleştiği yeni bir sınıf oluşturdum. Oluşturacağımız entegrasyon testleri DBUnitTestCase sınıfını genişleterek hem bir DBUnit sınıfı halini alacaklar hem de dbunit-dataset.xml dosyasında bulunan verilerin bu sınıf tarafından otomatik olarak veri tabanına yüklenmesini sağlayacaklar. DBUnitTestCase kod -10.49 da yer almaktadır.

```
Kod 10.50   DBUnitTestCase

package test.shop.common.test;

import java.io.File;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import org.dbunit.DBTestCase;
import org.dbunit.DatabaseTestCase;
import org.dbunit.database.DatabaseConfig;
import org.dbunit.database.DatabaseConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.ext.hsqldb.HsqldbDataTypeFactory;
import org.dbunit.operation.DatabaseOperation;

public abstract class DBUnitTestCase extends DBTestCase
{

    private static final String DRIVERCLASS =
        "org.hsqldb.jdbcDriver";

    private static final String CONNECTIONURL =
        "jdbc:hsqldb:hsql://localhost:9006/shop";

    private static final String USERNAME = "sa";

    private static final String PASSWORD = "";


    protected void setUp()
      {
```

```
        try
        {
            IDatabaseConnection connection =
                getConnection();
            IDataSet dataSet = getDataSet();
            try
            {
                DatabaseOperation.CLEAN_INSERT
                    .execute(connection, dataSet);
            }
            finally
            {
                connection.close();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
      }


    protected final IDataSet getDataSet()
    throws Exception
    {
        final URL url =
            DatabaseTestCase.class
            .getResource("/dbunit-dataset.xml");
        final File file = new File(url.getPath());
        return new FlatXmlDataSet(file);
    }


    protected IDatabaseConnection getConnection()
    throws Exception
    {

        Class driverClass = Class.forName(DRIVERCLASS);

        Connection jdbcConnection =
            DriverManager.getConnection(
                CONNECTIONURL, USERNAME, PASSWORD);

        IDatabaseConnection con =
            new DatabaseConnection(jdbcConnection);

        DatabaseConfig config = con.getConfig();
        config.setProperty(
                DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
```

```
                new HsqldbDataTypeFactory());
        return con;
    }
}
```

İlk entegrasyon testine geçebiliriz:

1. Entegrasyon Testi

Veri tabanında test_email email adresine sahip bir müşteri bulunmaktadır. Gerekli sınıflar aracılığıyla müşteri bilgileri edinilir.

```
Kod 10.51   LoginDaoImplTest.java

package test.shop.integration.persistence.login;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:spring-servlet-integration.xml" })
public class LoginDaoImplTest extends DBUnitTestCase {

    @Autowired
    ApplicationContext ctx;

    private LoginDao dao;

    @Before
    public void setUp() {
        super.setUp();

        dao = (LoginDao) ctx.getBean("loginDao");
    }
    ...
}
```

Veri tabanı işlemlerinden LoginDao interface sınıfı sorumlu olduğu için ilk entegrasyon testini bu interface sınıfın implementasyonu olan LoginDaoImpl sınıfı için oluşturuyoruz.

İlk entegrasyon testini veri tabanı yanı sıra Spring konfigurasyonunu da test edecek şekilde yapılandırıyoruz. LoginDaoImpl sınıfından bir nesneyi Spring aracılığıyla edineceğiz.

@ContextConfiguration anotasyonu ile kullandığımız Spring konfigürasyon dosyasını yüklüyoruz. LoginDaoImplTest sınıfı kod 10.50 de yer alan DBUnitTestCase sınıfını genişletmektedir. super.setUp() ile bu üst sınıfta yer

alan setUp() metodu koşturulmakta ve böylece dbunit-dataset.xml dosyasında tanımladığımız veriler her test öncesinde veri tabanına aktarılmatadır. ctx.getBean("loginDao") ile spring-servlet-integration.xml dosyasında tanımlanmış olan loginDao Spring nesnesini ediniyoruz. Kod 10.52 de ilk entegrasyon test yer almaktadır.

Kod 10.52 LoginDaoImplTest.java

```
@Test
public void testEmail() {
    try {
        String email = "test_email";
        String password = "test_password";

        LoginResult result = manager.login(email,
                password);

        assertEquals(email, result.getCustomer()
                .getEmail());
        assertEquals(password, result.getCustomer()
                .getPassword());
    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}
```

Bu test çalıştırıldığında doğal olarak şöyle bir hata oluşacaktır:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException:
            No bean named 'loginDao' is defined
```

spring-servlet-integration.xml dosyasında loginDao ismini taşıyan bir Spring Bean bulunmuyor! Şimdiye kadar hazırladığımız birim testlerde Spring ve gerekli konfigürasyon dosyasını kullanmadık, çünkü devamlı mock nesneler ile çalıştık. Bu bize Spring ayarları yapmadan çalışma imkanı verdi, lakin bu durum burada sona eriyor! Entegrasyon testlerinin neden gerekli olduğunu şimdi daha iyi görebiliyoruz. Entegrasyon testleri gerekli tüm altyapının mevcut olmasını gerekli kılar. Buna Spring için yapılması gereken konfigürasyon da dahildir.

Bu noktada bir taşla iki kuş vurmuş oluyoruz. İlk entegrasyon testimiz hem veri tabanı altyapısını hem de Spring konfigürasyonunu test edecek.

Kod 10.53 spring-servlet.xml

```
<bean id="loginDao" class="shop.persistence.login.LoginDaoImpl">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>
```

Kod 10.53 de loginDao Spring Bean tanımlaması yer almaktadır. LoginDaoImpl sessionFactory isminde bir değişken kullanmaktadır. Bu Hibernate in veri tabanı bağlantısı oluşturmak için sunduğu SessionFactory interface sınıfıdır. LoginDao ile Hibernate arasındaki bağlantı sessionFactory değişkeni üzerinden gerçekleşir. Burada dependency injection metodunu kullanarak, LoginDaoImpl sınıfının ihtiyaç duyduğu sessionFactory değişkenini enjekte ediyoruz. Spring konfigürasyonu üzerinden bir loginDao nesnesi edindiğimizde, bu nesnenin sessionFactory değişkeni Spring tarafından otomatik olarak oluşturulacaktır.

Kod 10.54 spring-servlet.xml

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
    annotation.AnnotationSessionFactoryBean">
    <property name="configurationClass">
        <value>org.hibernate.cfg.AnnotationConfiguration</value>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.HSQLDialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.use_sql_comments">true</prop>
        </props>
    </property>
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
    <property name="annotatedClasses">
        <list>
            <value>shop.domain.Customer</value>
        </list>
    </property>
</bean>
```

SessionFactory bir dataSource nesnesine ihtiyaç duymaktadır. dataSource

kullanmak istediğimiz HSQLDB veri tabanının ayarlarını ihtiva eder. Ayrıca sessionFactory bünyesinde Hibernate tarafından yönetilen sınıfların listesi yer alır (annotatedClasses). Customer sınıfından olan nesneler Hibernate tarafından veri tabanında kalıcı (persistent) hale getirilebilir. Bu nesnelere ait olan değişken değerleri veri tabanında customer isimli tabloda tutulur. Customer sınıfının nasıl annotation değerler kullanılarak veri tabanı için kullanılır hale getirildiğini daha sonra yakından inceleyeceğiz.

Kod 10.55 spring-servlet.xml

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName"
                    value="org.hsqldb.jdbcDriver" />
    <property name="url"
        value="jdbc:hsqldb:hsql://localhost:9006/shop" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

İlk entegrasyon testini çalışır hale getirebilmek için Customer sınıfını Hibernate tarafından veri tabanında kalıcı nesne olarak kullanılır hale getirmemiz gerekiyor.

Kod 10.56 Customer.java

```
1. package shop.domain;

2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;

8. @Entity
9. @Table(name="customer")
10. public class Customer
11. {
12. @Id
13. @GeneratedValue(strategy = GenerationType.AUTO)
14. @Column(name = "id")
15. private Long id;
16.
17. @Column(name = "email", nullable = false)
```

```
18. private String email;
19.
20. @Column(name = "password", nullable = false)
    private String password;

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }
}
```

Sekizinci satırda Customer sınıfı için @Entity anotasyonunu kullanıyoruz. Hibernate tarafından veri tabanında tutmak istediğimiz nesnelerin @Entity tipinde olması gerekiyor. Dokuzuncu satırda @Table anotasyonu ile kullanmak istediğimiz veri tabanı tablosunu tanımlıyoruz.

Customer sınıfından olan nesneleri veri tabanında birbirinden ayırt edebilmek için id isminde bir sınıf değişkeni tanımlıyoruz. Bu nesnenin veri tabanındaki ana anahtar (primary key) kolonudur. Bu değişken on ikinci satırda kullandığımız @Id anotasyonu ile veri tabanında primary key kolonu haline gelir. On üçüncü satırda Customer sınıfından oluşturulan her nesne için yeni bir rakamın id olarak kullanılmasını sağlıyoruz. On dördüncü satırda id isimli değişkeni @Column anotasyonu ile veri tabanında yer alan customer tablosunun id isimli kolonuna bağlıyoruz.

Customer sınıfı email ve password isimlerinde iki değişkene sahiptir. Bu değişkenlerin değerlerini veri tabanında bulunan customer tablosunda tutabilmek için customer tablosunun email ve password isminde iki yeni kolona ihtiyacı vardır. Sınıf değişkenleri ve tablo kolonları arasındaki bağlantıyı on

yedi ve yirminci satırlarda @Column anotasyonu ile gerçekleştiriyoruz.

Spring için gerekli konfigürasyonu oluşturduktan sonra ilk entegrasyon testimiz çalışır hale gelir.

Testi derleyebilmek ve çalıştırabilmek için bir dizi yeni Jar dosyasına ihtiyacımız vardır. Gerekli Jar dosyaların lib dizine eklenmesi gerekiyor. Şimdiye kadar gerekli olan Jar dosyaların listesi aşağıda yeralmaktadır.

- dbunit-2.2.jar
- emma_ant.jar
- emma.jar
- hamcrest-core-1.3.jar
- hamcrest-library-1.3.jar
- hsqldb.jar
- jmock-2.6.0.jar
- jmock-junit4-2.6.0.jar
- jsp-api.jar
- junit-4.11.jar
- selenium-java-client-driver.jar
- selenium-server.jar
- servlet-api.jar
- antlr-2.7.6.jar
- asm-attrs.jar
- asm.jar
- bsh-2.0b4.jar
- cglib-2.1.3.jar
- commons-beanutils-170.jar
- commons-collections-3.1.jar
- commons-dbcp-1.2.2.jar
- commons-logging-1.0.4.jar
- commons-pool-1.3.jar
- dom4j-1.6.1.jar
- ejb3-persistence.jar
- hibernate-annotations.jar
- hibernate-commons-annotations.jar
- hibernate-tools.jar
- hibernate-validator-4.1.0.Final.jar
- hibernate3.jar
- hsqldb.jar

- javax.validation-1.0.0.GA.jar
- jstl.jar
- jta.jar
- log4j-1.2.15.jar
- slf4j-api-1.7.7.jar
- spring-aop-3.2.8.RELEASE.jar
- spring-aspects-3.2.8.RELEASE.jar
- spring-beans-3.2.8.RELEASE.jar
- spring-build-src-3.2.8.RELEASE.jar
- spring-context-3.2.8.RELEASE.jar
- spring-context-support-3.2.8.RELEASE.jar
- spring-core-3.2.8.RELEASE.jar
- spring-expression-3.2.8.RELEASE.jar
- spring-framework-bom-3.2.8.RELEASE.jar
- spring-instrument-3.2.8.RELEASE.jar
- spring-instrument-tomcat-3.2.8.RELEASE.jar
- spring-jdbc-3.2.8.RELEASE.jar
- spring-jms-3.2.8.RELEASE-sources.jar
- spring-jms-3.2.8.RELEASE.jar
- spring-orm-3.2.8.RELEASE.jar
- spring-oxm-3.2.8.RELEASE.jar
- spring-test-3.2.8.RELEASE.jar
- spring-tx-3.2.8.RELEASE.jar
- spring-web-3.2.8.RELEASE.jar
- spring-webmvc-3.2.8.RELEASE.jar
- standard.jar

Gerekli olan Jar dosyaları Shop projesinin lib ve web/WEB-INF/lib dizinlerinde yer almaktadır.

Bir sonraki entegrasyon testine geçebiliriz:

2. Entegrasyon Testi

İşletme katmanı veri katmanı aracılığıyla müşteri bilgilerini edinir. Aranan müşterinin email adresi: test_email

Bu test bize LoginManagerImpl sınıfı için bir testin hazırlanması gerektiğini belirtiyor.

Kod 10.57 LoginMnagerImplTest.java

```
package test.shop.integration.business.login;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import shop.business.login.LoginManager;
import shop.business.login.LoginResult;
import test.shop.common.test.DBUnitTestCase;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:spring-servlet-integration.xml" })
public class LoginManagerImplTest extends DBUnitTestCase {

    @Autowired
    ApplicationContext ctx;

    private LoginManager manager;

    @Before
    public void setUp() {
        super.setUp();
        manager = (LoginManager) ctx
                .getBean("loginManager");

    }

    @Test
    public void testEmail() {
        try {
            String email = "test_email";
            String password = "test_password";

            LoginResult result = manager.login(email,
                    password);

            assertEquals(email, result.getCustomer()
                    .getEmail());
            assertEquals(password, result.getCustomer()
                    .getPassword());
        } catch (Exception e) {
            e.printStackTrace();
            fail();
        }
```

```
    }
}
```

spring-servlet.xml içinde loginManager isminde bir Bean tanımlamadığımız için ikinci entegrasyon testi aşağıda yer alan hatayı gösterir:

org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'loginManager' is defined

Kod 10.58 spring-servlet.xml

```
<bean id="loginManager" class="shop.business.login.LoginManagerImpl">
    <property name="dao">
        <ref bean="loginDao" />
    </property>
</bean>
```

Kod 10.58 de loginManager isminde yeni bir Spring Bean tanımlıyoruz. İmplementasyon sınıfı olarak LoginManagerImpl sınıfını kullanıyoruz. LoginManagerImpl sınıfının LoginDao sınıfına bağımlılığı vardır. Bu bağımlılığı tatmin etmek için loginDao nesnesi enjekte edilir.

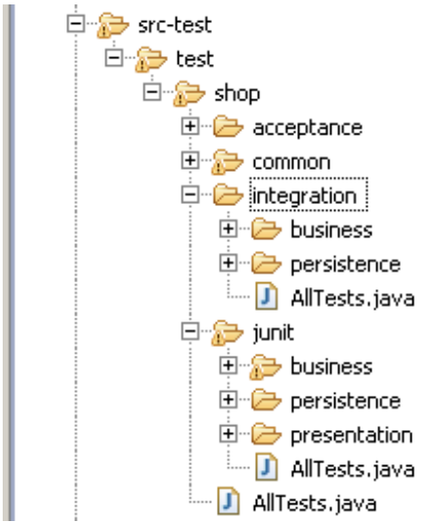
Bu değişikliklerin ardından her iki entegrasyon testimizde olumlu sonuç verecektir.

Her iki testide bir Test Suite sınıfında toplamak için AllTests sınıfını oluşturuyoruz:

Kod 10.59 AllTests.java

```
package test.shop.integration;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

import test.shop.integration.business.login.LoginManagerImplTest;
import test.shop.integration.persistence.login.LoginDaoImplTest;

@RunWith(Suite.class)
@SuiteClasses({ LoginManagerImplTest.class,
        LoginDaoImplTest.class })
public class AllTests {
}
```

Daha öncede oluşturduğumuz onay/kabul ve birim testleri ile genel Test dizin yapısı bir sonraki resimde yer almaktadır:



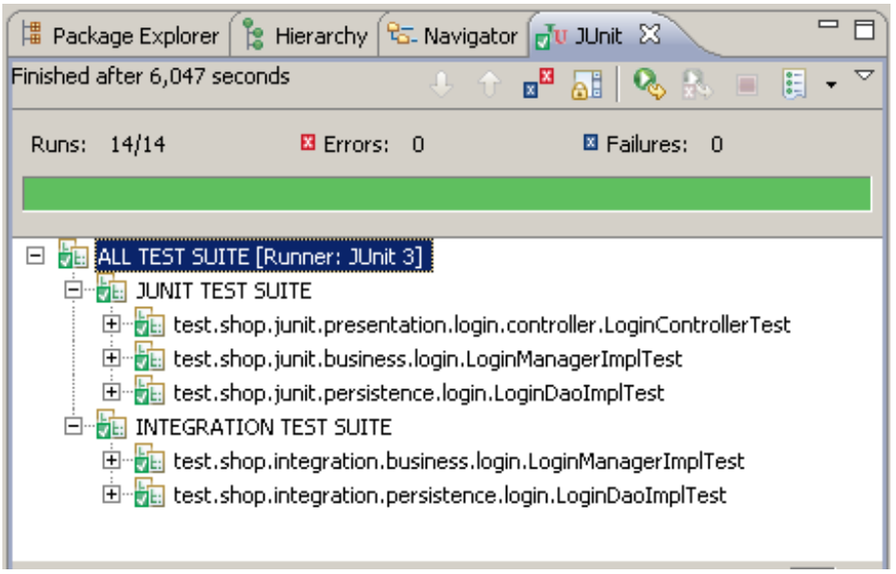
Resim 10.43 Test dizin yapısı

Oluşturduğumuz entegrasyon testleri src-test/test/shop/integration dizininde, birim testleri src-test/test/shop/junit dizininde, onay/kabul testleri src-test/test/shop/acceptance dizininde yer almaktadır. Birden fazla test sınıfını bir araya getirmek için AllTest.java isminde test suite sınıfları oluşturduk. Örneğin test.shop.integration.AllTest sınıfını kullanarak, her iki entegrasyon testini çalıştırabiliriz.

Bir adım daha ileri giderik, tüm birim ve entegrasyon testlerini bir test suite sınıfında toplayabiliriz.

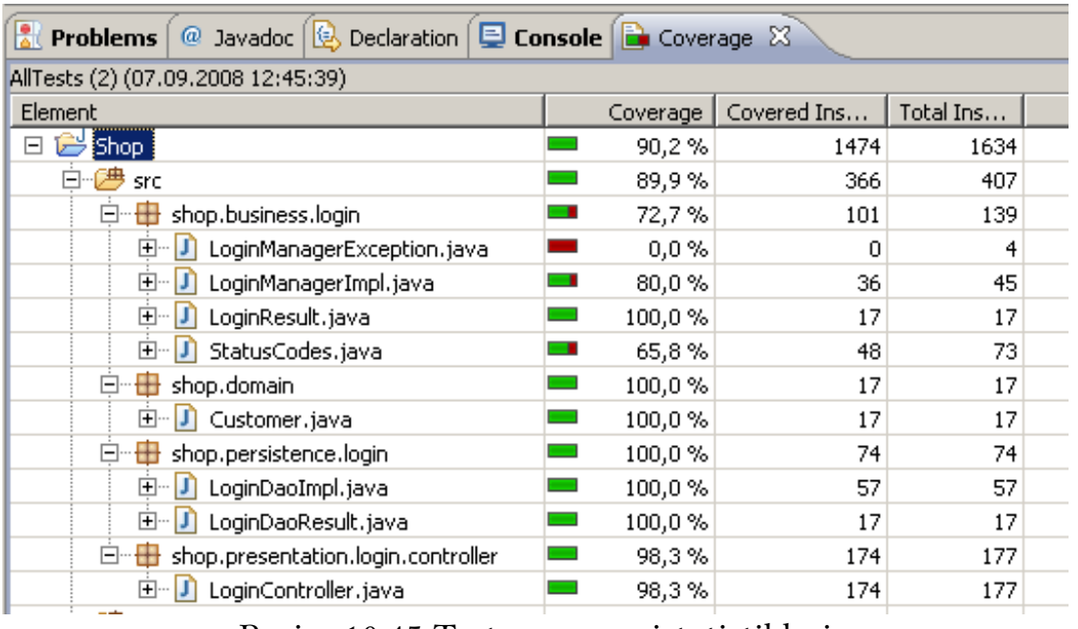
Kod 10.60 AllTests.java

```
package test.shop;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ test.shop.unit.AllTests.class,
        test.shop.integration.AllTests.class })
public class AllTests {
}
```



Resim 10.44 AllTests

Kod 10.60 de yer alan test suite bize oluşturduğumuz tüm testlerin merkezi bir sınıf aracılığıyla çalıştırılması imkanını sağlar. Resim 10.44 de görüldüğü gibi bütün birim ve entegrasyon testleri AllTests test suite ile koşturulabilmektedir



Element	Coverage	Covered Ins...	Total Ins...
Shop	90,2 %	1474	1634
src	89,9 %	366	407
shop.business.login	72,7 %	101	139
LoginManagerException.java	0,0 %	0	4
LoginManagerImpl.java	80,0 %	36	45
LoginResult.java	100,0 %	17	17
StatusCodes.java	65,8 %	48	73
shop.domain	100,0 %	17	17
Customer.java	100,0 %	17	17
shop.persistence.login	100,0 %	74	74
LoginDaoImpl.java	100,0 %	57	57
LoginDaoResult.java	100,0 %	17	17
shop.presentation.login.controller	98,3 %	174	177
LoginController.java	98,3 %	174	177

Resim 10.45 Test coverage istatistikleri

EclEmma ile test kapsama alanı istatistiklerini gözden geçirdiğimiz zaman, kapsama alanının %89.9 a yükseldiğini görmekteyiz. Bu istatistikleri elde etmek için kod 10.59 de oluşturduğumuz test suite sınıfını kullandık. Böylece test kapsama alanının tespiti için şimdiye kadar oluşturduğumuz tüm testleri kullanılmış olduk.

Onay/Kabul Testimiz Ne Oldu?

Bu bölümün başında bir onay/kabul testi oluşturarak, implementasyona başlamıştık hatırlarsanız. Bu test en son bıraktığımız halde çalışmaz durumdaydı, çünkü login için gerekli sınıfları implemente etmemiştik. Birim ve entegrasyon testleri ile login modülü için gerekli sınıfları implemente ettik. Bu durumda onay/kabul testi olumlu sonuç vermeli, öyle değil mi?

Tekrar onay/kabul testimizi hatırlayalım:

1. Onay/kabul Testi

Kullanıcı login sayfasına gider. Email adresi ve şifre alanlarını boş bırakarak login butonuna tıklar. Kullanıcıya "Lütfen email adresinizi ve şifrenizi giriniz!" hata mesajı gösterilir.

Kod 10.61 Ilk akseptans test sınıfı

```
package acceptance.login;

import com.thoughtworks.selenium.*;

public class LoginTest extends SeleneseTestCase
{
    public void setUp() throws Exception
    {
        setUp("http://localhost/", "*chrome");
    }


    public void testEmailAndPasswordEmpty() throws Exception
    {
        selenium.open("/");
        selenium.click("link=Login");
        selenium.waitForPageToLoad("30000");
        selenium.click("Login");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Lütfen email adresinizi " +
                "ve şifrenizi giriniz!"));
    }
}
```

Gösterim katmanı için LoginController sınıfını oluşturmuştuk. Onay/kabul testinin olumlu sonuç verebilmesi için login.jsp sayfasını oluşturmamız gerekiyor. login.jsp sayfası HTML kullanıcı arayüzünü ihtiva eder.

Kod 10.62 login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
                pageEncoding="UTF-8"%>
<%@taglib uri="http://www.springframework.org/tags/form"
                    prefix="form"%>
<html>
<head>
<title>Ara&ccedil; Kiralama Formu</title>
</head>
<body>
    <h2>Ara&ccedil; Kiralama Formu</h2>

    <form:form method="post" action="/app/login/login"
                        commandName="loginForm">
        <table>
            <tr>
                <td>E-posta adresi :</td>
                <td><form:input path="customer.email" />
                   <font color="Red">
                     <form:errors path="customer.email" delimiter=","/>
                   </font>
                </td>
            </tr>
            <tr>
                <td>Sifre :</td>
                <td><form:input path="customer.password" />
                  <font color="Red">
                    <form:errors path="customer.password" delimiter=","/>
                  </font>
                </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit"
                    value="G&ouml;nder" /></td>
            </tr>
        </table>

    </form:form>
</body>
</html>
```

login.jsp sayfasını web/WEB-INF/jsp/login dizinine yerleştiriyoruz.

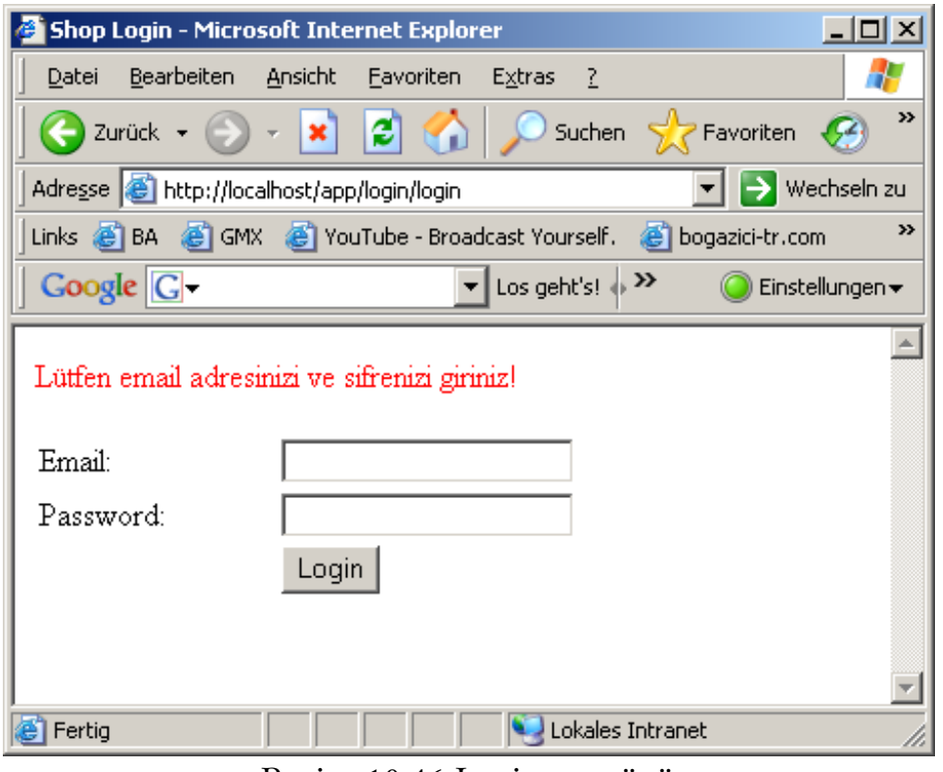
Kod 10.63 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
```

```
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Shop</display-name>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:spring-servlet.xml
        </param-value>
    </init-param>

  </servlet>


  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring-servlet.xml</param-value>
  </context-param>



  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>



  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Bir web uygulaması web.xml dosyası üzerinde konfigüre edilir. web.xml dosyası web/WEB-INF dizininde bulunur. Kod 10.63 de kullandığımız web.xml dosyası yer almaktadır.

Uygulama sunucusu olarak Tomcat 7 sürümünü kullanıyoruz. Kitabın altıncı bölümünde Eclipse ile Tomcat in nasıl entegre edildiğini görmüştük. Tomcat uygulama sunucusu Eclipse altında çalıştırdığımız taktirde bir sonraki resimde yer alan Login arayüzü ekrana gelecektir.



Resim 10.46 Login arayüzü

Bu eklemelerin ardından login modülü onay/kabul testleri için hazır duruma gelmiştir. Onay/kabul testini çalıştırabilmemiz için Selenium sunucusunun çalışır durumda olması gerekir. Selenium sunucusunu aktif ettikten sonra Eclipse altında onay/labul testini çalıştırabiliriz.

Diğer beş onay/kabul testini verdiğim örnekler doğrultusunda kolayca yapabilirsiniz. Bu bölümü daha fazla uzatmamak adına diğer onay/kabul testlerin hazırlanmasını alıştırma olarak sizlere bırakıyorum.

Sürekli Entegrasyon

Bu bölümü de sürekli entegrasyon için gerekli altyapıyı oluşturduk. Sürekli entegrasyonun Cruise Control kullanılarak nasıl yapıldığını kitabın on

dördüncü bölümünde yakından inceleyeceğiz.

11. Bölüm

Onay/Kabul Testleri

Giriş

Bu bölümde onay/kabul testleri olarak bilinen ve program kullanıcılarının beklentilerini dile getiren testlerden örnekler vermek istiyorum. Onay/kabul testlerinin olumlu sonuç vermesi programın kullanıcılar tarafından kabul görme kriterlerinin yerine getirildiği anlamına gelir.

Sekizinci bölümde onay/kabul testlerini nasıl tanımladığımızı tekrar hatırlayalım:

Onay/kabul testleri ile sistemin bütünü kullanıcı gözüyle test edilir. Bu tür testlerde sistem kara kutu olarak düşünülür. Bu yüzden onay/kabul testlerinin diğer bir ismi kara kutu testleridir (black box test). Kullanıcının sistemin içinde ne olup bittiğine dair bir bilgisi yoktur. Onun sistemden belirli beklentileri vardır. Bu amaçla sistem ile interaksiyona girer. Onay/kabul testlerinde sistemden beklenen geri bildirim test edilir.

Onay/kabul testlerini oluşturmak için Selenium , Watir, Sahi ve Canoo Webtest gibi programlardan (framework) faydalanabiliriz. Kitabın bu bölümünde sizlere Selenium ve Canoo Webtest programlarını yakından tanıtmak istiyorum. Önce Selenium ile başlayalım.

Selenium

Selenium onay/kabul testlerinde kullanabileceğimiz ve değişik test araçlarının bir arada geldiği bir test programıdır. Selenium IDE, Selenium Remote Control ve Selenium Grid programlarından oluşur. Selenium programını nasıl kullanabileceğimizi yakından inceleyelim.

Selenium ile oluşturduğumuz onay/kabul testleri HTML formatında ya da JUnit Java sınıfı formatında olabilir. Selenium IDE programını kullanarak test etmek istediğimiz websayfanın üzerinde gerekli aksiyonları video kaydı yapar gibi kayıtlayarak, otomatik olarak testleri oluşturabiliriz. Bu işlem için Firefox plugini olan Selenium IDE programından faydalanıyoruz.

Selenium IDE

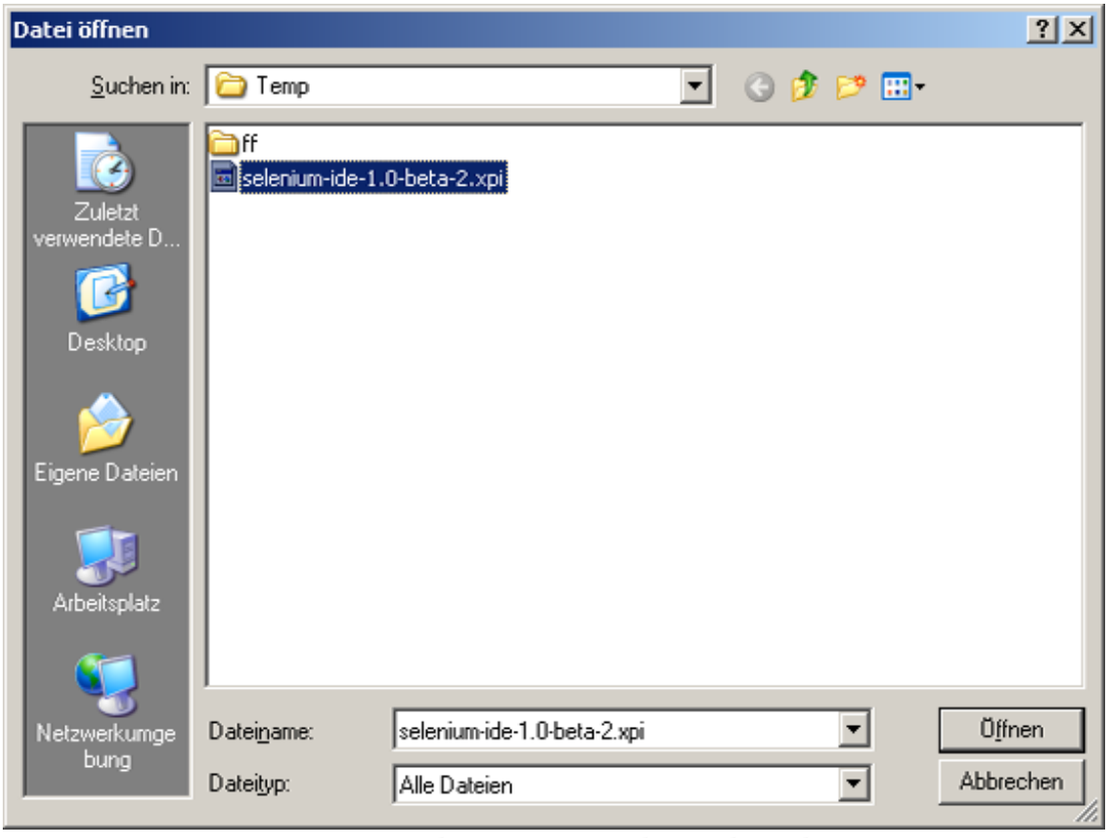
Selenium IDE nin Firefox için olan plugin sürümünü http://docs.seleniumhq.org/download/ adresinden temin edebilirsiniz. Ben

örneklerimde Version 1.0 Beta 2 sürümünü kullanacağım.

Download işleminin ardından selenium-ide-1.0-beta-2.xpi isimli dosyayı bilgisayarımın uygun bir dizinine kaydediyorum. Bu bir Firefox plugindir ve Firefox altında kurulumunu yapmamız gerekiyor. Bunun için Firefox web tarayıcısını çalıştırıyoruz.

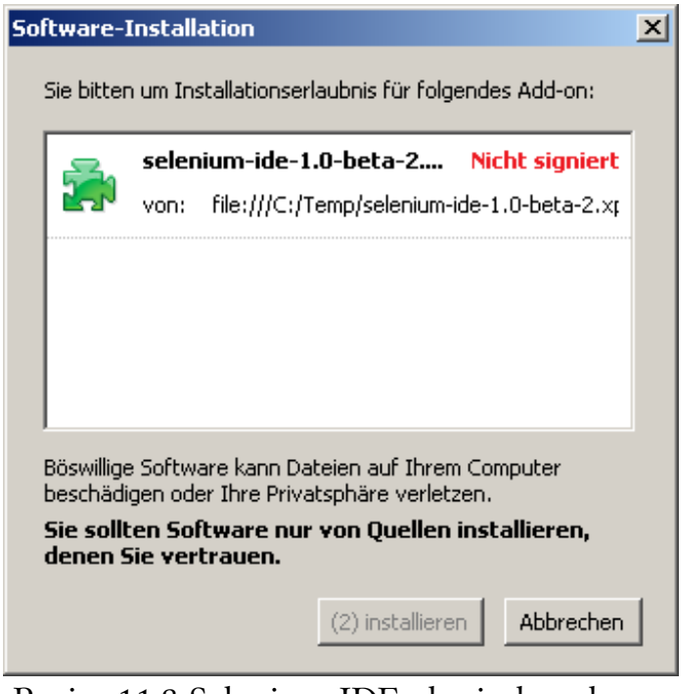
Resim 11.1 Firefox web tarayıcısı

Selenium IDE programının Firefox altında kurulumunu yapabilmek için Dosya (resimde Datei) menüsünden Dosya aç alt menüsünü seçiyoruz.



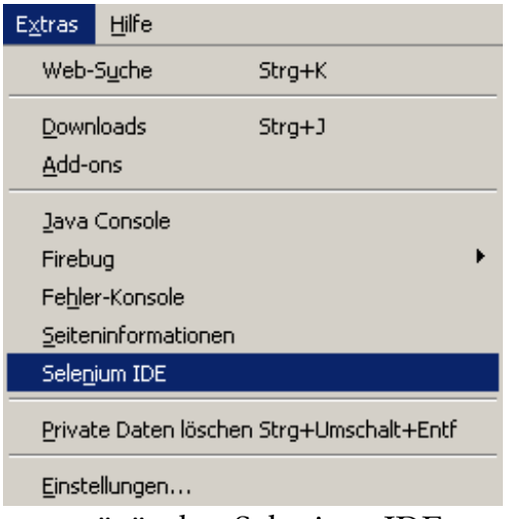
Resim 11.2 Selenium IDE plugin kurulumu

selenium-ide-1.0-beta-2.xpi dosyasını seçtikten sonra, Firefox bu dosyanın bir Firefox plugini olduğunu anlıyor ve resim 11.3 yer alan kurulum panelini gösteriyor.



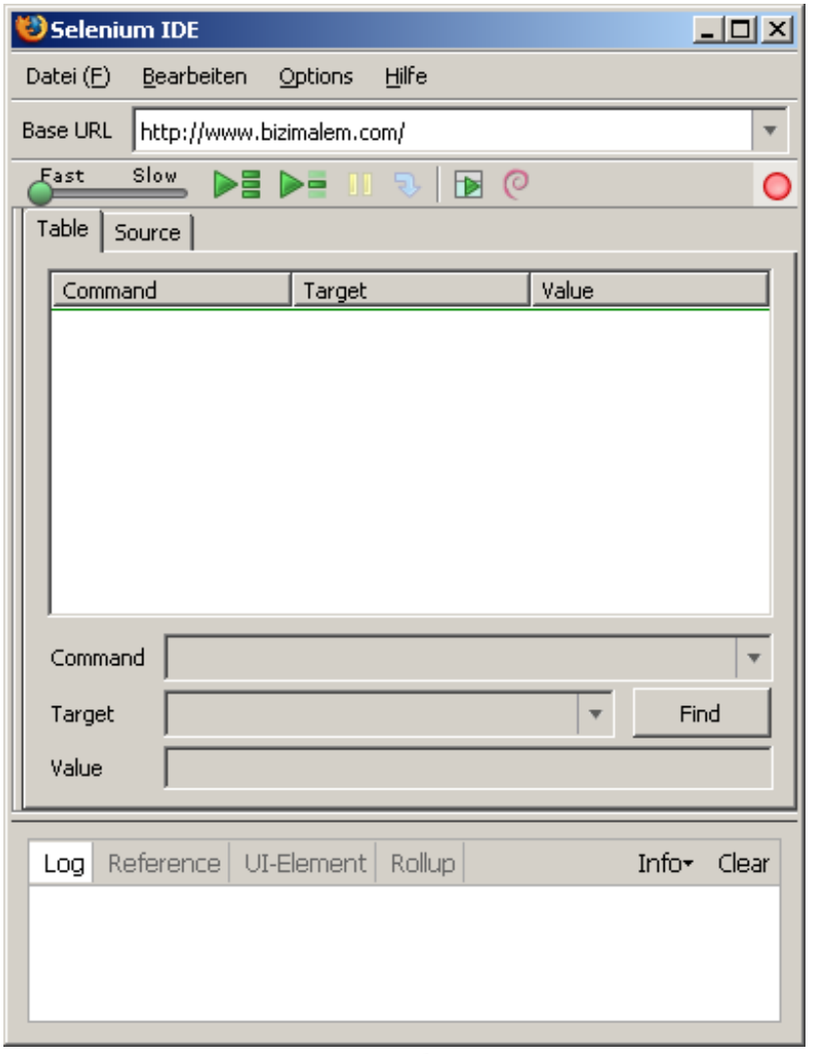
Resim 11.3 Selenium IDE plugin kurulumu

Bu işlemlerin ardından Selenium IDE Firefox plugini olarak kurulur ve kullanıma hazır hale getirilir. Selenium IDE programını Extralar > Selenium IDE menüsünden çalıştırabiliriz.



Resim 11.4 Ekstralar menüsünden Selenium IDE programına erişebiliriz

Selenium IDE resim 11.5 olduğu gibi tipik bir Firefox penceresi olarak karşımıza çıkar.

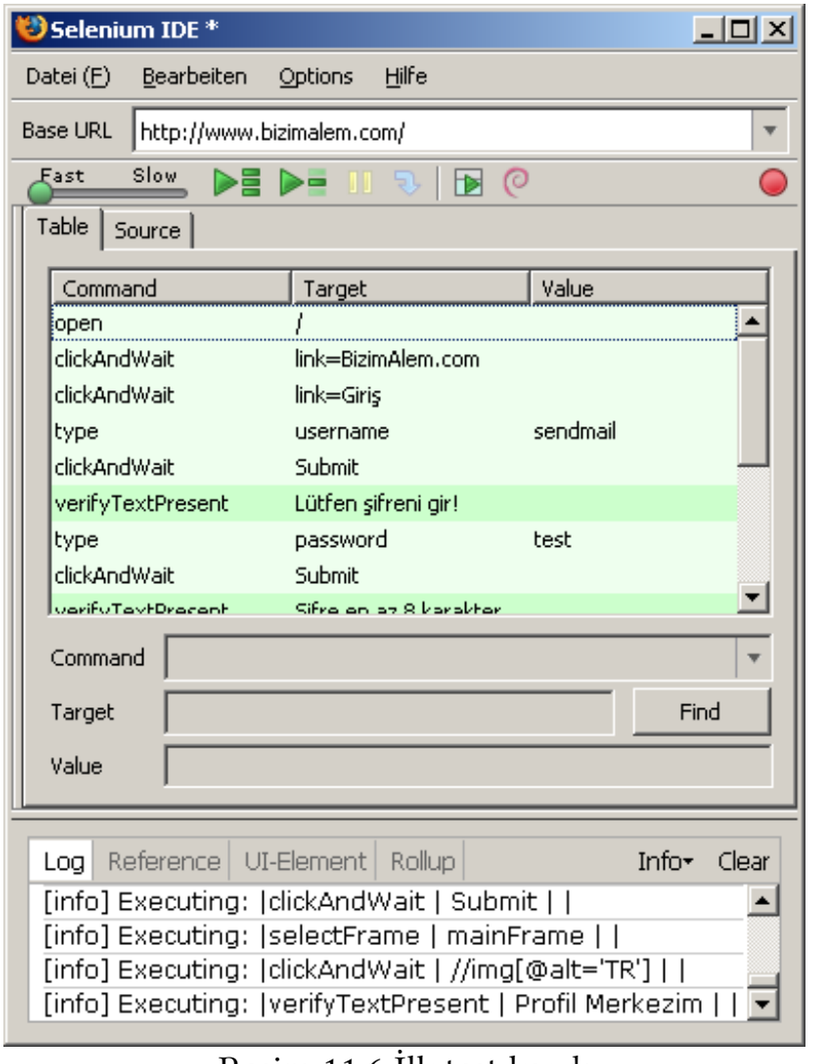


Resim 11.5 Selenium IDE

Selenium IDE resim 11.5 deki haliyle teste hazır durumdadır. Sağ üst bölümde bulunan kırmızı buton kayıtın aktif olduğunu ve Firefox üzerinden yapılan tüm işlemlerin test olarak kayıtlanacağını göstermektedir.

İlk testi kaydetmek için Firefox web tarayıcısına http://www.bizimalem.com adresini giriyoruz. BizimAlem özellikle Avrupa'da yaşayan Türklerin kullandığı bir web community platformudur. Ben ve ekibim tarafından bu kitapta yer alan tekniklerin de kullanılmasıyla oluşturduğumuz bir platformdur. Bir onay/kabul testinin nasıl hazırlandığını BizimAlem login sayfasını test ederek birlikte göreceğiz.

BizimAlem.com adresine bağlandıktan sonra attığımız her adım otomatik olarak Selenium IDE tarafından test komutlarına dönüştürülür. Bir sonraki tabloda BizimAlem e başarılı login yapmak için kullanılan test kaydı yer almaktadır.



Resim 11.6 İlk test kaydı

Kod 11.1 Selenium IDE akseptans test kaydı

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>New Test</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">New Test</td></tr>
</thead><tbody>
<tr>
    <td>open</td>
    <td>/</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
```

```
    <td>link=BizimAlem.com</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Giriş</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>username</td>
    <td>sendmail</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>Submit</td>
    <td></td>
</tr>
<tr>
    <td>verifyTextPresent</td>
    <td>Lütfen şifreni gir!</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>password</td>
    <td>test</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>Submit</td>
    <td></td>
</tr>
<tr>
    <td>verifyTextPresent</td>
    <td>Şifre en az 8 karakterden oluşmalı!</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>password</td>
    <td>sendmail</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>Submit</td>
    <td></td>
</tr>
<tr>
```

```
    <td>selectFrame</td>
    <td>mainFrame</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>//img[@alt='TR']</td>
    <td></td>
</tr>
<tr>
    <td>verifyTextPresent</td>
    <td>Profil Merkezim</td>
    <td></td>
</tr>
</tbody></table>
</body>
</html>
```

Kod 11.1 de yer alan kod Selenium IDE tarafından kayıtlanan onay/kabul testidir. Selenium testleri HTML formatında oluşturur. Görüldüğü gibi Selenium IDE web tarayıcısı üzerinde yaptığımız her işlemi adım adım kaydetmiştir. Play tuşuna tıklayarak, attığımız adımların otomatik olarak Selenium IDE tarafından tekrarlanmasını sağlayabiliriz. Bu durumda Selenium Firefox web tarayıcısını kullanarak, test bünyesinde kayıtlanmış olan komutları arka arkaya çalıştırır. Bir kere kayıtlanan test, otomatik olarak çalıştırılabilir.

Options > Format menüsünden HTML kodunu Java koduna dönüştürebiliriz. Kod 11.1 yer alan HTML testin Java karşılığı kod 11.2 de yer almaktadır.

```
Kod 11.2   Selenium Java test sınıfı

package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase
{
    public void setUp() throws Exception
    {
        setUp("http://www.bizimalem.com/", "*chrome");
    }

    public void testNew() throws Exception
    {
        selenium.open("/");
```

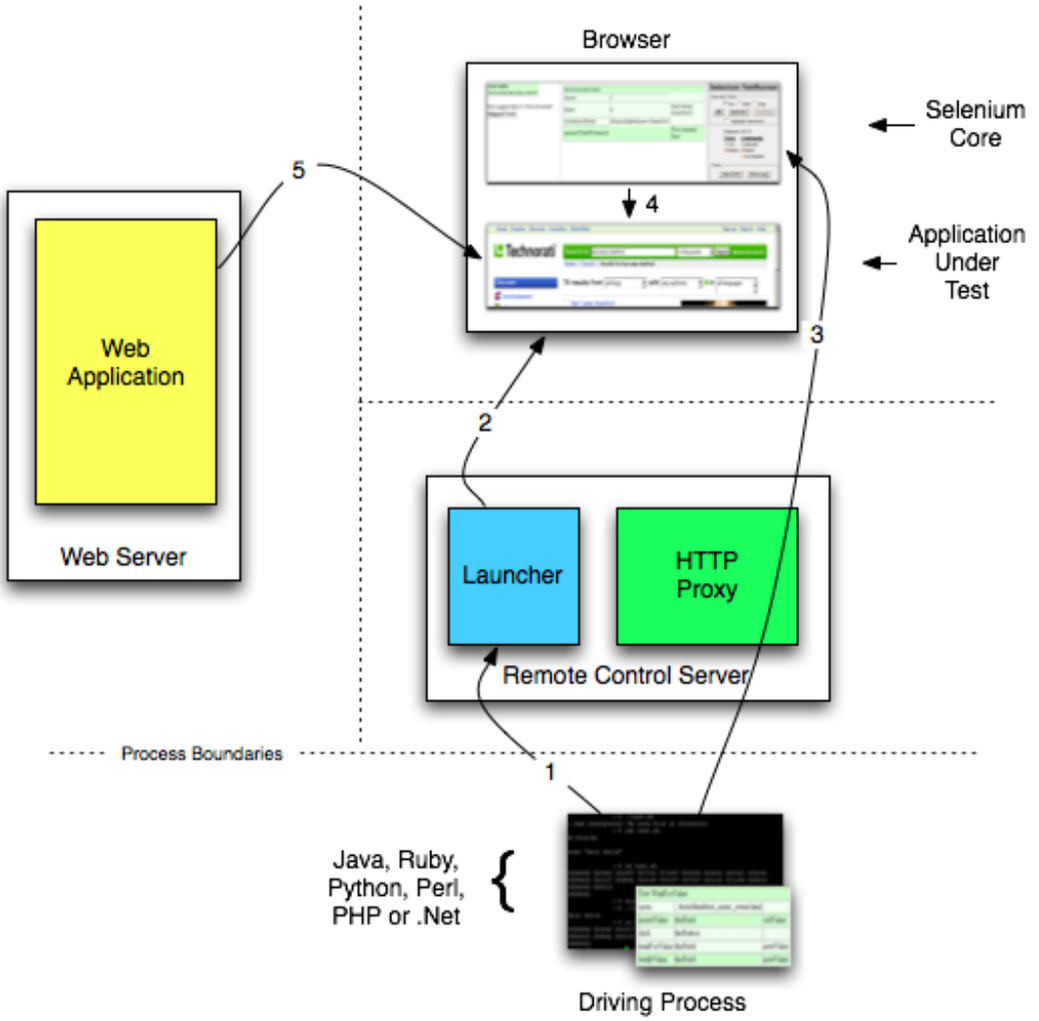
```
        selenium.click("link=BizimAlem.com");
        selenium.waitForPageToLoad("30000");
        selenium.click("link=Giriş");
        selenium.waitForPageToLoad("30000");
        selenium.type("username", "sendmail");
        selenium.click("Submit");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Lütfen şifreni gir!"));
        selenium.type("password", "test");
        selenium.click("Submit");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Şifre en az 8
                                    karakterden oluşmalı!"));
        selenium.type("password", "Nergisim1224");
        selenium.click("Submit");
        selenium.waitForPageToLoad("30000");
        selenium.selectFrame("mainFrame");
        selenium.click("//img[@alt='TR']");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Profil Merkezim"));
    }
}
```

Bu kod ne yazık ki Eclipse altında çalışır durumda değildir. Sebebini şöyle açıklayabiliriz: Selenium ile kaydedilen testlerin çalışabilmesi için Windows gibi bir kullanıcı arayüzü olan işletim sistemine ve Firefox gibi bir web tarayıcısına ihtiyaç duyulmaktadır. Testler web tarayıcısı içinde çalıştırılır. Bu sebepten dolayı Selenium IDE sinde Java koduna dönüştürdüğümüz testleri JUnit olarak hemen çalıştıramayız. Eğer Selenium testlerini JUnit olarak Eclipse altında çalıştırmak istiyorsak, Selenium tarafından kaydedilen komutların bir web tarayıcısına gönderilmesi ve web tarayıcısından gelen sonuçların değerlendirilmesi gerekmektedir, yani JUnit test ile web tarayıcısı arasında bir programın aracılık yapması gerekmektedir. Bu işlemi Selenium RC (Remote Control) programı gerçekleştirir.

Selenium Remote Control (RC)

Selenium RC ile Java, Perl, Php ve Ruby gibi dillerde yazılmış testler Selenium Remote Server komponenti tarafından aktif hale getirilen ve kontrol edilen bir web tarayıcısı içinde çalıştırılır. Test sınıfında yer alan komutlar birbiri ardına Selenium Remote Server tarafından web tarayıcısına gönderilir. Selenium sunucusu bu web tarayıcısıyla bağlantıyı AJAX (XmlHttpRequest) üzerinden gerçekleştirir. Test sınıfında yer alan komutlar HTTP GET/POST metotlarıyla

Selenium sunucusuna gönderilir. Selenium sunucusunu bir HTTP proxy olarak düşünebiliriz. Bizim testler aracılığıyla Selenium sunucusuna gönderdiğimiz komutlar Selenium sunucusu tarafından AJAX teknolojisi kullanılarak web tarayıcısına gönderilir.



Resim 11.7 Selenium RC çalışma modeli

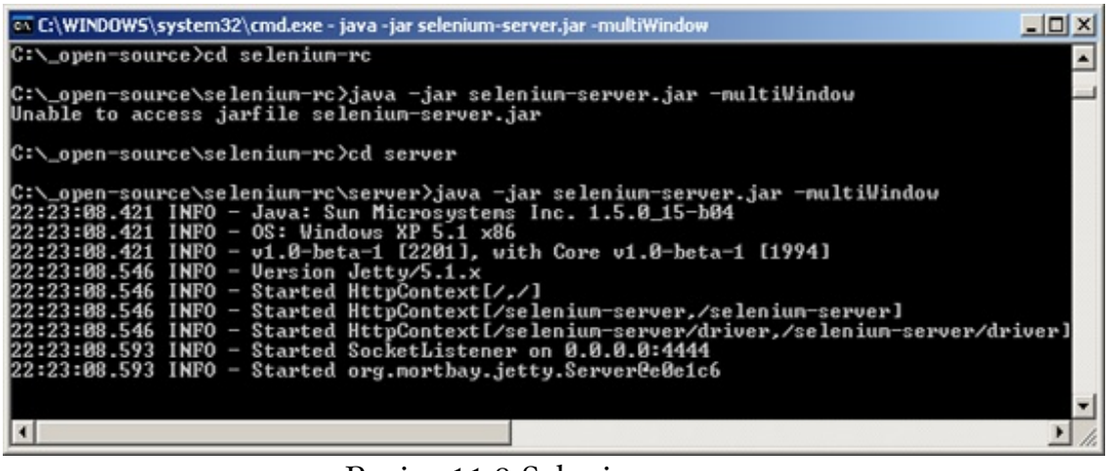
Java dilinde hazırladığımız bir Selenium onay/kabul testini Selenium sunucusu aracılığıyla çalıştırdığımızda aşağıdaki işlemler gerçekleşir.

1. Test içinden Selenium sunucuza bağlantı kurmak için Selenium sunucu client sınıfları (driver) devreye girer.
2. Test içinde kullandığımız URL adresi ile Selenium sunucusu bir web tarayıcısını başlatır. Selenium core ana sayfası yüklenir.
3. Selenium sunucusu test komutlarını web tarayıcısı içinde çalışan Selenium core programına gönderir. Bu komutlar test içinden Selenium sunucu client sınıflarınca Selenium sunucusuna HTTP GET/POST metotlarıyla gönderilmiştir.

4. Selenium core kendisine gönderilen komutları çalıştırır ve neticesini Selenium sunucusuna geri gönderir. Selenium sunucusu ve web tarayıcısı içinde çalışan Selenium core programı arasındaki bağlantı AJAX (XmlHttpRequest) üzerinden gerçekleşir.
5. Selenium sunucusu komutların sonuçlarını test sınıfına HTTP GET/POST aracılığıyla geri gönderir.

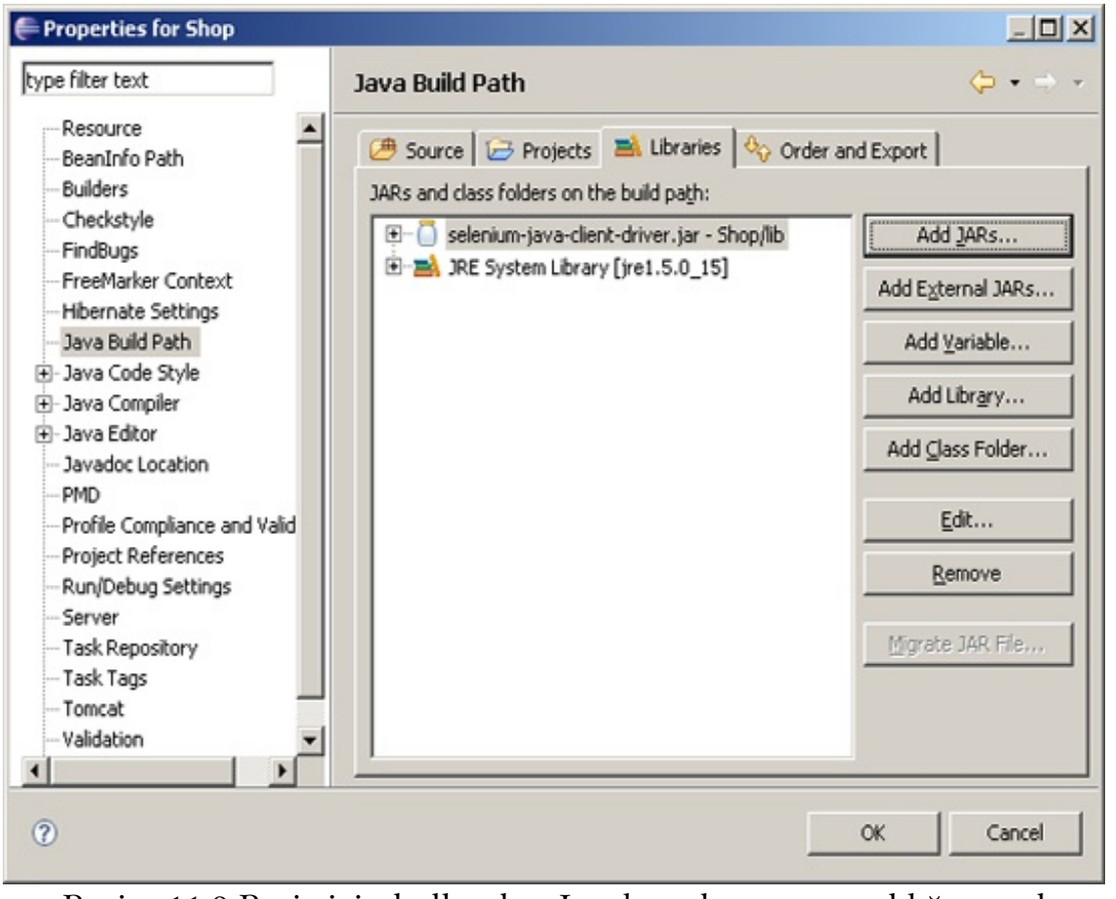
Selenium RC sunucu (server) ve kullanıcı (client) programlarından oluşmaktadır. Güncel sürümü http://docs.seleniumhq.org/download/ adresinden temin edebilirsiniz. Selenium sunucusunu şu şekilde çalıştırabiliriz:

```
java -jar selenium-server.jar -multiWindow
```



Resim 11.8 Selenium server

Kod 11.2 yer alan Selenium Java testini derleyebilmek için selenium-java-client-driver-1.0-beta-1 dizininde yer alan selenium-java-client-driver.jar dosyasını projenin classpath değişkenine eklememiz gerekiyor. Sadece bu durumda Eclipse gerekli Selenium client sınıflarını bularak, test sınıfını derleyebilir. Proje altında lib isminde bir dizin oluşturarak, gerekli Jar dosyasına bu dizine kopyalıyoruz. Bu Jar dosyasını Eclipse e tanıtmak ve proje bünyesinde kullanmak için resim 11.9 da yer alan işlemi yapıyoruz. Ayrıca junit.jar dosyasınında projenin lib dizinine eklenmesi gerekmektedir.



Resim 11.9 Proje için kullanılan Jar dosyalarının yer aldığı panel

Resim 11.10 Selenium sunucusu tarafından otomatik olarak çalıştırılan Firefox web tarayıcısı

Eclipse altında kod 11.2 yer alan Java test sınıfını çalıştırdığımız taktirde Selenium sunucusu resim 11.10 görüldüğü gibi Firefox tarayıcısı çalıştırır. Testin çalışabilmesi için Selenium sunucusunun çalışır durumda olması gerekiyor (resim 11.8). Sağ üst panelde Selenium sunucusu tarafından web tarayıcısına gönderilen komutları görüyoruz. Bu komutlar aslında Java test sınıfında tanımlanan komutlardır. Bu komutlar Selenium client sınıfları aracılığıyla Selenium sunucusuna ve oradan da web tarayıcısına gönderilir. Sonuçlar tekrar Selenium sunucusu tarafından Java test sınıfına aktarılır. Bu şekilde JUnit benzeri bir Java sınıfından bir web arayüzünü test etmek mümkün olmaktadır.

Selenium Ant Entegrasyonu

Oluşturduğumuz Selenium testleri Ant aracılığıyla otomatik olarak çalıştırabiliriz. Bir sonraki tabloda tipik bir Selenium Ant entegrasyon örneğini görmekteyiz.

Kod 11.3 Ant build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Run Test" default="run_test" basedir=".">

    <property name="test.dir" value="src\test" />

    <property name="testLibDir" value="lib" />

    <path id="run.cp">
        <pathelement path="build"/>
        <fileset dir="build/">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="lib"/>
        <fileset dir="lib/">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="run_test" description="Start Proxy ;
                                Run TestNG ; stop Proxy">
        <parallel>
            <antcall target="start-server"></antcall>
            <sequential>
                <echo taskname="waitfor" message="Wait for
                            proxy server launch" />
                <waitfor maxwait="2" maxwaitunit="minute"
                                checkevery="100">
                    <http url="http://localhost:4444/selenium-server/
                                driver/?cmd=testComplete"/>
                </waitfor>
                    <antcall target="run_testNG"></antcall>
                    <antcall target="stop-server"></antcall>
            </sequential>
        </parallel>
    </target>

    <target name="run_testNG" description="Run TestNG">
        <testng classpathref="run.cp" haltOnfailure="false">
            <xmlfileset dir="." includes="testng.xml" />
        </testng>
    </target>
```

```
    <target name="start-server">
        <java jar="lib/selenium-server.jar" fork="true"
                                spawn="true">
            <arg line="-timeout 30"/>
            <jvmarg value="-Dhttp.proxyHost=proxy.corporate.com"/>
            <jvmarg value="-Dhttp.proxyPort=3128"/>
        </java>
    </target>

    <target name="stop-server">
        <get taskname="selenium-shutdown"
            src="http://localhost:4444/selenium-server/
                        driver/?cmd=shutDown"
            dest="result.txt" ignoreerrors="true" />
        <echo taskname="selenium-shutdown"
            message="DGF Errors during shutdown are expected" />
    </target>

    <taskdef resource="testngtasks"
            classpath="lib/testng-5.0-jdk15.jar" />

</project>
```

WebTest

Selenium ile bir webtabanlı uygulamayı test edebileceğimizi gördük. Web tabanlı uygulamaları test etmek için kullanabileceğimiz diğer bir platform WebTest dir. Güncel sürümünü bu adresten temin edebilirsiniz.

WebTest ile onay/kabul testleri XML formatında olup, Ant skriptleri gibi oluşturulur. Hazırlanan testler Ant ya da Windows Console altında çalıştırılabilir. Ant ile entegre olmasından dolayı WebTest sürekli entegrasyon süreci için daha kullanışlıdır. Selenium gibi bir grafiksel arayüze ve web tarayıcısına ihtiyaç duymamaktadır. Kod 11.4 de basit bir WebTest testi yer almaktadır.

Kod 11.4 WebTest test skript

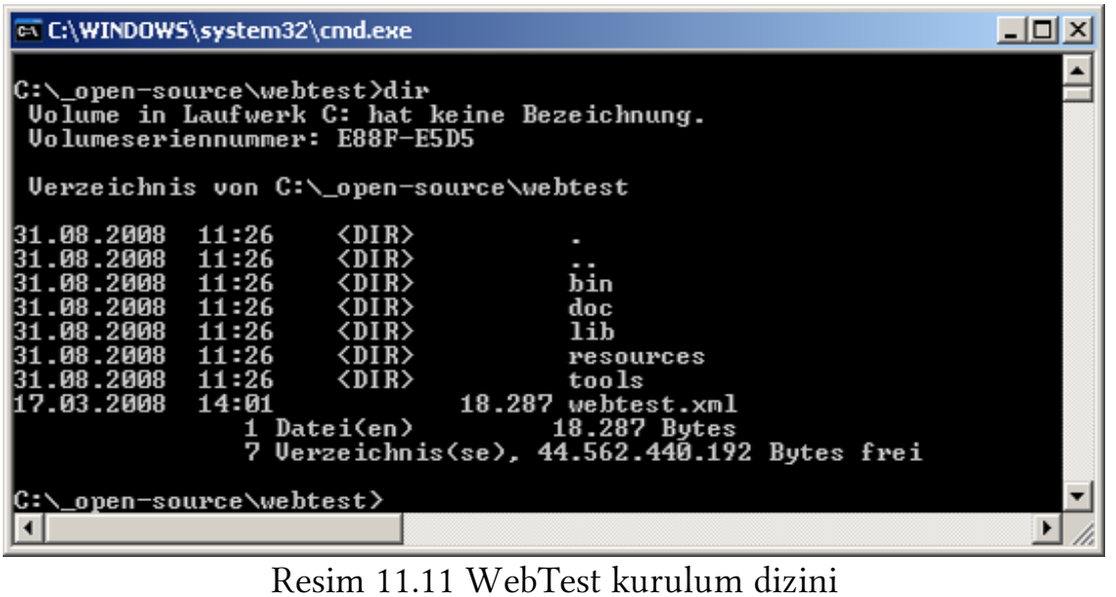
```
<target name="login" >
  <webtest name="normal" >
    &config;
    <steps>
      <invoke description="get Login Page"  url="login.jsp"   />
      <verifyTitle   description="we should see the login title"
```

```
                    text="Login Page"/>
      <setInputField description="set user name"  name="username"
                    value="scott"/>
      <setInputField description="set password"  name="password"
                    value="tiger"/>
      <clickButton   description="Click the submit button"
                    label="let me in" />
      <verifyTitle   description="Home Page follows if login ok"
                    text="Home Page"/>
    </steps>
  </webtest>
</target>
```

WebTest ile websayfasında yapılmak istenen işlemler steps komutu bünyesinde guruplanır. Değişik işlemler için değişik tipte ve isimde komutlar tanımlanmıştır. Örneğin login isminin bulunduğu bir alana bir isim yazabilmek için setInputFiled komutu kullanılır. Bir butona tıklamak için clickButon komutu mevcuttur. Bunlar gibi WebTest yüze yakın komut ihtiva etmektedir.

WebTest Kurulumu

WebTest i kullanabilmek için en az JDK 1.5 ve Ant 1.6.5 e ihtiyaç duyulmaktadır. WebTest download sayfasından güncel sürümü indirdikten sonra istediğiniz bir dizin içine kurulumu gerçekleştirin. Dizin yapısı resim 11.11 da göründüğü şekilde olacaktır.



Resim 11.11 WebTest kurulum dizini

Kurulumu kontrol etmek için doc/sample dizinde bulunan installTest.xml testini kullanabiliriz.

Kod 11.5 installTest.xml

```
<project name="InstallationCheck" basedir="." default="all">

    <property name="webtest.home" value="${basedir}/../.."/>
    <import file="${webtest.home}/webtest.xml"/>

    <target name="all" depends="mayPrintANTError, checkWebTest"/>

    <target name="checkWebTest" depends="wt.defineTasks">
        <echo message="webtest.home is ${webtest.home}"/>
        <webtest name="check calling and parsing a local file">
            <config
                host=""
                port="0"
                basepath="/"
                summary="false"
                saveresponse="false"
                haltonfailure="true"
                protocol="file"/>
            <steps>
                <invoke
                    description="get local file"
                    url="${basedir}/testfile.html"/>
                <verifyTitle
                    description="check the title is parsed correctly"
                    text="Test File Title"/>
            </steps>
        </webtest>
    </target>

    <target name="checkANT">
        <available classname="org.apache.tools.ant.ProjectComponent"
            property="ant.version.ok"/>

    </target>

    <target name="mayPrintANTError" unless="ant.version.ok"
                            depends="checkANT">
        <echo message="You have a non-compliant version of ANT"/>
        <echo message="Consider moving WEBTESTHOME/lib/ant.jar"/>
        <echo message="to ANT_HOME/lib."/>
    </target>

</project>
```

Testin başlangıcında webtest.xml dosyasını import etmemiz gerekiyor. Bu dosya

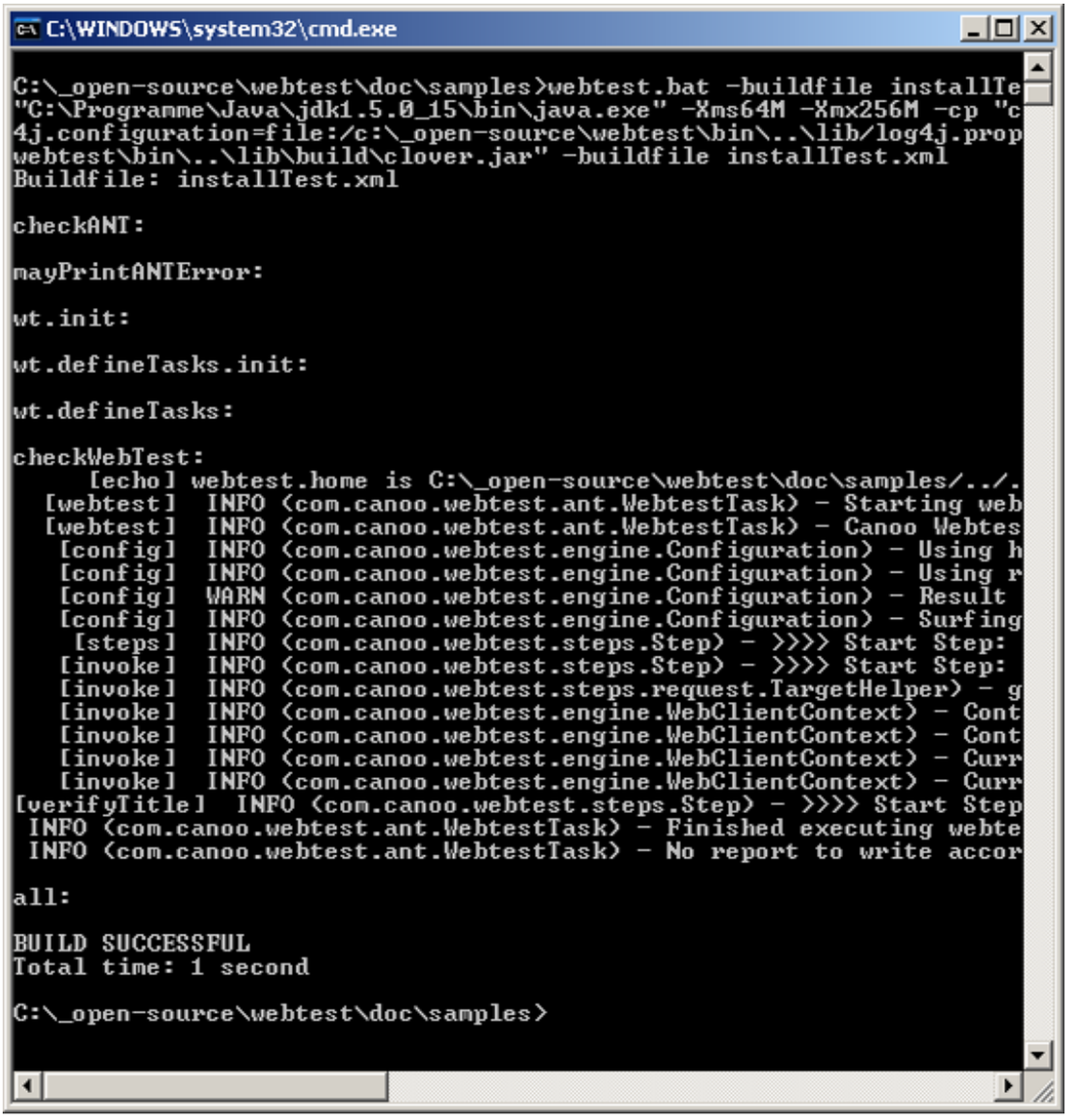
içinde WebTest için kullanılan Ant Task lar tanımlanmaktadır. Testin başlangıç noktası all olarak tanımlanan hedeftir (target). Bu hedef ilk önce mayPrintANTError isimli hedefi kullanarak, sistemdeki Ant sürümünü kontrol eder.

```
<target name="checkANT">
    <available classname="org.apache.tools.ant.ProjectComponent"
        property="ant.version.ok"/>
</target>
```

Ant ın sistemde bulunamaması durumunda test yarıda bırakılarak hata oluştuğu bildirilir. Bunun ardından gerçek test checkWebTest hedefiyle başlar. Bu hedef bünyesinde WebTest komutları kullanılarak, installText.xml ile aynı dizinde olan testfile.html dosyası üzerinde işlemler yapılır.

Bu testi şu şekilde Windows Console altında çalıştırabiliriz:

```
bin/webtest.bat -buildfile installTest.xml
```



Resim 11.12 WebTest selftest

Resim 11.12 de yer aldığı gibi WebTest test XML dosyaları Ant skriptleri gibi çalışır. BUILD SUCCESSFUL mesajı ile testin başarıyla tamamlandığını görüyoruz.

Sürekli Entegrasyon ve WebTest

CruiseControl gibi sürekli entegrasyon için kullanılan programlar Ant yapılandırma skriplerini (build.xml) kullanır. Uygulamanın derlenmesi, yapılandırılması ve JUnit testlerinin çalıştırılması Ant skripleri ile otomatize edilmek zorundadır.

WebTest altyapı olarak Ant ı kullandığı için testlerin Ant skriptlerine entegrasyonu çok kolaydır. Bunun için Ant skript dosyasının (build.xml)

başında webtest.xml ve taskdef.xml dosyalarının import edilmesi gerekmektedir. Bu iki dosya içinde WebTest in Ant skriplerini kullanılan Task lar tanımlanmıştır.

Kod 11.6 build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Shop" default="TESTweb">

     <property file="ant.properties"/>
     <import file="${webtest.home}/lib/taskdef.xml"/>
     <import file="${webtest.home}/webtest.xml"/>

   <target name="TESTweb" depends="init">

           <antcall target="compile">
             <param name="destination"
                value="${base.web.web-inf.classes}" />
             <param name="cleaned" value="true" />
            <param name="stage" value=""/>
    </antcall>

    <antcall target="tomcat-start" />
    <antcall target="hsqldb-start" />
    <sleep seconds="10"/>
    <ant antfile="${web.test.alltests.xml}" />
    <antcall target="tomcat-stop" />
    <antcall target="hsqldb-stop" />
    <formatResults/>

    </target>
</project>
```

12. Bölüm

Spring Çatısı

Bu bölüm Pratik Spring isimli kitabımdan alıntıdır.

Spring Java dünyasında yazılım geliştirmeyi basitleştirmek için geliştirilmiş bir yazılım çatısıdır (framework). Spring'i diğer çatılardan ayıran en büyük özellik temellerinin dependency injection, yani bağımlılıkların enjekte edilmesi prensibine ve AOP'ye (Aspect Oriented Programming) dayanmasıdır. Kitabın ikinci bölümünde dependency injection ve dokuzuncu bölümünde AOP konusunu detaylı olarak inceleyeceğiz.

İki binli yılların başlarında kullanıma sunulan J2EE (Java Enterprise Edition) ve EJB (Java Enterprise Bean) teknolojileri Java ile kurumsal bazlı uygulamaların geliştirilmesini amaçlamaktaydı. Nitekim çok kısa bir zamanda Java kurumsal projelerde kullanılan popüler bir teknoloji haline geldi. Bunun başlıca sebeplerinden birisi de IBM gibi büyük firmaların Java'ya verdiği destektir.

Java 1996 yılında ilk sürümü ile sadece nesneye yönelik bir programla diliyken, J2EE ve EJB teknolojileri ile birlikte büyük bir açık kaynaklı yazılım camiasını da (open source community) kapsayan bir teknoloji platformu haline geldi. Büyük bir yazılım ekosistemine sahip olan Java platformu günümüzde de kurumsal projelerde kullanılan en popüler teknoloji platformudur.

Aklınıza bu kadar geniş bir kapsama alanına sahip bir teknolojinin yanında nasıl olur da Spring gibi popüler bir çatı var olabilir sorusu gelebilir. Öncelikle şunu belirtelim: JEE (Java Enterprise Edition) Spring, Spring'de JEE değildir. Her ikisi de Java dilini kullanarak uygulama geliştirmek için geliştirilmiş teknoloji platformlarıdır. Spring var olma nedenini J2EE'ye borçlu. Bunu açıklayalım.

İki binli yılların başlarında EJB 1.x ve 2.x teknolojileri ile yazılım geliştirmiş olanlar çok iyi bilirler. Bu teknolojileri kullanarak yazılım yazmak kadar programcı için zahmetli bir uğraşı yoktur. EJB yazılım geliştirme modeli bir uygulama sunucusunun kullanımını gerektirmektedir. Geliştirilen EJB uygulamalarının çalışabilmeleri için JBoss, Weblogic ya da Glassfish gibi uygulama sunucusuna ihtiyaç duyulmaktadır. Bu ilk bakışta kötü bir şey değil. Uygulama sunucuları EJB uygulamaları için ihtiyaç duydukları transaksiyon yönetimi ya da güvenlik gibi her türlü servisi sunmaktadırlar. Programcı için bu servisleri bedel ödemeden almak, onun işletme mantığına konsantre olmasını sağlamaktadır. Ama uygulama sunucusundan bu hizmetleri alabilmek için işletme mantığının da belli bir yapıda kodlanmış ve konfigüre edilmiş olması

gerekmektedir. Bu en azından EJB 3 öncesi programcıların başını ağrıtan bir durum idi. Bunun yanı sıra EJB modüllerini uygulama sunucusu dışında test etmek mümkün değildi. Ben o tarihlerde çalıştığım kurumsal projelerde çok iyi hatırlıyorum. Günün hatırı sayılır bir bölümünü uygulamayı test edebilmek için uygulama sunucularını çalıştırıp, durdurmakla geçiririrdim. Uygulama sunucusu çalışmıyorsa, işletme mantığını test etmek imkansızdı. Bir EJB 2 uygulamasını uygulama sunucusu içinde tam çalışır hale getirmek en kötü şartlarda on ya da on beş dakika sürebilir. Artık gerisini siz düşünün.

EJB 2 teknolojisi sağladığı imkanlarla kurumsal gereksinimlerin hakkını verebilecek uygulamalar geliştirmeyi mümkün kılsa da, hantallığından dolayı birçok programcının tabiri caizse nefret ettiği bir teknoloji olarak tarihe geçmiştir. Bu Spring çatışının ortaya çıkmasına sebep olmuştur.

Rod Johnson 2002 yılında kaleme aldığı **Expert One-on-One: J2EE Design and Development** isimli kitabında Interface 21 adını taşıyan altyapı ile nasıl daha hızlı ve kolay kurumsal projelerin geliştirilebileceğini anlatıyor. Interface 21 daha sonra açık kaynaklı yazılıma dönüştürüldü ve bugün tanıdığımız Spring çatısının temellerini oluşturdu.

Rod Johnson'un yazılım yaparken önerdiği temel prensip EJB'ler yerine sade Java nesnelerin (POJO; Plain Old Java Object) kullanımı. Buradan yola çıkarak Spring için şunu söyleyebiliriz: Zaten yapıları itibariyle karmaşık olan kurumsal projeler EJB 2 gibi teknolojiler kullanıldığında daha da karmaşık hale gelmektedir. Spring sunduğu POJO tabanlı programlama modeli ile kurumsal projelerin daha hızlı gerçekleştirilmelerini ve yazılımcıların verimliliğini artırmayı hedeflemektedir. **Kısaca Spring Java ile yapılan yazılımı basitleştirmek ve sadeleştirmek için vardır.**

Spring Filozofisi

Özellikle nesneye yönelik programlama teknikleri kullanıldığında, nesneler arasında var alan bağımlılıklar çok karmaşık bir yapının oluşmasına neden olabilmektedir. Uygulama geliştirme esnasında bağımlılıkların kontrol altına alınmasına dair bir çalışma yapılmadığı taktirde, yazılımcının verimliliği ve uygulamanın kod kalitesi düşecektir. Kaliteyi artırmanın ve yazılımcının daha verimli olmasını sağlamanın bir yöntemi, tüm bağımlılıkların ve oluşan karmaşık yapının dış bir uygulama çatışı (framework) tarafından yönetilmesini sağlamak olabilir. Bu bağımlılıkların uygulama tarafından değil, kullanılan

uygulama çatışı tarafından yönetilmesi anlamana gelmektir. Bu yazılım filozofisine kontrolün tersine çevrilmesi ya da **Inversion of Control (IoC)** ismi verilmektedir. Spring çatısının var oluşu ve çalışma prensipleri bu filozofiye dayanmaktadır.

Dependency Injection

Java gibi nesneye yönelik bir programlama dili ile geliştirilen uygulamalar ideal şartlarda kodun tekrar kullanıldığı modüler bir yapıdadır. Modüller birbirlerini kullanarak, yapmaları gereken işlemleri gerçekleştirirler. Bu modüller arası bağımlılıkların oluşmasını sağlar.

```
class RentalController {
    private RentalService service = new RentalServiceImpl();
}
```

Yukarıda yer alan RentalController sınıfı/modülü bunun güzel bir örneğini teşkil etmektedir. RentalController sınıfı RentalService interface sınıfına bağımlıdır. Böyle bir bağımlılık modüler bir yapının oluşturulması ve kodun tekrar kullanımını sağlamak açısından zaruridir. Lakin RentalController bünyesinde somut bir implementasyon sınıfı olan RentalServiceImpl sınıfından new operatörü ile yeni bir nesne oluşturulması, mevcut bağımlılığın değiştirilemez ve tek bir tipte olması gerektiği anlamına gelmektedir. Bağımlılığı yeniden yapılandırabilmek için kodu değiştirmek ve yeniden derlemek gerekmektedir. Bağımlılıklarını kendisi yöneten bir uygulamada kod kalitesini düşüren bu tür bağımlılıkların oluşturulmasıdır. Oysaki bağımlılıkların tersine çevrilmesi prensibine (DIP; Dependendy Inversion Principle) göre bağımlığın yönü somut değil, soyut sınıflara doğru olmalıdır.

```
class RentalController {
    private RentalService service = rentalServiceFactory.instance();
}
```

Somut bir sınıfa olan bağımlılığı yok etmek için rentalServiceFactory gibi bir fabrika (factory) sınıfından faydalanabiliriz. Fabrika tasarım şablonunu simgeleyen rentalServiceFactory bünyesinde hangi somut RentalService implementasyonunun kullanıldığını gizlemekte ve RentalController sınıfını bahsettiğim somut bağımlılıktan kurtarmaktadır. Lakin buradaki sorun rentalServiceFactory nesnesine olan bağımlılıktır. Bu nesnenin de bir şekilde new operatörü ile oluşturulması gerekmektedir.

```
class RentalController {

    private RentalService service;

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

Bağımlılıkları oluşturma işlemi ile hiç uğraşmasak, bunu başka birisi bizim için yapsa nasıl olurdu? Yukarıda yer alan kod örneğinde service değişkenine gerekli değer setService() metodu aracılığı atanmaktadır. Biran için setService() metodunun dış bir mekanizma tarafından koşturulduğunu düşünelim. Bu mekanizma setService() metodunu kullanarak herhangi bir RentalService implementasyonunu RentalController sınıfına enjekte edebilir. Bu işleme bağımlılıkların enjekte edilmesi yani dependency injection (DI) ismi verilmektedir. Bu işlemi yapan da Spring çatışıdır.

Aşağıda tipik bir Spring XML konfigürasyon örneği yer almaktadır. Yönetimi Spring'e devredilen bağımlılıklar için bu tarz konfigürasyon dosyaları oluşturulur. Spring bu konfigürayon dosyalarını kullanarak nesnelere arası gerekli bağımlılıkları oluşturur, yani bağımlılıkları enjekte eder.

```
<bean id="rentalController"
        class="com.kurumsaljava.spring.RentalController>
    <property name="service" ref="rentalService"/>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl/>
```

Bağımlılıkların enjekte edilmesi prensibi ile çok sade yapıda olan sınıflar oluşturabiliriz. Kendi bağımlılıklarını yönetmek zorunda olmayan bir sınıf asıl işi olan işletme mantığına konsantre olabilir. Tek sorumluluk prensibi açışından bakıldığında da bu bir gerekliliktir.

Hollywood Prensibi

VIP (Very Important Person) olan şahıslara erişmek zordur. Onlar genelde "bizi aramayın, biz sizi ararız" şeklinde iletişimi tercih ederler. Holywood prensibi olarak bilinen bu prensibi IoC konseptini açıklamak için kullanabiliriz.

Bağımlılıkların enjekte edilmesi Hollywood prensibine göre çalışmaktadır.

RentalController sınıfı kendi başına bir konstrüktör ya da fabrika metodu koşturarak bir service nesnesi edinmeye çalışmaz. Bunu yapsaydı eğer, o zaman bu VIP şahsı telefonda aramak ve benim service nesnesine ihtiyacım var demek gibi bir şey olurdu. Bunun yerine VIP şahıs, yani Spring RentalController sınıfında yer alan setService() metodunu kullanarak RentalController sınıfıyla iletişime geçmektedir. Spring RentalController sınıfına bir RentalService nesnesi enjekte edebilmek için setService() metodunu koşturmaktadır. Spring sınıfların set() metotlarını ya da konstrüktörlerini kullanarak gerek duyulan bağımlılıkları enjekte etmektedir. Bağımlılığı enjekte edebilmek için bu metotları koşturması, yani sınıfı araması gerekmektedir.

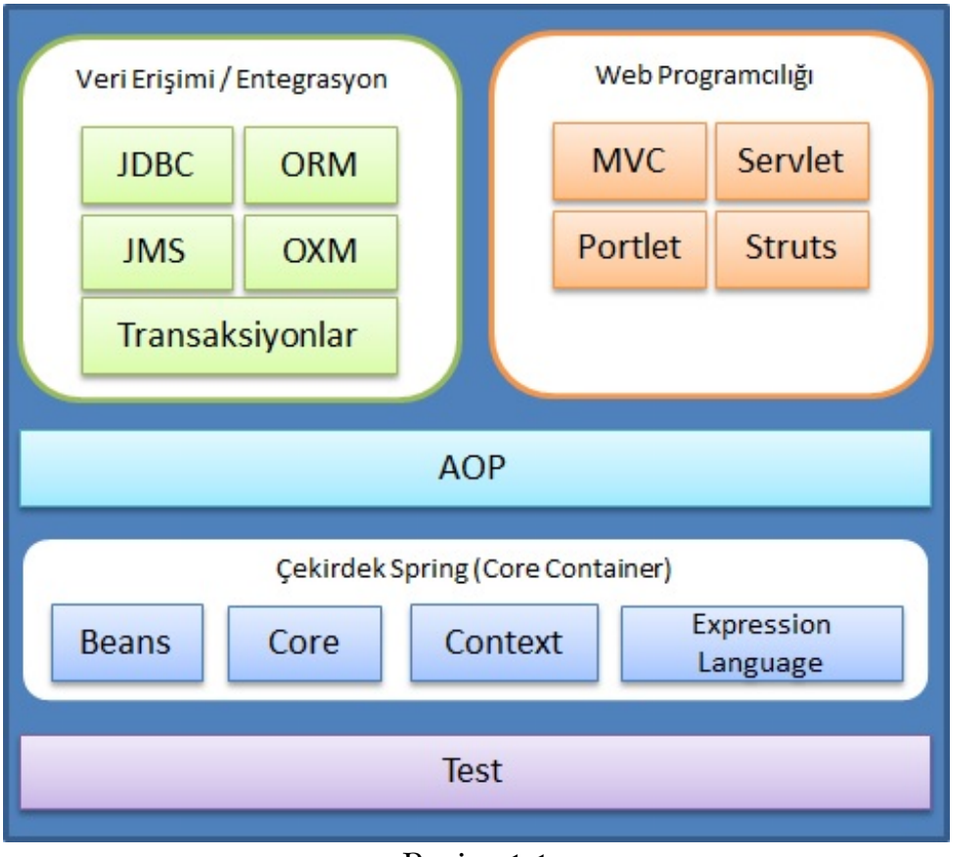
Spring koşturulacak metodun seçiminde konstrüktör ya da set() metoduyla sınırlı değildir. Hollywood prensibi sınıf bünyesinde yer alan sınıf metotları üzerinde de kullanılabilir. Bu Spring'in konfigürasyon dosyasında belirlenen herhangi bir sınıf metodunu koşturulabileceği anlamına gelmektedir. Kitabın on yedinci bölümünde inceleyeceğimiz Spring Task ve Scheduling modülünde herhangi bir POJO (Plain Old Java Object) sınıfın herhangi bir metodunu şu şekilde koşturmak mümkündür:

```
<task:scheduled ref="rentalDownloader"
        method="download" cron="*/5 * 9-17 * * MON-FRI"/>
```

Spring tarafından koşturulması gereken metodun ismi method element özelliğinde yer almaktadır. Görüldüğü gibi rentalDownloader nesnesi görevini yerine getirmek için hangi metodun koşturulması gerektiğini bilme sorumluluğundan arındırılmaktadır. Sadece konfigürasyon dosyası üzerinde değişiklik yaparak, POJO sınıfın çalışma tarzı adapte edilebilmektedir. Spring birçok modülünde bu mekanizmadan faydalanmaktadır.

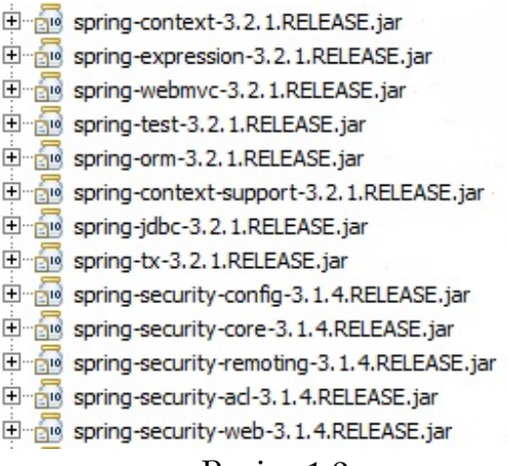
Spring Modülleri

Spring değişik modüllerden oluşan modüler bir yapıya sahiptir. Spring bir kurumsal projeyi gerçekleştirmek için gerekli her şeyi ihtiva etmekle birlikte, değişik Spring modüllerini ihtiyaçlarımız doğrultuşunda seçerek, kullanabiliriz. Spring'in modüler yapısı genel hatlarıyla resim 1.1 de görülmektedir.



Resim 1.1

Spring 3.2.4 sürümü ile yirmi değişik modülden oluşmaktadır. Her modül kendi JAR dosyasında yer almaktadır. Bu modüllerde bazıları resim 1.2 de yer almaktadır.



Resim 1.2

Spring Modülleri İle Neler Yapabiliriz?

Bu bölümde altbaşlıklar halinde Spring modüllerini tanıtmak istiyorum.

Çekirdek Sunucu (Core Container) Modülü

Spring uygulamalarının temelini çekirdek (core) sunucu (container) oluşturmaktadır. Bir Spring uygulaması çalışmaya başladığında Spring tarafından yapılan ilk işlem, içinde Spring nesnelerinin (Spring Bean) yer aldığı ve bu nesneler arasında bağımlılıkların enjekte edilmesini sağlayan bir sunucu (container) oluşturmaktır. Bağımlılıkların enjekte edilmesi için BeanFactory kullanılır. BeanFactory yapısını kitabın üçüncü bölümünde inceleyeceğiz.

Çekirdek sunucu Core, Context, Beans ve Expression Language modüllerinden oluşmaktadır. Bu modüllerin ne olduklarını ve nasıl kullanılabileceklerini kitabın ikinci bölümünden itibaren örnek kodlar üzerinde inceleyeceğiz.

Diğer tüm Spring modülleri çekirdek sunucu üzerinde inşa edilmiş modüllerdir. Her Spring uygulaması mutlaka çekirdek sunucuyu oluşturan modüllerde yer alan sınıflar kullanılarak konfigüre edilir. Spring çekirdek sunucu yapı itibari ile bir EJB uygulama sunucusuna benzer. Bünyesinde yer alan tüm nesnelere konfigürasyonları doğrultusunda ihtiyaç duydukları servisleri sunar. Bunu yaparken AOP teknolojisini kullanır.

Spring AOP Modülü

Spring uygulamalarını POJO sınıfların oluşturduğunu söylemiştik. Bu sınıflar mevcut yapıları itibari ile bir kurumsal projelerin gereksinimlerini tatmin edecek yapıda değildirler. Örneğin EJB komponentler uygulama sunucusu tarafından güvenlik ya da otomatik transaksiyon yönetimi gibi yetilerle donatılırlar. Aynı şey Spring tarafından AOP kullanılarak POJO sınıflar üzerinde gerçekleştirilir. Örneğin sadece işletme mantığını ihtiva eden bir POJO sınıfa AOP kullanılarak transaksiyonel özellik kazandırılabilir. Kitabın dokuzuncu bölümünü AOP konusuna ayırdım.

Veri Erişimi Modülü

JDBC tabanlı veri tabanı işlemleri için Spring bünyesindeki JDBC modülünü kullanabiliriz. JDBC kodu yazmış olanlar bilirler. JDBC checked exception türünü kullandığı için JDBC kodu çok kısa zamanda okunmaz bir hale gelebilir. Spring bünyesinde yer alan JDBC modülü ile çok sade JDBC kodu yazmak mümkündür. Sağladığı DAO (Data Access Object) katmanı ile, veri katmanı ile veri tabanı arasında esnek bir bağın oluşmasını destekler. Kitabın beşinci bölümü bu modülü tanıtmaktadır.

JDBC yerine Hibernate ya da EclipseLink gibi bir ORM (Object Relational

Mapping) teknolojisini tercih edenler Spring ORM modülü ile bu teknolojileri Spring uygulamalarında kullanabilirler. Spring ORM modülü bir ORM çatısı olma iddiasını gütmemektedir. Daha ziyada mevcut ORM teknolojilerini entegre ederek, tek bir programlama modeli üzerinden kullanımlarını sağlamaktadır. Kitabın yedinci bölümünde Spring ile Hibernate, sekizinci bölümünde JPA (Java Persistence API) kullanımını yakından inceleyeceğiz.

Spring OXM (Object XML Mapping) modülü Java<=>XML dönüşümünü sağlamak için kullanabileceğimiz Spring modülüdür. Yine diğer modüllerde de olduğu gibi bir OXM çatısı olma iddiası gütmemektedir. Daha ziyada JAXB, Castor, XMLBEans ve XStream gibi teknolojileri kullanarak Java<=>XML dönüşümünü sağlamaktadır. Kitabın on ikinci bölümününde yer alan REST ve on dördüncü bölümününde yer alan Web Service konularında Spring OXM modülünün kullanımını inceleyeceğiz.

Spring JMS (Java Messaging Service) JMS mesajları oluşturmak (produce) ve tüketmek (consume) için kullanılan modüldür.

Transaksiyon modülü deklaratif ve programsal transaksiyon yönetimi yapılmasını sağlamaktadır. Kitabın altıncı bölümünde Spring ile transaksiyon yönetimini inceleyeceğiz.

Spring MVC Modülü

Spring genelde entegratif bir çatıdır, yani mevcut teknolojileri aynı çatı altında toplayarak, belli bir programlama modeli sunar. Bunun bozulduğu istisnalardan bir tanesi Spring MVC çatısıdır. Bu çatı ile web tabanlı uygulamalar geliştirmek mümkündür. Spring ile Struts ya da Wicket gibi web çatılarını kullanmak mümkün iken, Spring burada kendi web çatısını geliştirmeyi tercih etmiştir. Kitabın onuncu bölümünde Spring MVC web çatısını yakından inceleyeceğiz.

Spring Remoting Modülü

Spring MVC modülü ile web tabanlı uygulamalar geliştirmek mümkün iken, Spring Remoting modülü ile POJO bazlı sınıfları servis sunucusu haline dönüştürmek mümkündür. POJO sınıflar RMI (Remote Method Invocation), Hessian, Burlap, JAX-WS (Web Service) ve HTTP invoker protokol ve teknolojileri kullanılarak servis sunucu haline getirilebilir. Ayrıca Spring Remoting ile mevcut servis sunucuları ile iletişimde kullanılabilecek kullanıcılar (client) oluşturulabilir. Kitabın on üçüncü bölümü bu modülün kullanılış tarzını tanıtmaktadır.

Spring Test Modülü

Yazılımcı olarak benim Spring'in en çok ilgimi çeken tarafı, test güdümlü yazılımı mümkün kılan bir test çatısına sahip olmasıdır. Spring Test JUnit ve TestNG çatılarını kullanarak birim ve entegrasyon testlerinin geliştirilmesini mümkün kılmaktadır. Kitabın on beşinci bölümünde Spring ile test seçeneklerini inceleyeceğiz.

Spring Uygulama Portföyü

Spring çekirdek uygulama çatısı haricinde, bu çekirdek çatı üzerine inşa edilmiş uygulamaları da ihtiva etmektedir. Bu uygulamalardan bazıları:

- ***Spring Integration*** - Spring programlama modelini kurumsal entegrasyon şablonları (Enterprise Integration Patterns) destekleyecek şekilde genişletir. Spring Integration birbirlerinden tamamen bağımsız olan yazılım modüllerinin mesajlaşma (messaging) teknikleri kullanılarak bir uygulama oluşturacak şekilde bir araya getirilmelerini amaçlamaktadır.
- ***Spring Batch*** - Kullanıcı arayüzüne ihtiyaç duymadan seri bir şekilde işlem yapmayı (batch application) mümkün kılar. Özellikle bankalar gibi müşteri işlemlerini mesai saatlerinden sonra topluca yapan kuruluşların bu yöndeki ihtiyaçlarını karşılamak amacıyla geliştirilmiştir.
- ***Spring Social*** - Spring uygulamalarını Facebook, Twitter, ve LinkedIn gibi sosyal medya platformları ile entegre etmek için geliştirilmiş uygulamadır.
- ***Spring Mobile*** - Spring MVC web yazılım çatısını genişleten ve mobil web uygulama yazılımını kolaylaştırmak için kullanılan uygulamadır.
- ***Spring Web Services*** - SOAP bazlı web servis uygulamaları geliştirmek için kullanılmaktadır.
- ***Spring Web Flow*** - Temelinde Spring MVC web çatısını kullanan Web Flow kurumsal bir işlemi birden fazla web sayfasına bölerek, işlemin adım, adım yapılmasını sağlayan uygulamadır. Web Flow sayfalar arası otomatik oturum ve durum yönetimini yapmaktadır.
- ***Spring LDAP*** - Spring bazlı uygulamaları için LDAP (Lightweight Directory Access Protocol) kullanımını basitleştirmektedir.
- ***Spring Security*** - Spring uygulamalarında güvenlik konfigürasyonunu yapmak için kullanılan uygulama modülüdür.
- ***Spring Data*** - Relasyonel veri tabanı sistemleri yanı sıra map-reduce, NOSQL, bulut gibi yeni veri tabanı ve veri erişimi teknolojilerinin

kullanımını kolaylaştırmaktadır. JPA, MongoDB, Neo4j, Redis, Hadoop, Gemfire, Rest, Solr, CouchBase ve ElasticSearch gibi teknolojileri destekleyen alt modülleri mevcuttur.
- ***Spring XD*** - Büyük veri (big data) uygulamalarında import, export, analiz ve batch işleme gibi işlemleri kolaylaştırmak için oluşturulmuş uygulamadır.
- ***Spring Roo*** - Java ile uygulama geliştiren yazılımcılar için hızlı uygulama geliştirme (rapid application development) aracıdır.

Spring 3 İle Gelen Yenilikler

Bu bölümde Spring 3.0, 3.1 ve 3.2 sürümlerinde yer alan yenilikleri sizlerle paylaşmak istiyorum. Bu sürümlerde göze çarpan yenilikler şunlardır:

Spring 3.0

- Bu sürüm ile tüm Spring çatısı Java 5 ile gelen Generics, Varargs ve diğer Java dili yeniliklerini kullanacak şekilde elden geçirilmiştir. Spring'de anotasyon desteği 2.5 sürümü ile gelmiş olsa bile, Spring 3.0 sürümü ile bu destek daha da artırılmış ve JSR-330 ile gelen standart Java anotasyonların kullanımı mümkün hale gelmiştir. Bu şekilde anotasyon bazlı konfigürasyon kullanıldığında standart Java anotasyonları kullanılarak, kod bazındaki Spring çatısına olan bağımlılık ortadan kaldırılabilmektedir.
- Spring uygulamalarında konfigürasyon XML dosyaları üzerinden yapılmaktadır. Spring 3.0 sürümü ile XML dosyası kullanmadan anotasyon bazlı konfigürasyon yapmak mümkün hale gelmiştir.
- Spring 3.0 konfigürasyon imkanlarını daha esnek hale getirmek için Spring Expression Language (SpEL) modülünü ihtiva etmektedir.
- 3.0 sürümü ile Spring MVC uygulamalarını daha kolay konfigüre etmek için yeni XML mvc isim alanı oluşturulmuştur. Yeni eklenen @CookieValue ve @RequestHeaders gibi anotasyonlarla Spring MVC uygulamalarının anotasyon bazlı konfigürasyonu genişletilmiştir.
- Web uygulamaları geliştirmek için kullanılan Spring MVC, 3.0 sürümü ile REST (Representational State Transfer) desteği sağlamaktadır. Spring MVC ile bir REST uygulaması oluşturmak için Spring MVC controller sınıfları kullanılmaktadır. RestTemplate kullanıcı (client) uygulamalar geliştirmek için kullanılmaktadır.
- jdbc isim alanında yer alan embedded-database konfigürasyon elementi ile

HSQL, H2, ve Derby gibi veri tabanı sistemlerinin kullanımı kolaylaştırılmıştır.
- Java EE 6 ile kullanıma sunulan @Asynchronous anotasyonu ile metotlar asenkron koşturulabilmektedir. Spring 3.0 sürümünde yer alan @Async anotasyonu ile bu desteği sağlamaktadır.
- Spring 3.0 JSR-303 (Bean Validation) bünyesinde yer alan anotasyonları desteklemektedir.

Spring 3.1

- Yeni bir caching modülü (Cache Abstraction) ihtiva etmektedir.
- Bu sürümle Spring bean tanımlamalarını profil bazında gruplamak (bean definition profiles) mümkün hale gelmiştir. Profiller yardımı ile uygulama değişik ortamlara göre adapte edilmiş konfigürasyon dosyalarını kullanabilmektedir.
- Oluşturulan yeni Environment isimli sınıf ile profil bazlı bilgilerin yer aldığı yeni bir alan oluşturulmuştur. Bu alan içinde profil bilgileri yanı sıra tanımlanan değişken (property) değerleri de yer almaktadır. Environment sınıfı kullanılarak bu bilgilere ulaşılabilir.
- constructor-arg elementini daha kısa yazmak için c isim alanı oluşturulmuştur. Bunun kullanımını üçüncü bölümde yakından inceleyeceğiz.
- Bu sürüm Hibernate 4.x desteği vermektedir.
- 3.1 sürümü öncesi enjeksiyon için kullanılan set metotlarının void veri tipinde bir değeri geri vermeleri gerekiyordu. Bu yeni sürümle set metotları herhangi bir yapıda olabilmektedir.
- Spring'in test çatısı olan Spring TestContext bünyesindeki @ContextConfiguration anotasyonu @Configuration anotasyonunu taşıyan konfigürasyon sınıflarını desteklemektedir. Ayrıca entegrasyon testlerinde kullanılmak üzere değişik uygulama profillerini destekleyen @ActiveProfiles anotasyonu oluşturulmuştur.
- JPA bünyesinde sınıflar META-INF/persistence.xml dosyasında tanımlanmaktadır. Spring 3.1 sürümü ile gelen LocalContainerEntityManagerFactoryBean ile classpath içinde yer alan sınıflar otomatik olarak taranarak persistence.xml kullanmayan bir JPA altyapısı oluşturulabilmektedir.
- Spring MVC controller sınıflarında kullanılan @RequestMapping anotasyon tanımlaması consumes ve produces elementleri kullanılarak genişletilmiştir. consumes controller sınıfının hangi türde verileri

işleyebileceğini belirlerken, produces kullanıcıya gönderilecek cevabın hangi formatta olması gerektiğini tanımlamaktadır. Böylece örneğin bilgileri XML formatında alan ve kullanıcıya cevabı JSON formatında gönderen controller sınıfları tanımlamak mümkün hale gelmiştir.
- Yeni oluşturulan @RedirectAttributes anotasyonu ile controller metotlarında yönlendirme (redirect) işlemi için parametre tanımlaması yapılabilmektedir.
- @RequestBody anotasyonu kullanılan bir controller metodunda @Valid anotasyonu kullanılarak otomatik validasyon işlemi yapmak mümkün hale gelmiştir.

Spring 3.2

- Spring MVC uygulamalarını uygulama sunucusuna bağımlı olmadan test edebilmek için yeni Spring MVC Test çatısı oluşturulmuştur.
- Bu sürüm Java EE 7'nin bir parçası olan JCache desteği sağlamaktadır.
- Spring MVC, Servlet 3 sürümünde tanımlanan asenkron metot koşturma (asynchronous request processing) özelliğini desteklemektedir.
- RestTemplate HTTP cevaplarında (response) yer alan verileri Java Generics kullanarak (örnegin List) edinebilmektedir.
- Spring bu sürümünde Jackson JSON 2 kütüphanesini ve bir şablon (template) yönetim çatısı olan Tiles 3 sürümünü desteklemektedir.

Spring'in Uygulama Geliştirmedeki Rolü

Spring kurumsal Java projeleri geliştirmek için geniş çaplı altyapısal destek sağlamaktadır. Entegratif yönüyle mevcut Java API (Application Programming Interface) ve çatıların (framework) kullanımını mümkün olduğu kadar tek bir programlama modelinde toplamaktadır. Değişik API ve çatıları tek bir programlama modeli ile kullanabilmek programcıların verimliliğini artıran bir durumdur.

Spring plumbing code olarak isimlendirilen, işletme mantığının mecbur kılınan programlama modeli neticesinde gereksiz kod kalabalığı ile şişmesini sunduğu konfigürayon yöntemleri ile engellemektedir. Böylece POJO sınıflar geliştirerek, sadece işletme mantığına konsantre olmak mümkün hale gelmektedir.

Spring'in çekirdeği uygulama konfigürasyonu, kurumsal entegrasyon, test etme ve veri erişimi konuları için çözümler sunmaktadır. Spring ile bir uygulamayı

değişik modülleri bir araya getirerek oluşturmak mümkündür. Modüllerin birbirlerini bulmaları gerekliliği yoktur. Her modül Spring konfigürasyonu aracılığı ile uygulamanın genel yapısına zarar vermeden başka bir modül ile yer değiştirebilir. Uygulamayı oluşturan modüllerin sessiz, sedasız değiştirilebilir yapıda olması, uygulamanın test edilebilirliğini olumlu etkilemektedir. Sunduğu test imkanları ile Spring uygulamalarını, buna Spring MVC ile oluşturulan web uygulamaları da dahildir, uygulama sunucusu olmadan test etmek mümkündür.

Kaynakları yönetmesi, veri erişimi için kullanılan API'lerin kullanımını kolaylaştıran sınıflar sunması, JDBC, Hibernate, JPA ve IBatis gibi popüler veri erişimi teknolojilerini desteklemesi ile Spring veri erişimi programcılığını kolaylaştırmaktadır.

Spring Struts, Wicket ya da JSF gibi web çatıları ile entegre edilebilmektedir. Bu entegrasyon ile bu çatılarda Spring'in sunduğu uygulama konfigürasyonu modeli kullanılabilmektedir. Bunun yani sıra ihtiva ettiği Spring MVC ve Spring Web Flow çatıları ile web uygulama geliştirmeyi desteklemektedir.

Spring Yazılım Geliştirme Ortamı

Spring STS (SpringSource ToolSuite) ismini taşıyan ve Eclipse bazlı bir yazılım geliştirme ortamına sahiptir.

STS ile tam teşekküllü bir yazılım geliştirme ortamı edinmenin yanı sıra mevcut bir Eclipse Helios (3.6), Indigo (3.7), Juno (3.8/4.2) ya da Keppler 4.3 sürümü Eclipse Marketplace üzerinde STS plugin seti yüklenerek Spring yazılım ortamına dönüştürülebilir.

Resim 1.3

Spring Jar Dosyalarını Nasıl Edinebilirim?

Spring çatısını oluşturan Jar dosyalarını edinmenin en hızlı yolu bir Maven projesi oluşturmak ve aşağıda yer alan Maven bağımlılığını projeye eklemektir. Kitabın her bölümü için oluşturduğum Maven projelerinde bu şekilde gerekli Jar dosyalarını projeye ekledim.

```
Kod 1.1 - pom.xml

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

Eğer Maven kullanmıyorsanız, Spring.io sayfasından güncel Spring sürümünü edinebilirsiniz.

Spring Hello World

Her teknolojiyi öğrenmeye klasik Hello World uygulaması ile başlanır. Bu bölümde Spring ile bu tarz bir uygulamanın nasıl geliştirilebileceğini bir örnek üzerinde göstermek istiyorum.

İlk işlem olarak HelloWorldService isminde bir interface sınıf tanımlıyoruz. Bu interface sınıf kod 1.2 de yer almaktadır. getMessage() metodu istediğimiz türde bir mesajı geri verecektir.

```
Kod 1.2 - HelloWorldService

public interface HelloWorldService {
    String getMessage();
}
```

HelloWorldService sınıfını implemente eden sınıf kod 1.3 de yer almaktadır. getMessage() metodunu Hello World kelimelerini geriye verecek şekilde yapılandırıyoruz.

```
Kod 1.3 - HelloWorldServiceImpl

public class HelloWorldServiceImpl implements HelloWorldService {
    @Override
    public String getMessage() {
        return "Hello World";
    }
}
```

Şimdi HelloWorldService sınıfını kullanan başka bir sınıf tanımlayalım. Kod 1.4 de yer alan MessageManager sınıfı bünyesinde service ismi altında HelloWorldService sınıfını kullanmaktadır. Bir Spring anotasyonu olan @Autowired ile MessageManager sınıfına, daha doğrusu bu sınıftan olan nesneye bir HelloWorldService nesnesi enjekte edilmektedir.

```
Kod 1.4 - MessageManager

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MessageManager {
```

```
    @Autowired
    HelloWorldService service;

    public void printMessage() {
        System.out.println(this.service.getMessage());
    }
}
```

Spring uygulamasını konfigüre etmek ve koşturmak için Application sınıfını (kod 1.5) oluşturuyoruz. @Configuration sınıf bazlı Spring konfigürasyonu kullandığımıza işaret etmektedir. Bu Spring uygulaması sıfır XML konfigürasyonu kullanmaktadır. @ComponentScan ile Spring'e gerekli sınıfları classpath içinde araması gerektiğini belirtiyoruz.

@Bean anotasyonu bir Spring bean tanımlamak için kullanılmaktadır. Kod 1.4 de yer alan MessageManager sınıfına HelloWorldService tipinde bir nesne enjekte edebilmek için Spring'in hangi HelloWorldService implementasyon sınıfının kullanıldığını bilmesi gerekmektedir. Bu uygulamada kullandığımız HelloWorldService implementasyonu kod 1.3 de yer alan HelloWorldServiceImpl sınıfıdır. Spring getMessageService() metodunu yeni bir HelloWorldServiceImpl nesnesi oluşturmak için kullanacak, akabinde bu nesneyi kod 1.4 de yer alan service değişkenine enjekte edecektir, çünkü bu değişken @Autowired ile işaretlenmiştir.

```
Kod 1.5 - Application

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
                AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class Application {

    @Bean
    public HelloWorldService getMessageService() {
        return new HelloWorldServiceImpl();
    }

    public static void main(final String[] args) {
        final ApplicationContext context =
             new AnnotationConfigApplicationContext(
```

```
                                Application.class);
        final MessageManager manager =
                context.getBean(MessageManager.class);
        manager.printMessage();
    }
}
```

Kod 1.5 de yer alan Application sınıfı hem uygulamayı konfigüre etmek, hem de koşturmak için kullanılmaktadır. @Configuration, @ComponentScan ve @Bean anotasyonları uygulamayı konfigüre etmek için kullanılırken, main() metodu uygulamayı koşturmaktadır. main() metodu bünyesinde yeni bir AnnotationConfigApplicationContext nesnesi oluşturulmaktadır. Bu nesne konfigüre ettiğimiz Spring uygulamasını temsil etmektedir. Bu nesne üzerinden getBean() metodu aracılığı ile bir MessageManager nesnesi edinebiliriz. manager.printMessage() metodu koşturulduğunda ekranda Hello World kelimeleri yer alacaktır.

13. Bölüm

Spring MVC

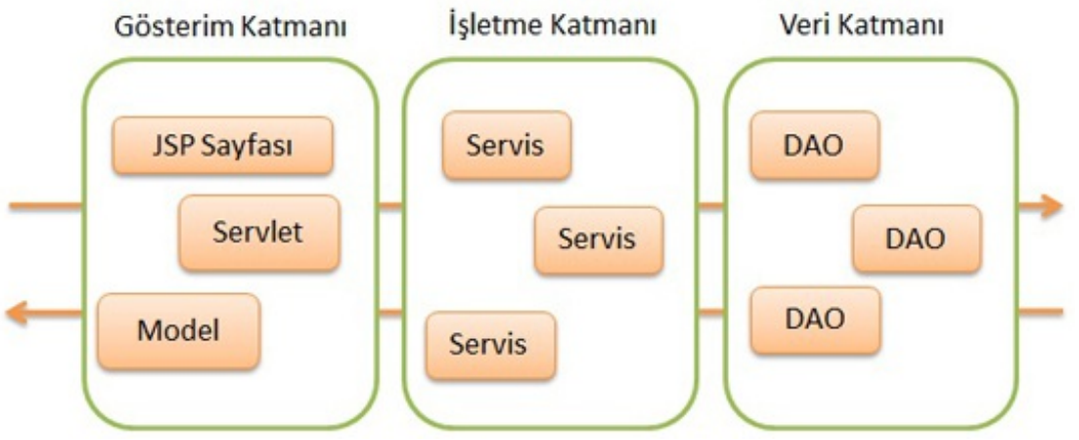
Bu bölüm Pratik Spring isimli kitabımdan alıntıdır.

Java dünyasında web uygulamaları geliştirirken kullanılabilecek en temel çatı (framework) servlet çatısıdır. Kod 10.1 HelloWorldServlet ismini taşıyan bir servlet örneği yer almaktadır. Bu servlet web tarayıcısında "Hello World" cümlesinin görünmesini sağlamaktadır. HelloWorldServlet sınıfını yakından incelediğimizde, çoğu servlet sınıfında uygulanan bir yöntemle karşılaşmaktayız: iş mantığı ile HTML kodunun (örneğin <h1>) bir servlet sınıfı bünyesinde iç içe geçmiş durumda olması.

```
Kod 10.1 - HelloWorldServlet

public class HelloWorldServlet extends HttpServlet {

    private String message;

    @Override
    public void init() throws ServletException {
        message = "Hello World";
    }

    @Override
    public void doGet(final HttpServletRequest request,
            final HttpServletResponse response)
                throws ServletException, IOException {
        response.setContentType("text/html");
        final PrintWriter out = response.getWriter();
        out.println("<h1>" + message + "</h1>");
    }
}
```

İş mantığı ile HTML kodunun bir Java sınıfı bünyesinde birlikte yer almaları, böyle bir sınıfın bakımını zorlaştıran bir durumdur. Uygulamanın web arayüzü değiştirilmek istendiğinde, HTML kodunu bünyesinde taşıyan Java sınıflarının değiştirilmesi gerekir. Bu yapıya sahip servlet tabanlı web uygulamalarının bakım ve geliştirme maliyetlerinin ne kadar yüksek olabileceklerini tahmin edebilirsiniz.

Bu sorunu gidermek ve iş mantığı ile HTML kodunu birbirlerinden ayrı yerlerde tutmak amacıyla JSP (Java Server Pages) teknolojisi geliştirilmiştir. JSP üç katmanlı bir mimaride gösterim katmanını geliştirmek için kullanılan teknolojilerden birisidir.



Resim 10.1

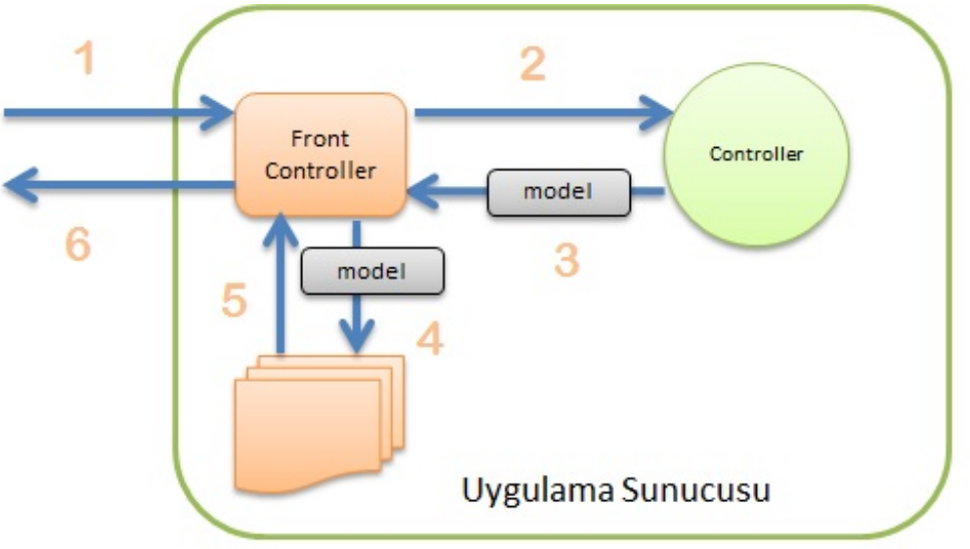
JSP ve Servlet teknolojisi kombine edilebilir. Bu ikilinin beraberliğinden Struts ve Webworks gibi web çatıları doğmuştur. JSTL (Java Standard Tag Library) ve JSP birlikte kullanılarak verilerin sadece gösteriminin yapıldığı arayüzler (view) oluşturulabilir. Servlet teknolojisi kullanılarak arayüzler arası navigasyon ve veri validasyonu (controller) yapılabilir. POJO (Plain Old Java Objects) kullanılarak arayüzlerde gösterilen verilerin yer aldığı model sınıfları oluşturulabilir. Hepsi bir araya geldiğinde MVC (Model View Controller) ismi verilen web çatıları oluşturmak mümkündür.

MVC bir tasarım şablonudur. Bu tasarım şablonuna göre gösterim katmanını oluşturan komponentler belirli bir görevi yerine getirecek şekilde yapılandırılır. MVC kısaltmasındaki her harf bir görev alanını ifade etmektedir. Ait oldukları görev alanında çalışan komponentler sadece ve sadece o görev alanının gerektirdiği görevleri yerine getirirler. Bu tek sorumluluk prensibine (Single Responsibility Principle) işaret etmektedir. Tek bir sorumluluğu olan bir komponent tek bir sebepten dolayı değiştirilebilir. Örneğin bir controller sınıfı arayüz, navigasyon, validasyon ve veri tabanı işlemlerini tek başına yapsa idi, dört değişik sebepten dolayı değişikliğe maruz kalabilirdi ki bu da bu controller sınıfının çok sık bir şekilde ve gereksiz yere değişikliğe uğraması anlamına gelirdi. Tek sorumluluk prensibine uymayan komponentler istikrarsız olduklarından uygulamanın genelde daha kırılgan olmalarını sağlarlar.

Spring herhangi bir web çatısına entegre edilebildiği gibi, Spring bazlı web programcılığını mümkün kılmak için Spring MVC, Spring Webflow ve Spring BlazeDS Integration isminde web çatılarına sahiptir. Bu bölümde Spring MVC web çatısını yakından inceleyeceğiz.

Spring MVC ile Kullanıcı İsteğinin İşlenişi

Spring'in temel web çatısı Spring MVC ismini taşımaktadır. Bu web çatısı MVC tasarım şablonu kullanılarak tasarlanmıştır. Temel işleyiş tarzı resim 10.2 de yer almaktadır.



Resim 10.2

Spring MVC kendi konfigürasyonu için Spring'i kullanmaktadır. Controller sınıfları Spring bean olarak tanımlanır. Spring 2.5 ile birlikte anotasyon bazlı konfigürasyon yapılabilmektedir. Örneğin @Controller anotasyonu herhangi bir bir Java sınıfını bir Spring MVC controller sınıfına dönüştürmektedir. Spring MVC Servlet API'si üzerine inşa edilmiş bir web çatısıdır. Bu Spring MVC'nin çekirdeğinde Servlet ya da HttpServlet sınıflarının kullanıldığı, kullanıcı isteklerinin HttpServletRequest, kullanıcıya gönderilen cevapların HttpServletResponse sınıfları aracılığı ile şekillendirildiği anlamına gelmektedir.

Spring MVC'nin merkezinde DispatcherServlet sınıfı yer almaktadır. Front Controller tasarım şablonu kullanılarak inşa edilen bu yapıda DispatcherServlet kullanıcı isteklerinin uygulama tarafından tatmin edilisini koordine etmektedir. Resim 10.2 de FrontController Spring MVC'nin DispatcherServlet sınıfını temsil etmektedir. DispatcherServlet Struts çatısında ActionServlet, JSF çatısında FacesServlet sınıfları ile aynı görevi yerine getirmektedir.

Kullanıcı web tarayıcısı aracılığı ile herhangi bir linke tıkladığı zaman, bu bir yeni HTTP isteği (request) oluşturur. Bu istek doğrudan DispatcherServlet (resim 10.2:1) tarafından karşılanır. DispatcherServlet'in görevi isteği herhangi bir controller sınıfına (resim 10.2:2) yönlendirmektir. Controller isteği işleme koyan Spring nesnesidir (Spring bean). Bir Spring MVC uygulamasında birden

fazla controller sınıfı olabilir. DispatcherServlet isteği hangi controller sınıfına yönlendireceğine karar verebileceği bir mekanizmaya ihtiyaç duymaktadır. Bu amaçla DispatcherServler handler mapping ismi verilen yapıları kullanır. Handler mapping yapıları kullanıcının isteğini taşıyan web adresi ile bu isteği isleyecek olan controller sınıfları arasındaki bağı oluşturmak için kullanılır. Kullanıcı isteği doğru controller sınıfına eriştikten sonra bu controller sınıfı tarafından işleme konur. Bu çoğu zaman gösterim katmanında yer alan controller sınıfının işletme katmanında yer alan herhangi bir servis sınıfına erişerek, gerekli iş mantığının koşturulmasını sağlaması anlamına gelmektedir. Controller tarafından yapılan işlem web tarayıcısında gösterilmek üzere bir netice doğurabilir. Bu sebeple bu neticenin tekrar kullanıcıya doğru geriye taşınması gerekir. Bu neticenin yer aldığı sınıfa model sınıfı ismi verilir. Controller görevini yerine getirdikten sonra model sınıfını gösterimi yapacak olan view elementinin ismi ile birlikte DispatcherServlet'e yönlendirir (resim 10.2:3). Model sınıfında yer alan bilgilerin kullanıcıya gösterilmeden önce HTML kullanılarak formatlanması gerekir. Bilgileri formatlamak için JSP gibi bir gösterim teknolojisi kullanılır. Formatlamayı yapmak üzere model sınıfı DispatcherServlet tarafından seçilen view elementine yönlendirir (resim 10.2:4). Controller tarafından belirlenen view ismi doğrudan bir JSP sayfasına işaret etmez. Bunu daha çok mantıksal bir isim olarak düşünmek gerekir. DispatcherServlet view resolver (view resolver komponentini daha sonra detaylı inceleyeceğiz) yardımı ile hangi JSP sayfasının gösterimi yapacağını belirler. JSP kullanılabilecek gösterim teknolojilerinden sadece bir tanesidir. View elementi model sınıfını kullanarak gösterim için gerekli HTML kodunu oluşturur (resim 10.2:5). Bu kod HttpServletResponse yardımı ile kullanıcıya taşınır ve web tarayıcısında gösterilir. Bu noktada kullanıcı isteğinin uygulama tarafından cevaplanması işlemi son bulmuştur (resim 10.2:6).

Spring MVC Kurulumu

Spring MVC kurulumu web.xml bünyesinde DispatcherServlet'in tanımlaması ile başlar. Java tabanlı web uygulamaları WEB-INF dizininde yer alan web.xml dosyası ile konfigüre edilir. DispatcherServlet konfigürasyonu kod 10.2 de yer almaktadır.

```
Kod 10.2 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
```

```
    <servlet-class>org.springframework.web.servlet.
                        DispatcherServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>rentacar</servlet-name>
    <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```

servlet-name elementi Spring MVC uygulamasını konfigüre etmek için önemli bir rol oynamaktadır. Kod 10.2 de yer alan DispatcherServlet tanımlaması uygulamayı konfigüre etmek için gerekli olan Spring MVC XML dosyasını WEB-INF dizininde arayacaktır. Konfigürasyon dosyasının bulunabilmesi için isminin [servlet-name]-servlet.xml formatında olması gerekir. Kod 10.2 yer alan örnekte servlet-name elementinde rentacar imini kullandığımız için Spring MVC XML konfigürasyon dosyasının ismi rentacar-servlet.xml olmalıdır.

Uygulama geliştiricisi kullanmak istediği Spring MVC XML konfigürasyon dosyasının ismini kendisi belirleyebilir. Bunu gerçekleştirmek için contextConfigLocation ismindeki bir parametrenin DispatcherServlet'e tanıtılması gerekmektedir. Bu parametrenin değeri kullanılmak istenen konfigürasyon dosyasının lokasyonu ve ismidir. Kod 10.3 de yer alan DispatcherServlet tanımlaması için contextConfigLocation parametresi kullanılmıştır. Bu parametrenin degeri /WEB-INF/mvc-config.xml dir. DispatcherServlet WEB-INF dizininde yer alan mvc-config.xml dosyasını uygulamayı konfigüre etmek için kullanacaktır.

```
Kod 10.3 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
    <servlet-class>org.springframework.web.servlet.
                    DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Tekrar kod 10.2 ye göz attığımızda servlet-mapping isminde bir elementin kullanıldığını görmekteyiz. Bu element kullanıcı isteği ile bu isteği işleyen

servlet arasında bağ kurmak için kullanılır. Kod 10.2 de yer alan örnekte .mvc ekini taşıyan tüm linkler rentacar ismini taşıyan servlet, yani DispatcherServlet'e yönlendirilir. Örneğin http://localhost/welcome.mvc linkine tıklandığında oluşan HTTP isteği (request) DispatcherServlet tarafından işlem görür. url-pattern elementine değişik değerler atayarak DispatcherServlet'e yönlendirilecek kullanıcı istek türlerini yönlendirebiliriz. Kullanılabilecek bazı şablonlar şunlardır:

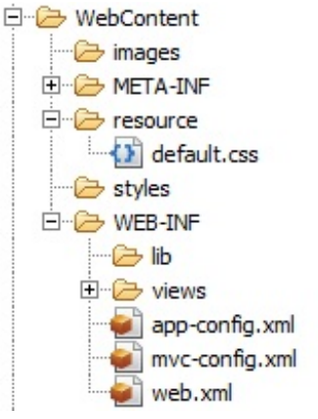
- ***<url-pattern>/</url-pattern>*** - Bir isim alanı (domain name; örneğin http://pratikprogramci.com/) altındaki tüm linkleri DispatcherServlet'e yönlendirir.
- ***<url-pattern>/app/*</url-pattern>*** - Sadece /app/ (örneğin http://pratikprogramci.com/app/welcome) altındaki tüm linkleri DispatcherServlet'e yönlendirir.
- ***<url-pattern>*.abc</url-pattern>*** - .abc ekibini (örneğin http://pratikprogramci.com/welcome.abc) taşıyan tüm linkleri DispatcherServlet'e yönlendirir.

Seçilen url-pattern / olduğu taktirde uygulamaya yöneltilen her istek DispatcherServlet tarafından cevaplanacaktır. Bu Javascript, CSS ve JPEG/GIF gibi kaynaklar için yapılan isteklerin de DispatcherServlet'e yönlendirilmesi anlamına gelmektedir. Statik olan bu kaynakların DispatcherServlet'e yönlendirilmesi gereksizdir. Uygulama sunucusu bu statik kaynakları DispatcherServlet araya girmeden doğrudan kullanıcıya sunabilir. Statik ve dinamik kaynakların Spring tarafından ayrıştırılmalarını sağlamak için mvc:resources elementi kullanılabilir.

```
10.4 - mvc-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/
                   context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/
                   schema/beans
            http://www.springframework.org/schema/beans/
                   spring-beans-3.0.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/
                   spring-mvc-3.0.xsd">
```

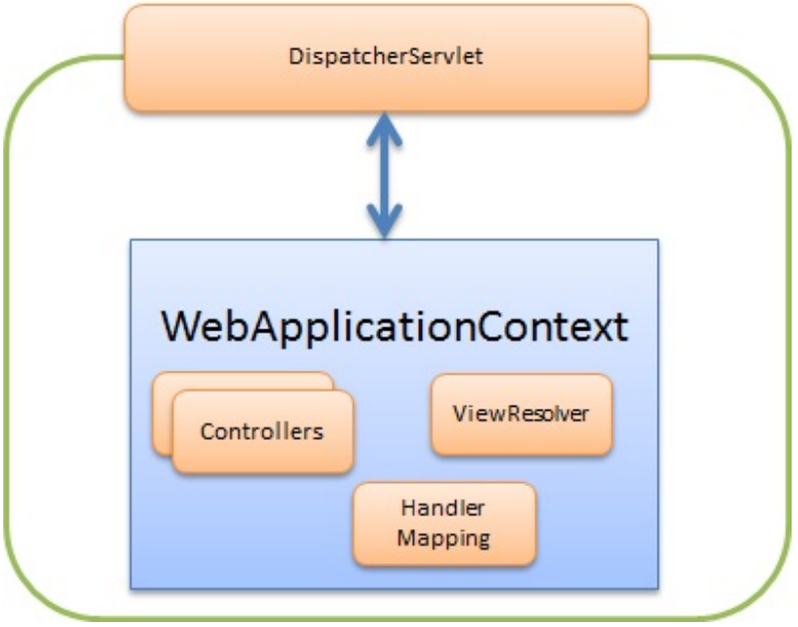
```
    <mvc:resources mapping="/static/**" location="/resource/"/>
</beans>
```

Kod 10.4 de yer alan örnekte mvc:resources elementi yardımı ile statik kaynakların doğrudan sunumunu gerçekleştirmek için bir konfigürasyon oluşturulmaktadır. Mapping element özelliği statik kaynakların web üzerinden hangi adres altında erişilebilir olduklarını, location element özelliği uygulama sunucusunun fiziksel olarak bu kaynaklara nasıl ulaşabileceğini tanımlamaktadır. Bu konfigürasyona göre /static/ konumuna sahip tüm kaynaklar (örneğin http://localhost/static/default.css) /resource/ isimli dizinden alınarak kullanıcıya sunulacaktır. Dizin yapısı resim 10.3 de yer almaktadır.



Resim 10.3

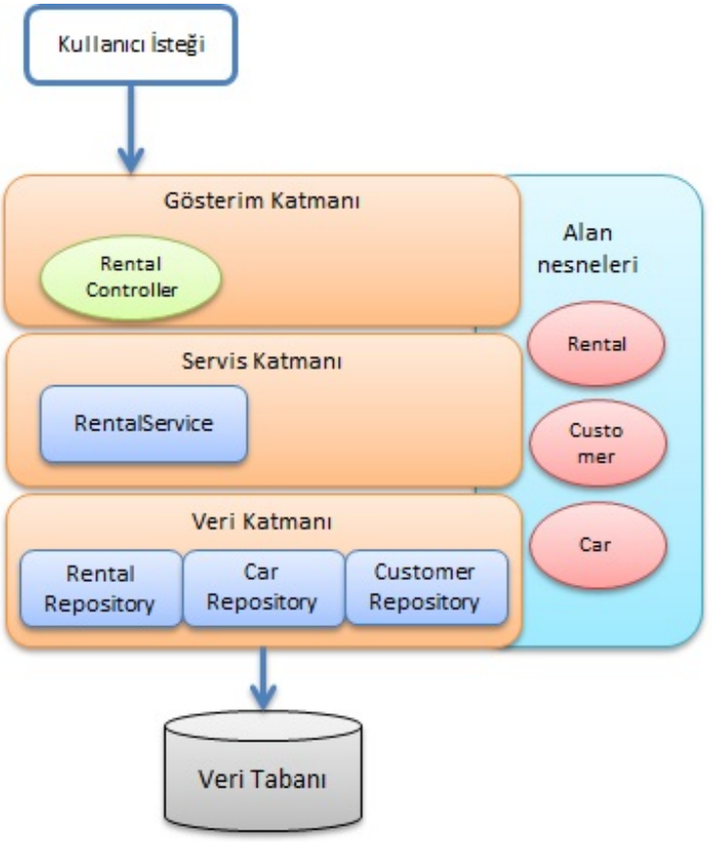
Spring MVC uygulamamızı konfigüre etmek için çıkış noktamız kod 10.4 de yer alan mvc-config.xml ve kod 10.3 de yer alan web.xml dosyalarıdır. Spring MVC çatısında her DispatcherServlet kendi WebApplicationContext nesnesine sahiptir. WebApplicationContext Controller, HandlerMapping, ViewResolver ve diğer Spring nesnelerini ihtiva eder. Spring MVC uygulamasının herhangi bir yerinden WebApplicationContext aracılığı ile bu nesnelere erişmek ve kullanmak mümkündür.



Resim 10.4

Spring MVC ve Uygulama Mimarisi

Diğer bölümlerde Spring'in kullanımını hayali olan bir araç kiralama servisi uygulaması üzerinde örneklemeye çalıştım. Bu bölümde bu uygulamayı web tabanlı bir Spring MVC uygulaması haline getireceğiz. Uygulamayı geliştirmeden önce nasıl bir mimariye sahip olacağını yakından inceleyelim. Resim 10.5 de web tabanlı araç kiralama servisi uygulamasının mimarisi yer almaktadır.

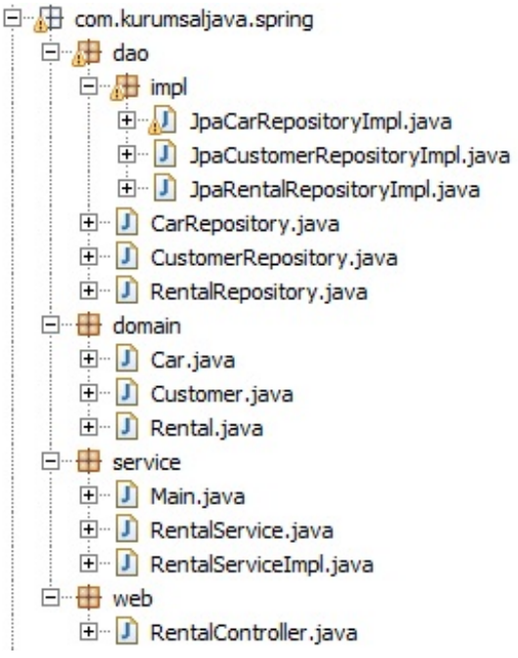


Resim 10.5

Uygulama mimarisi üç katmandan (layer; tier) oluşmaktadır. En üst katman gösterim katmanıdır. Bu katman Spring MVC'yi kullanarak oluşturduğumuz katmandır. Tüm controller, model ve view elementleri bu katmanda yer alır. Araç kiralama sürecini yönetmek üzere oluşturacağımız RentalController sınıfını bu katmanda konuşlandıracağız.

Gelen kullanıcı istekleri gösterim katmanında yer alan controller sınıfları tarafından servis katmanında yer alan servis sınıflarına yönlendirilir. Araç kiralama uygulaması örneğinde RentalController sınıfı araç kiralama işlemini gerçekleştirmek için servis katmanında yer alan RentalService sınıfını kullanmaktadır. RentalService veri tabanı işlemlerini yerine getirmek için veri katmanında yer alan CustomerRepository, RentalRepository ve CarRepository sınıflarını kullanmaktadır. Veri katmanında yer alan repository sınıfları JPA aracılığı ile veri tabanı işlemlerini gerçekleştirmektedir.

Alan (domain) nesneleri olan Car, Customer ve Rental sınıfları tüm katmanlar tarafından ortak kullanılmaktadır. Alan nesneleri servis ve veri katmanında klasik alan nesnesi vazifesini görürken, gösterim katmanında model nesneleri olarak kullanılmaktadırlar.



Resim 10.6

Resim 10.6 da uygulama sınıflarının yer aldığı paket yapısı görülmektedir. Her katman için bir paket oluşturulmuştur. Gösterim katmanını oluşturan sınıflar web, servis katmanını oluşturan sınıflar service, veri katmanını oluşturan sınıflar dao paketinde yer almaktadır. Kullanılan alan nesneleri domain isimli paket içinde yer almaktadır.

Controller Tanımlaması

Her web uygulamasının bir giriş (home; index) sayfası bulunur. Giriş sayfasında yer alan linkler aracılığı ile kullanıcı uygulama ile interaksiyona girer. Bu bölümde araç kiralama servisi uygulaması için giriş ve kiralama işleminin yapıldığı sayfaları oluşturacağız.

Resim 10.7

Resim 10.7 de uygulamanın giriş sayfası yer almaktadır. Böyle bir sayfanın kullanıcıya gösterilmesini sağlamak için HTML kodunun yer aldığı JSP sayfasını ve içeriğin oluşmasını sağlayan IndexController sınıfını oluşturmamız gerekiyor. Kod 10.5 de IndexController sınıfının kodu yer almaktadır.

```
Kod 10.5 - IndexController

package com.kurumsaljava.spring.web;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class IndexController {

    @RequestMapping(value = "/")
    public String index(final ModelMap model) {
        model.addAttribute("titel", "Araç Kiralama Servisi");
        return "index";
    }
}
```

IndexController sınıfı tipik bir Spring MVC controller sınıfında olması gereken özelliklere sahiptir. IndexController sınıfında ilk göze çarpan @Controller anotasyonudur. Bu anotasyon bir Java sınıfını bir Spring MVC controller sınıfı olarak işaretler. Anotasyon bazlı konfigürasyonu aktif hale getirmek için konfigürasyon dosyasına context:component-scan ve mvc:annotation-driven

elementlerini eklememiz gerekmektedir. Kod 10.6 de bu iki elementini kullanılış şekli yer almaktadır. Spring classpath içinde yer alan tüm Java sınıflarını taradıktan sonra @Controller anotasyonunu taşıyan tüm sınıfları Spring MVC controller sınıfı olarak kullanıma hazır tutacaktır.

Kod 10.5 de yer alan IndexController sınıfının herhangi bir üstsınıfının olmadığını görmekteyiz. Bunun yanı sıra bu sınıfın doğrundan servlet çatısına bağımlılığı bulunmamaktadır. Bu controller sınıflarının test edilebilirliğini olumlu etkileyen bir durumdur. Bir Spring MVC uygulamasının nasıl test edildiğini ilerleyen bölümlerde yakından inceleyeceğiz.

```
10.6 - mvc-config.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/
                beans
        http://www.springframework.org/schema/beans/
                spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
                spring-context-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/
                spring-mvc-3.0.xsd">

    <context:component-scan
                base-package="com.kurumsaljava"/>

    <mvc:annotation-driven/>

    <mvc:resources mapping="/static/**" location="/static/"/>
</beans>
```

Kullanıcı ve uygulama geliştirici olarak beklentimiz http://localhost:8080/ adresine istek yapıldığında resim 10.7 de yer alan sayfanın görünmesidir. Burada uygulamaya yapılan istek / adresidir. DispatcherServlet'in bu isteği IndexController sınıfına yönlendirebilmesi için bu adres ile IndexController sınıfı arasında bir bağ oluşturmamız gerekmektedir. Bu bağı oluşturmak için @RequestMapping anotasyonu kullanılır. Adres ile controller arasında oluşturulan bu bağa handler mapping ismi verilmektedir. Kod 10.5 de yer alan

örnekte @RequestMapping anotasyonu / adresine gelen tüm kullanıcı isteklerinin IndexController sınıfında yer alan index() metoduna yönlendirilmesini sağlar. @RequestMapping anotasyonun yer aldığı metot herhangi bir ismi taşıyabilir.

Spring MVC bünyesinde değişik türde handler mapping implementasyonları mevcuttur. Bunlardan bazıları:

- ***BeanNameUrlHandlerMapping*** - Talep edilen web adresi ile (örneğin /rental) aynı Spring bean ismini taşıyan controller sınıfları arasında bağ (mapping) kurmak için kullanılır.
- ***DefaultAnnotationHandlerMapping*** - @RequestMapping anotasyonunu taşıyan controller sınıfları ile kullanıcı isteğini arasında bağ kurmak için kullanılan handler mapping türüdür.
- ***ControllerClassNameHandlerMapping*** - Talep edilen web adresi ile (örneğin /rental) aynı ismi taşıyan controller sınıfları arasında bağ (mapping) kurmak için kullanılır.
- ***SimpleUrlHandlerMapping*** - Talep edilen web adresinin istek anında bir controller sınıfı ile dinamik olarak eşlenme işlemini yapmak için kullanılan handler mapping türüdür.

Kod 10.6.1 de yer alan örnekte /index=indexController şeklinde bir eşleme yapılmaktadır.

```
Kod 10.6.1 - applicationContext.xml
<bean
    class="org.springframework.web.servlet.handler.
                            SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/index">indexController</prop>
        </props>
    </property>
</bean>

<bean id="indexController"
    class="com.kurumsaljava.spring.web.IndexController"/>
```

Arkasında bir controller sınıfı olmayan view elementlerini göstermek için kod 10.6.2 de yer alan UrlFilenameViewController sınıfı kullanılabilir. Bu controller implementasyonu istek yapılan sayfa ismini view ismine dönüştürerek, bu view elementinin kullanıcıya gösterilmesini sağlamaktadır. Örneğin kullanıcı

/secure/index şeklinde bir sayfa talebinde bulunduysa, UrlFilenameViewController /WEB-INF/views/index.jsp (bknz. Kod 10.7) JSP sayfasının gösterilmesini sağlar.

```
Kod 10.6.2 - applicationContext.xml

<bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.
                SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/**">urlFilenameViewController</prop>
        </props>
    </property>
</bean>

<bean id="urlFilenameViewController"
    class="org.springframework.web.servlet.mvc.
        UrlFilenameViewController" />
```

Handler mapping sınıfları HandlerMapping ismini taşıyan interface sınıfı implemente ederler. Yazılımcının bu interface sınıfını implemente ederek yeni handler mapping sınıfları oluşturması mümkündür. Uygulama bünyesinde bir handler mapping tanımlanmadığı taktirde Spring otomatik olarak BeanNameUrlHandlerMapping ya da DefaultAnnotationHandler handler mapping implementasyonunu kullanır. Anotasyon bazlı controller sınıfları kullandığımız için bu bizim örneğimizde DefaultAnnotationHandler sınıfıdır.

Model Taşıyıcı ModelMap

Kod 10.5 de yer alan index() metodunun ModelMap tipinde bir parametresi mevcuttur. Bu nesne conroller ile view arasında veri taşımak için kullanılan bir yapıdır. Verileri taşıyan model nesneleri ModelMap aracılığı ile view elementlerinin kullanımına sunulur. Bu yapıya eklenen her model nesnesinin bir anahtarı mevcuttur. Bu anahtar kullanılarak view içinde iken verileri tekrar elde etmek mümkündür. Kod 10.5 de yer alan örnekte index ismini taşıyan view elementi için titel anahtarına sahip String veri tipinde bir model nesnesi ModelMap'e eklenmektedir.

View Resolver Tanımlaması

Index() metodu index değerini taşıyan bir String nesnesini geri vermektedir. Bu gösterimi yapacak olan JSP sayfasının ismidir. DispatcherServlet IndexController sınıfında yer alan index() metodundan bu değeri edindikten sonra, gerekli JSP sayfasına yönlendirmeyi gerçekleştirir. Bu ismi taşıyan bir JSP sayfasını bulabilmesi için mvc-config.xml dosyasında bir view resolver'in tanımlanması gerekmektedir. Bunun bir örneği kod 10.7 da yer almaktadır.

```
Kod 10.7 - mvc-config.xml

<bean class="org.springframework.web.servlet.view.
                InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

DispatcherServlet kod 10.7 de kullandığımız InternalResourceViewResolver yardımı ile index.jsp ismini taşıyan JSP dosyasını /WEB-INF/views/index.jsp lokasyonunda bulmaya çalışacaktır. DispacherServlet gösterimden sorumlu JSP sayfasını tespit ettikten sonra kontrolü bu JSP sayfasına bırakarak, JSP sayfasının model sınıfları yardımı ile HTML içeriği oluşturmasını sağlayacak, bu işlem tamamlandıktan sonra kontrolü tekrar üstlenerek bu içeriğin HttpServletResponse nesnesi olarak web tarayıcısına gönderilmesini sağlayacaktır. Bu işlemin ardından kullanıcı ile uygulama arasındaki istek döngüsü son bulmaktadır.

WEB-INF dizininde yer alan JSP sayfalarına web tarayıcısı üzerinden doğrudan erişmek mümkün değildir. WEB-INF dizinde yer alan tüm kaynaklar uygulama sunucusu tarafından korunur. Bu açıdan bakıldığında InternalResourceViewResolver JSP sayfalarının kaynak kodunun korunması için iyi bir seçenektir.

```
Kod 10.8 - index.jsp

<html>
<body>
    <h1>${titel}</h1>

    <a href="/rental">Araç kiralama formu </a>
</body>
</html>
```

Kod 10.8 de index.jsp sayfasının kaynağı yer almaktadır. Dolar işareti ile

başlayan tanımlama (${titel}) bir yer tutucudur (placeholder). Titel kelimesi IndexController.index() metodunda oluşturarak ModelMap nesnesine eklediğimiz String tipindeki modelin anahtarıdır. JSP sayfası içinde bu modelin içeriğine erişebilmek için bu anahtarı kullanmamız gerekmektedir. Spring MVC çatısı JSP sayfalarında dolar işareti ile karşılaştığı zaman bunun bir model element anahtarı olabileceğini düşünerek, bu yer tutucunun modelin değeri ile yer değiştirmesini sağlar. Bu şekilde JSP sayfalarında dinamik içerik oluşmuş olur.

View Resolver Türleri

Gösterimi yapan view elementlerinin MVC çatısı tarafından lokalize edilerek model sınıflarında yer alan bilgilerin kullanıcıya gösterilmesi (model rendering) gerekmektedir. Bu görevi Spring MVC bünyesinde view resolver sınıfları üstlenmektedir. Kod 10.7 de InternalResourceViewResolver ismini taşıyan ilk view resolver sınıfı ile tanıştık.

Spring MVC view elementlerinin bulunma işleminde kullanılmak üzere ViewResolver ve View isminde iki interface sınıf ihtiva etmektedir. ViewResolver view element isimleri ile view elementleri arasında eşleme (mapping) yapmak için kullanılır. Örneğin kod 10.7 de kullandığımız InternalResourceViewResolver sınıfı kod 10.5 deki index() metodunun son satırında yer alan index ismini /WEB-INF/views/index.jsp ile eşlemektedir. Eğer kod 10.5 de yer alan index() metodu abc değerini geri vermiş olsaydı, InternalResourceViewResolver /WEB-INF/views/abc.jsp şeklinde bir eşleme yaparak abc.jsp sayfasının gösterimi yapmasını sağlardı.

ViewResolver interface sınıfı yanı sıra bahsettiğim View interface sınıfı kullanıcı isteğinin (HttpServletRequest) seçilen view elementine aktarılmasını sağlamak için kullanılmaktadır.

ViewResolver interface sınıfını implemente eden bazı Spring MVC view resolver sınıfları şunlardır:

- ***AbstractCachingViewResolver*** - Bazı view elementleri için gösterim öncesi hazırlık yapmak gerekli olabilir. Her gösterim öncesinde bu hazırlık işlemlerinin tekrarını önlemek için view elementleri önbelleğe (cache) alınabilir. AbstractCachingViewResolver sınıfı view elementlerinin önbelleğe alınmalarını sağlamak için genişletilebilecek view resolver sınıfıdır.

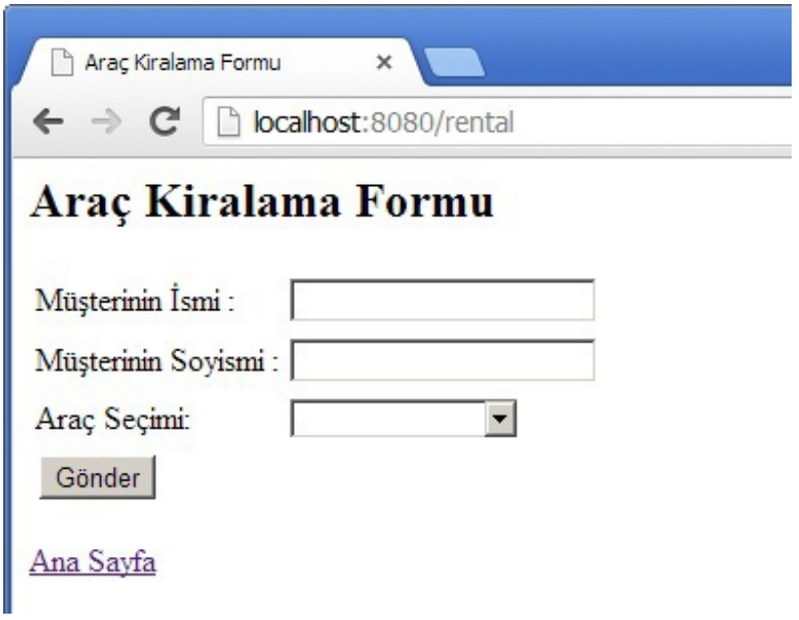
- ***XmlViewResolver*** - Bir XML dosyasında yer alan ve bir Spring bean gibi tanımlı olan view elementlerini bulmak için kullanılan view resolver türüdür. Varsayılan XML /WEB-INF/views.xml isimli dosyasıdır.
- ***ResourceBundleViewResolver*** - Property dosyalarında yer alan view tanımlamalarını kullanarak istenilen view elementini bulmak için kullanılan view resolver türüdür. Varsayılan property dosyası ismi views.properties'dir. Bu dosyasının classpath içinde olması gerekir.
- ***UrlBasedViewResolver*** - InternalResourceViewResolver gibi sınıfların üstsınıfı olan view resolver sınıfıdır. Kullanılan web adresi ile view elementi arasında ilişki kurmak için kullanılır. Örneğin talep edilen adres /index ise bu view resolver uygulama sunucusunun ana dizininde (doc root) yer alan index.jsp sayfasını seçer.
- ***VelocityViewResolver / FreeMarkerViewResolver*** - UrlBasedViewResolver sınıfının bir altsınıfı olan bu view resolver JSP gibi bir gösterim teknolojisi olan Velocity ve Freemarker ile yapılmış view elementlerinin seçimi için kullanılır.
- ***ContentNegotiatingViewResolver*** - Kullanıcı isteği (HttpServletRequest) bünyesinde yer alan "request file name" ya da "Accept header" elementlerinin değerine göre view element seçimi yapan view resolver türüdür. "request file name" ya da "Accept header" kullanıcının talep ettiği içerik tipini belirler.
- ***JasperReportsViewResolver*** - View elementinin Jasper Reports olarak hazırlanmış bir dosyalar arasında seçimi için kullanılan view resolver sınıfıdır.
- ***TilesViewResolver*** - Tiles template olarak hazırlanmış bir view elementinin seçimi için kullanılır.
- ***XsltViewResolver*** - XSLT tabanlı bir view elementinin seçimi için kullanılan view resolver türüdür.

Görüldüğü gibi Spring MVC uygulamalarında gösterim için kullanılan teknoloji JSP ile sınırlı degildir. Freemarker ya da Velocity gibi şablon (template) mekanizmaları sunan gösterim teknolojilerini de kullanmak mümkündür. Gerekli view resolver sınıfının konfigürasyonu kullanılan gösterim teknolojisinin Spring MVC ile entegrasyonunu sağlamaktadır.

Araç Kiralama Formu

Araç kiralama işlemini web tabanlı hale getirebilmek için bir HTML form

oluşturmamız gerekmektedir. Böyle bir form prototipi resim 10.8 de yer almaktadır.



Resim 10.8

Böyle bir formu oluşturabilmek için form elementlerini ihtiva eden bir JSP sayfasına, forma kaydedilen bilgileri işleyen bir servis sınıfına, navigasyon ve veri validasyonundan sorumlu bir controller sınıfına, hem kullanıcı verilerini hem de işlem sonuçlarını bünyesinde taşıyan bir model sınıfına ve veri tabanı işlemlerini gerçekleştiren DAO sınıflarına ihtiyacımız bulunmaktadır.

Controller sınıfının içeriğini oluşturarak başlamamız doğru olacaktır, çünkü hangi sayfanın gösterileceğini tayin eden controller sınıfıdır. RentalController ismini taşıyan controller sınıfı kod 10.9 da yer almaktadır.

```
Kod 10.9 - RentalController

package com.kurumsaljava.spring.web;

import java.util.LinkedHashMap;
import java.util.Map;
import javax.inject.Inject;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import com.kurumsaljava.spring.domain.Rental;
import com.kurumsaljava.spring.service.RentalService;

@Controller
@RequestMapping("/rental")
public class RentalController {

    private static final String RENTAL_FORM = "rentalForm";

     @Inject
     private RentalService service;


    private static final String DONE_VIEW = "done";
    private static final String RENTAL_VIEW = "rental";

    @RequestMapping(method = RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }

    private void buildCarList(final ModelMap model) {
        final Map<String,String> cars =
            new LinkedHashMap<String,String>();
         cars.put("", "");
         cars.put("1", "Ford Fiesta");
         cars.put("2", "Renault Twingo");
         cars.put("3","Audi A4");
         model.put("cars", cars);
    }

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit( final ModelMap model,
            @ModelAttribute(RENTAL_FORM) @Valid
                final Rental rental,
            final BindingResult result) {
        if (result.hasErrors()) {
            buildCarList(model);
            return RENTAL_VIEW;
        } else {
            processRental(rental);
            return DONE_VIEW;
        }
    }

    private void processRental(final Rental rental) {
        service.rentACar(rental);
```

```
    }
}
```

@Controller anotasyonu ile daha önce tanışmıştık. @Controller anotasyonu sınıfın rolünü tanımlamaktadır. Bu anotasyon yardımı ile RentalController sınıfı bir Spring MVC controller sınıfı haline gelmektedir.

@RequestMapping anotasyonuyla da daha önce tanışmıştık. Kod 10.5 de yer alan örnekte @RequestMapping anotasyonunu metot bazında kullanmıştık. Bu anotasyonu hem metot hem de sınıf bazında kullanmak mümkündür. Sınıf bazında ve metot bazında aynı anda kullanıldığı taktirde, sınıf bazında kullanım şekli controller sınıfının sorumlu olduğu ana uygulama adresini tayin eder. Metot bazında kullanılan anotasyonlar bu adresi tamamlayıcı nitelikte olur. Örneğin araç kiralama formunu kullanıcıya göstermek için kullanılan adres kod 10.9 da http://localhost/rental iken, kod 10.10 da http://localhost/rental/new şeklindedir.

```
Kod 10.10 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/new", method =
                    RequestMethod.GET)
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Kullanıcı isteği ile gönderilen parametreleri (request parameter) @RequestMapping anotasyonu yardımı ile değerlendirmek mümkündür. Kod 10.11 de yer alan örnekte kullanıcı isteği /rental/form?new olduğu taktirde getForm() metodu devreye girecektir.

```
Kod 10.11 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/form", method = RequestMethod.GET,
```

```
                params="new")
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Kod 10.11 de yer alan örnekte new parametresi bir değer taşımamaktadır. Bu parametreye bir değer atayarak kullanıcı isteğini uygulamaya göndermek mümkündür. Örneğin /rental/form?new=1 şeklinde bir isteği karşılamak için @RequestMapping anotasyonunun params="new=1" olarak şekillendirilmesi yeterli olacaktır. Eğer bir parametrenin kullanıcı isteği bünyesinde yer almasını istemiyorsak, bu isteğimizi ünlem işareti kullanarak ifade edebiliriz. params=!new seklindeki bir tanımlama istek bünyesinde new parametresinin olmaması durumunda işlem görecektir.

@RequestMapping anotasyonunda yer alan headers elementi ile kullanıcı isteği ile gönderilen HTTP başlık (header) parametrelerini değerlendirmek mümkündür. Bunun bir örneği kod 10.12 de yer almaktadır.

```
Kod 10.12 - RentalController

@Controller
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(value="/form", method = RequestMethod.GET,
            headers="key=value")
    public String getForm(final ModelMap model) {
        model.addAttribute(RENTAL_FORM, new Rental());
        buildCarList(model);
        return RENTAL_VIEW;
    }
}
```

Tekrar kod 10.9 da yer alan RentalController sınıfına göz atalım. getForm() metodu üzerinde yer alan RequestMapping anotasyonunda method = RequestMethod.GET ibaresi yer almaktadır. Eğer gönderilen kullanıcı isteği HTTP GET özelliğine sahip ise, bu durumda RentalController bünyesinde yer alan getForm() metodu devreye girecektir. Bu metodun görevi resim 10.8 de yer alan araç kiralama formunun kullanıcıya gösterilmesini sağlamaktır. RequestMapping anotasyonunu kullanarak ifade etmek istediğimiz şey şudur:

Eğer kullanıcı http://localhost/rental şeklinde bir istekte bulunuyorsa ve gönderilen bu isteğin türü HTTP GET ise, bu durumda kullanıcıya araç kiralama formunu göster.

Resim 10.8 der yer alan formun kullanıcıya gösterildiğini farz edelim. Kullanıcı gerekli form alanlarını doldurmuş ve Gönder butonuna tıklamış olsun. Kullanıcının girdiği bu bilgiler uygulamaya bir HttpServletRequest nesnesi olarak erişir. HttpServletRequest.getParameter() metodunu kullanarak kullanıcının girmiş olduğu değerlere erişebiliriz. Bu değerler Rental sınıfından bir nesne olarak elimize ulaşsaydı ve HttpServletRequest ile uğraşmak zorunda kalmasaydık daha iyi olmaz mıydı? Spring MVC'yi kullanmamızın ana nedenlerinden bir tanesi de, servlet çatısına derin dalış yapıp, HttpServletRequest gibi sınıflarla uğraşmak zorunda kalmak istemeyişimizdir. Kullanıcıya gösterilen formu ve ihtiva ettiği bilgileri bir POJO sınıf olarak tasarlayıp, Spring MVC tarafından kullanıcı bilgileri ile otomatik olarak donatılmasını sağlayabiliriz. Bu amaçla kod 19.9 da yer alan getForm() metodunda model nesnesine yeni bir Rental nesnesi eklemektedir. Bu Rental nesnesi Gönder tuşuna tıklandıktan sonra kullanıcının girmiş olduğu tüm bilgileri ihtiva edecektir. Peki Spring MVC formda yer alan verileri Rental sınıfından olan nesneye nasıl eklemektedir? Bunu anlayabilmek için formu oluştururken kullandığımız JSP sayfasına bir göz atmamız gerekmektedir. Formu kullanıcıya göstermek için oluşturduğumuz rental.jsp kod 10.13 de yer almaktadır.

```
Kod 10.13 - rental.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>
<%@taglib uri="http://www.springframework.org/tags/form"
            prefix="form"%>
<html>
<head>
<title>Ara&ccedil; Kiralama Formu</title>
</head>
<body>
    <h2>Ara&ccedil; Kiralama Formu</h2>

    <form:form method="post" action="/rental"
            commandName="rentalForm">
        <table>
            <tr>
                <td>Müşterinin İsmi :</td>
                <td><form:input path="customer.firstname" />
```

```
                    <font color="Red">
                      <form:errors path="customer.firstname"
                             delimiter=", "/>
                    </font>
                </td>
            </tr>
            <tr>
                <td>Müşterinin Soyismi :</td>
                <td><form:input path="customer.name" />
                  <font color="Red">
                    <form:errors path="customer.name"
                             delimiter=", "/>
                  </font>
                </td>
            </tr>
            <tr>
                <td>Araç Seçimi:</td>
                <td>
                    <form:select path="car.userSelection">
                        <form:options items="${cars}" />
                    </form:select>
                   <font color="Red">
                     <form:errors path="car.userSelection"
                             delimiter=", "/>
                   </font>
                </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit"
                             value="G&ouml;nder" /></td>
            </tr>
        </table>

    </form:form>

<p><a href="/rental">Ana Sayfa</a>
</body>
</html>
```

Kod 10.13 der yer alan JSP sayfasında HTML elementleri ile Spring MVC uygulaması arasında bağ oluşturmak için Spring MVC'nin ihtiva ettiği form ismini taşıyan JSP Custom Tag Library (taglib) kullanılmaktadır. Taglib elementlerini HTML elementleri olarak düşünebiliriz. Taglib'ler JSP sayfalarında Java kodu ile program yazmak yerine, HTML benzeri elementler kullanarak dinamik içerik oluşturmak için kullanılmaktadır. Örneğin form:form taglib elementi ile bir HTML formu oluşturulmaktadır. Bu form ile kullanıcının

form aracılığı ile girdiği verileri tutmak için kullanılacak olan Rental nesnesi arasında bağ oluşturmak için commandName elementi kullanılmaktadır. Bu elementin değeri olan rentalForm RentalController.getForm() metodunda model nesnesine eklenen Rental nesnesinin anahtarıdır.

Kullanıcının girmiş olduğu bilgileri Rental nesnesinde yer alan değişkenlere atamak için form:input elementi kullanılmaktadır. Form:input path="customer.firstname" direktifi form üzerinden girilen müşteri ismini doğrudan Rental nesnesinde yer alan Customer nesnesinin name ismini taşıyan değişkenine atamaktadır. Rental sınıfının yapısı kod 10.14 de yer almaktadır. Görüldüğü gibi bir JPA entity sınıfını form nesnesi olarak kullanmak mümkündür. Bu nesneye gösterim katmanında gerekli veriler atandıktan sonra veri katmanında JPA kullanarak bu nesne ve ihtiva ettiği veriler veri tabanına eklenebilir.

```
Kod 10.14 - Rental

package com.kurumsaljava.spring.domain;

@Entity
@Table(name = "rental")
public class Rental {

    @Id
    @GeneratedValue
    @Column(name = "id")
    private long id;

    @Valid
    @OneToOne
    private Car car;

    @Valid
    @OneToOne
    private Customer customer;

    @Column(name="rented")
    private boolean rented;
}
```

Resim 10.8 de yer alan formda müşteri isim ve soyismi için iki alan, kiralanacak aracın türünü belirlemek için bir araç listesi bulunmaktadır. Bu listenin içeriği dinamik olarak kod 10.9 da yer alan getForm() metodunda oluşturulmaktadır. buildCarList() metodu araç türlerini ihtiva eden bir map oluşturarak, bu map

nesnesini cars ismi altında model nesnesine eklemektedir. Bu listenin resim 10.8 de yer alan forma gösterilmesi için kod 10.13 de yer alan JSP sayfasında form:select kullanılmaktadır. Bu element bünyesinde form:options items="${cars} yardımı ile araç türlerinin yer aldığı map nesnesi edinilerek, HTML Select bileşeni oluşturulmaktadır.

Kullanıcı forma gerekli bilgileri girip, Gönder butonuna tıkladığında bu istek uygulama sunucusuna bir HTTP POST isteği olarak gönderilir. Bunun sebebi kod 10.13 de yer alan form:form method="post" action="/rental" tanımlamasıdır. Bu istek /rental adresini taşıdığı için DispatcherServlet tarafından tekrar RentalController sınıfına yönlendirilir. İsteğin controller bünyesindeki hangi metot tarafından cevaplanacağını @RequestMapping anotasyonu belirler. Bu metot processSubmit() ismini taşımaktadır, çünkü bu metodun sahip olduğu @RequestMapping anotasyon method = RequestMethod.POST şeklinde konfigüre edilmiştir.

Formu işleyen processSubmit() metodunun ModelMap, Rental ve BindingResult tipinde üç metot parametresi bulunmaktadır. Bu parametreler haricinde @ModelAttribute ve @Valid anotasyonları kullanılmıştır.

@ModelAttribute anotasyonu ile model içinde yer alan bir nesneye erişmek mümkündür. Kod 10.9 da yer alan processSubmit() metodunda @ModelAttribute anotasyonu model içinde yer alan Rental nesnesini edinmek için kullanılmaktadır. Bu nesne kullanıcının form üzerinde girdiği verileri ihtiva etmektedir. Spring MVC otomatik olarak form alanlarına girilen verileri Rental nesnesine yerleştirir. Bu işleme data binding ismi verilmektedir.

Spring MVC data binding işlemi için WebDataBinder sınıfını kullanmaktadır. WebDataBinder kullanıcı isteği içinde yer alan HTTP parametreleri ile form verilerini tutan nesne arasında değişken ismi bazında eşitleme yapar. Örneğin kod 10.13 de yer alan JSP sayfasında müşteri ismini tutmak için form:input path="customer.firstname şeklinde bir tanımlama yapılmıştır. Kullanıcı Gönder tuşuna tıkladığında WebDataBinder rental.getCustomer().setFirstname() çağrısı ile kullanıcının girmiş olduğu müşteri ismini rental nesnesinde yer alan customer nesnesinin firstname isimli değişkenine yerleştirir. Bu işlem mevcut tüm form elementleri için gerçekleştirilir.

@Valid anotasyonu otomatik veri kontrolü (validation) yapmak için kullanılmaktadır. Bu anotasyon ile Spring MVC JSR (Java Specification Requests) 303 ile Java EE 6'nın bir parçası haline gelen Bean Validation

çatısının kullanımını mümkün kılmaktadır. Kullanıcı formu doldurup, Gönder tuşuna tıkladığında Spring MVC önce WebDataBinder yardımı ile rental nesnesini yapılandıracak ve akabinde @Valid anotasyonuna denk geldiği taktirde otomatik veri kontrol işlemini gerçekleştirecektir. Spring MVC veri kontrolünü Bean Validation çatısında yer alan anotasyonlar yardımı ile yapmaktadır. Kod 10.15 de yer alan Customer sınıfında müşteri ismi (firstname) ve soyismi için veri kontrolü @NotBlank ve @Length anotasyonları kullanılarak yapılmaktadır. @NotBlank form elementine mutlaka bir değerin girilmesi gerektiğini ifade ederken, @Length ile girilen verinin kaç harften oluşabileceği tayin edilmektedir. Kod 10.15 de yer alan örnekte müşteri soyisminin en az bir, en fazla on harften oluşması gerekmektedir. Aksi taktirde veri kontrolü neticesi olarak bir hata oluşacaktır. Data binding ve veri kontrolü sırasında oluşan tüm hatalar BindingResult içinde yer almaktadır. Result.hasErrors() ile hata mevcudiyeti kontrol edilebilir. Hata durumunda formun tekrar kullanıcıya oluşan hatalar ile birlikte gösterilmesi gerekmektedir. return RENTAL_VIEW; tekrar form sayfasına dönüş yapılmasını sağlar.

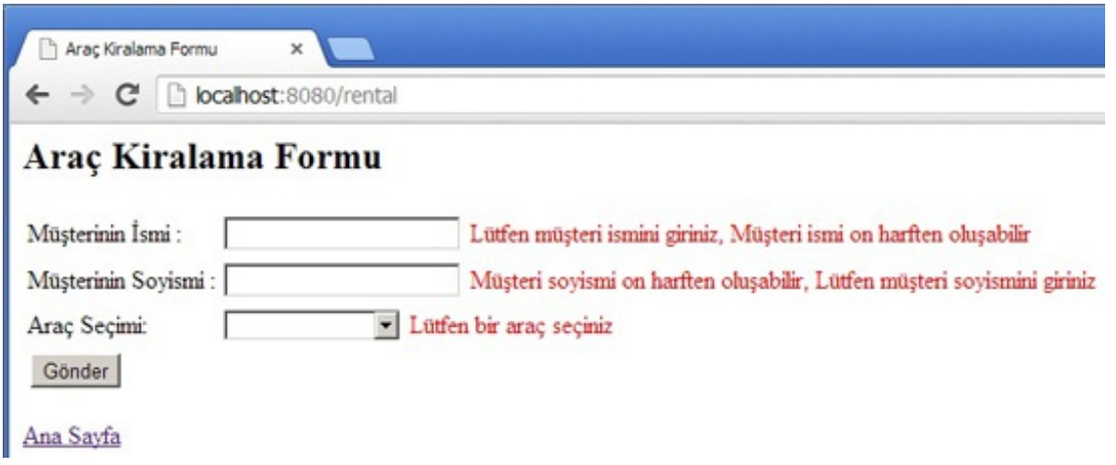
```
Kod 10.15 - Customer

public class Customer {

    private long id;

    @NotBlank
    @Length(min=1, max=10)
    private String name;

    @NotBlank
    private String firstname;

    private int age;
}
```

Bean Validation çatısında veri kontrolü yapmak için kullanılabilecek anotasyonlar şunlardır:

- ***@AssertFalse*** - Değişkenin false değerinde olmasını bekler.
- ***@AssertTrue*** - Değişkenin true değerinde olmasını bekler.
- ***@DecimalMin*** - Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden yüksek olmasını bekler.
- ***@DecimalMax*** - Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden düşük olmasını bekler.

- ***@Digits*** - Değişkenin bu anotasyon ile belirlenen virgül öncesi ve sonrası basamaklardan oluşmasını bekler.
- ***@Past*** - java.util.Date ve java.util.Calendar için kullanılabilir. Değişken değerinin geçmiş bir zaman diliminde olmasını bekler.
- ***@Future*** - java.util.Date ve java.util.Calendar için kullanılabilir. Değişken değerinin gelecek bir zaman diliminde olmasını bekler.
- ***@Min*** - Sadece integer değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden yüksek olmasını bekler.
- ***@Max*** - Sadece integer değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen değer ile eşit ya da bu değerden düşük olması bekler.
- ***@NotNull*** - Değişkenin null değerine sahip olmamasını bekler.
- ***@Null*** - Değişkenin null değerine sahip olmasını bekler.
- ***@Pattern*** - Sadece String veri tipine sahip değişkenler için kullanılabilir. Değişkenin bu anotasyon ile belirlenen regex ifadesi ile uyuşmasını bekler.
- ***@Size*** - String, Collection, Map ve Array veri tiplerine sahip değişkenler için kullanılabilir. Değişkenin sahip olduğu uzunluğun bu anotasyon ile belirlenen min ve max değerleri içinde olmasını bekler.
- ***@Valid*** - Veri kontrolünün rekursif olarak yapılmasını sağlar. Bunun bir örneği kod 10.14 de yer almaktadır. Eğer Rental sınıfı bünyesinde car ve customer nesneleri için @Valid anotasyonu kullanılmamış olsaydı, Customer sınıfında yer alan (kod 10.15) firstname ve name değişkenlerinin veri kontrolü yapılması mümkün olmazdı, çünkü veri kontrolü giriş noktası processSubmit() (kod 10.9) metot parametrelerinden birisi olan ve @Valid ile işaretlenmiş rental nesnesidir.

Veri kontrolü ve data binding işlemlerinde oluşan hataların ekranda kullanıcıya gösterilebilmesi için JSP sayfasında form:errors (kod 10.13) elementi kullanılmaktadır. Resim 10.9 da form bünyesinde oluşabilecek hatalar yer almaktadır. Bu hataların kullanıcıya gösterilebilmesi için formun tekrar yüklenmesi gerekmektedir. Bu işlem processSubmit() metodunda BindingResult nesnesinde yer alan hataların (result.hasErrors()) kontrolü ve return RENTAL_VIEW; komutuyla gerçekleşmektedir.



Resim 10.9

Resim 10.9 da yer alan hata mesajları nerede tutulmaktadır? Bu mesajları bir değer (property) dosyasından edinmek mümkündür. Data binding ve veri kontrolü sırasında oluşan her hatanın bir anahtarı bulunmaktadır. Eğer bir değer dosyasında bu anahtarlara kullanıcıya gösterilmek istenen hata mesajı atanırsa, Spring MVC oluşan hatalar için bu değer dosyasını kullanır. Araç kiralama servisi için kullanılan hata mesajları dosyası kod 10.16 da yer almaktadır.

```
Kod 10.16 - messages.properties

Length.rentalForm.customer.firstname =
        M\u00fc\u015fteri ismi on harften olu\u015fabilir
NotBlank.rentalForm.customer.firstname =
        L\u00fctfen m\u00fc\u015fteri ismini giriniz
Length.rentalForm.customer.name =
        M\u00fc\u015fteri soyismi on harften olu\u015fabilir
NotBlank.rentalForm.customer.name =
        L\u00fctfen m\u00fc\u015fteri soyismini giriniz
NotBlank.rentalForm.car.userSelection =
        L\u00fctfen bir ara\u00e7 se\u00e7iniz
```

Örneğin müşteri ismi girilmeden Gönder butonuna tıklandığında, veri kontrolü esnasında oluşan hatanın anahtarı NotBlank.rentalForm.customer.firstname olacaktır. Görüldüğü gibi bu anahtar veri kontrolü için kullanılan anotasyon (NotBlank), form (rentalForm), değişkenin yer aldığı nesne (customer) ve değişken isminden (firstname) oluşmaktadır. Bu şekilde tüm oluşabilecek hatalar için hata anahtarlarını kestirmek ve kod 10.16 da yer alan messages.properties ismini taşıyan dil dosyasını oluşturmak mümkündür. Oluşturulan bu dosyaya resource bundle ismi verilmektedir. Resource bundle dosyaları çok dilli uygulamalarda kullanılan yapılardır.

Örneğin hata mesajlarının ingilizce dilinde görünmeleri isteniyorsa messagesenEN.properties, almanca dilinde görünmeleri isteniyorsa messagesdeDE.properties ismini taşıyan dil dosyaları oluşturulması ve gerekli tercümenin yapılması yeterli olacaktır. Spring MVC kullanılan dil türüne göre (Locale) gerekli dil dosyasını seçerek, hata mesajlarının bu dilde gösterilmesini sağlayacaktır. Spring MVC'nin dil dosyasını bulabilmesi için konfigürasyon dosyasında dil dosyasının ismi ve lokasyonunun ResourceBundleMessageSource yardımı ile tanımlanması gerekmektedir. Bunun bir örneği kod 10.17 de yer almaktadır.

```
Kod 10.17 - mvc-config.xml

<bean
    class="org.springframework.context.support.
        ResourceBundleMessageSource"
    id="messageSource">
    <property name="basename" value="messages" />
</bean>
```

Kod 10.17 de yer alan örnekte Spring MVC dil dosyalarını classpath içinde arayacaktır. Spring MVC'nin beklediği dil dosya ismi messages'dir. basename parametresi dil dosyasının kök ismini tayin eder. Çok dilli uygulamalarda dil dosya ismi kök dosya ismi artı kullanılan dil türevidir (örneğin ingilizce için en_EN).

Controller Sınıfları ve Bağımlılıkların Enjekte Edilmesi

Resim 10.9 da yer alan araç kiralama formunun kullanıcı tarafından eksiksiz olarak doldurulup, işlenmek üzere uygulamaya gönderildiğini farz edelim. Veri kontrolü başarıyla yapıldıktan sonra kod 10.9 da yer alan processSubmit() metodundaki else kod bloğu devreye girecektir. Bu kod bloğunda processRental() metodu yer almaktadır. Bu metot bünyesinde de service.rentACar(rental) şeklinde bir satır yer almaktadır. Service @Inject anotasyonu yardımı ile RentalController sınıfına enjekte edilmiş RentalService veri tipinde bir nesnedir. RentalServiceImpl ismini taşıyan ve RentalService interface sınıfını implemente eden sınıf kod 10.18 de yer almaktadır.

```
Kod 10.18 - RentalServiceImpl
```

```
package com.kurumsaljava.spring.service;

import javax.inject.Inject;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.
            Transactional;
import com.kurumsaljava.spring.dao.CarRepository;
import com.kurumsaljava.spring.dao.CustomerRepository;
import com.kurumsaljava.spring.dao.RentalRepository;
import com.kurumsaljava.spring.domain.Car;
import com.kurumsaljava.spring.domain.Customer;
import com.kurumsaljava.spring.domain.Rental;

@Service
public class RentalServiceImpl implements RentalService {

    @Inject
    private CustomerRepository customerRepository;
    @Inject
    private RentalRepository rentalRepository;
    @Inject
    private CarRepository carRepository;

    public RentalServiceImpl() {
    }

    @Transactional
    @Override
    public Rental rentACar(final Rental rental) {

        final Customer dbCustomer =
            customerRepository.getCustomerByName(
                    rental.getCustomer().getName());

        if (dbCustomer == null) {
            customerRepository.save(rental.getCustomer());
        }

        final Car car = carRepository.findCarById(1);

        rental.setCar(car);
        rental.setRented(true);
        rental.setCustomer(rental.getCustomer());
        rentalRepository.save(rental);
        return rental;
    }

    public void setCustomerRepository(
            final CustomerRepository customerRepository) {
```

```
        this.customerRepository = customerRepository;
    }

    public void setRentalRepository(
            final RentalRepository rentalRepository) {
        this.rentalRepository = rentalRepository;
    }

    public void setCarRepository(
            final CarRepository carRepository) {
        this.carRepository = carRepository;
    }
}
```

RentalServiceImpl uygulamanın servis katmanında yer alan bir sınıftır (bknz. resim 10.5). @Service anotasyonu aracılığı ile bir Spring bean haline gelir. Spring konfigürasyon dosyasında context:component-scan konfigürasyon elementi kullanıldığı taktirde Service, @Component ve @Repository gibi anotasyonu taşıyan sınıflar Spring tarafından @Inject anotasyonu ile talep edilen sınıflara enjekte edilir. Gösterim katmanında yer alan RentalController sınıfı kendisine enjekte edilen RentalService nesnesi aracılığı ile araç kiralama işlemini gerçekleştirmektedir. Görüldüğü gibi Spring MVC kendi bünyesinde Spring dependency injection mekanizmalarını kullanabilmektedir.

Kod 10.18 de yer alan RentalServiceImpl sınıfına yine @Inject anotasyonu kullanılarak veri katmanında yer alan repository sınıfları enjekte edilmektedir. Kullanılan repository sınıflarından birisi olan CustomerRepository sınıfının JPA implementasyonu kod 10.19 da yer almaktadır.

```
Kod 10.19 - JpaCustomerRepositoryImpl

package com.kurumsaljava.spring.dao.impl;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
import com.kurumsaljava.spring.dao.CustomerRepository;
import com.kurumsaljava.spring.domain.Customer;

@Repository
public class JpaCustomerRepositoryImpl implements
                CustomerRepository {

    @PersistenceContext
    private EntityManager entityManager;
```

```
    @Override
    public Customer getCustomerByName(final String name) {

        Customer result = null;
        try {
            result = (Customer) entityManager.
            createQuery("from Customer c where c.name=:name")
                    .setParameter("name", name)
                    .getSingleResult();

        } catch (final NoResultException e) {
        }
        return result;
    }

    @Override
    public void save(final Customer customer) {
        entityManager.persist(customer);
    }
}
```

Spring MVC ile Çoklu Konfigürasyon Kullanımı

Bir Spring MVC uygulaması için gerekli Spring konfigürasyonu birden fazla konfigürasyon dosyasına dağıtılabilir. Bu uygulama için gerekli konfigürasyonun sadeleşmesini ve daha anlaşılır hale gelmesini sağlayacaktır. Örneğin uygulamanın tüm konfigürasyonu gösterim katmanı için mvc-config.xml, servis katmanı için service-config.xml, veri katmanı için persistence-config.xml ismini taşıyan, birbirinden bağımsız konfigürasyon dosyalarından oluşabilir.

Gösterim katmanını konfigüre etmek için oluşturduğumuz mvc-config.xml isimli konfigürasyon dosyasının DispatcherServlet tarafından yüklenmesini sağlamak için kod 10.20 de yer alan servlet tanımlamasını yapmıştık.

```
Kod 10.20 - web.xml

<servlet>
    <servlet-name>rentacar</servlet-name>
    <servlet-class>org.springframework.web.servlet.
            DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
```

```
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Birden fazla Spring konfigürasyon dosyasını yüklemek için ContextLoaderListener sınıfı kullanılmaktadır. web.xml dosyasında bu sınıfı ihtiva eden bir listener tanımlaması kod 10.21 de yer almaktadır.

```
Kod 10.21 - web.xml

<listener>
    <listener-class>
        org.springframework.web.context.
                    ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener sınıfının istenilen konfigürasyon dosyalarını yükleyebilmesi için context-param elementi ile bu dosyalardan oluşan bir listenin tanımlanması gerekmektedir. Böyle bir tanımlama kod 10.22 de yer almaktadır.

```
Kod 10.22 - web.xml

<context-param>
    <param-name>
        contextConfigLocation
    </param-name>
    <param-value>
        /WEB-INF/service-config.xml
        /WEB-INF/persistence-config.xml
        classpath:other-config.xml
    </param-value>
</context-param>
```

Kod 10.22 de yer alan tanımlama ile WEB-INF dizinde yer alan service-config.xml ve persistence-config.xml konfigürasyon dosyaları, ayrıca classpath içinde yer alan other-config.xml dosyası ContextLoaderListener tarafından yüklenecektir. Context-param ile bir konfigürasyon dosyası listesi oluşturulmadığı taktirde ContextLoaderListener /WEB-INF/applicationContext.xml lokasyonundaki konfigürasyon dosyasını yüklemeye çalışacaktır. /WEB-INF/applicationContext.xml konfigürasyon dosyasında import komutu kullanılarak ta adı geçen diğer konfigürasyon

dosyaları yüklenebilir.

@RequestParam Anotasyonu Kullanımı

@RequestParam anotasyonu kullanıcı isteği ile gelen parametrelerin metot parametrelerine atanması için kullanılmaktadır. Kod 10.23 de @RequestParam anotasyonunu kullanan redirectToPage() metodu yer almaktadır. @RequestParam("page") final String page ile pageName değişkenine bir kullanıcı isteği parametresi olan (request parameter) page'in degeri atanmaktadır. redirectToPage() metodu kullanıcının page parametresi ile tayin ettiği sayfanın gösterilmesini sağlamaktadır.

```
Kod 10.23 - RentalController

@RequestMapping(value = "/redirect", method =
            RequestMethod.GET)
public String redirectToPage(@RequestParam("page")
        final String pageName, final ModelMap model) {
    return pageName;
}
```

Kod 10.23 de yer alan metodun işlem yapabilmesi için uygulamanın http://localhost/redirect?page=list şeklinde çağrılması gerekmektedir. ?page=list şeklindeki ifade page ismini taşıyan bir kullanıcı isteği parametresi oluşturur. Bu parametrenin degeri list dir. WEB-INF dizininde list.jsp ismini taşıyan bir JSP sayfası bulunması durumunda, uygulama redirectToPage() metodu aracılığı ile bu sayfanın gösterilmesini sağlayacaktır.

redirectToPage() metodu kullanıcı isteği bünyesinde page parametresinin mevcudiyeti durumunda işlem görür. Eğer page parametresinin eksik olduğu kullanıcı isteklerinde de redirectToPage() metodunun işlem görmesi bekleniyorsa, @RequestParam(value = "page", required = false) şeklinde bir tanımlama yapılması gerekmektedir.

@PathVariable Anotasyonu Kullanımı

@PathVariable anotasyonu aracılığı ile kullanıcı isteğini temsil eden web adresin belli parçalarına erişmek ve bunları parametre olarak değerlendirmek mümkündür. Kod 10.24 de yer alan örnekte web adresin bir parçası olan userid parametresi yardımı ile bir kullanıcının kiraladığı araçların listesi

oluşturulmaktadır. Bu web adresi http://localhost/list/1000 ya da http://localhost/list/200 şeklinde olabilir. 1000 ya da 200 rakamları kullanıcının veri tabanındaki kayıt anahtarıdır (primary key). @PathVariable anotasyonu bu değerin web adresinden edinilerek userid isimli metot parametresine eşitlendikten sonra metot gövdesinde kullanılmasını mümkün kılmaktadır.

```
Kod 10.24 - RentalController

@RequestMapping(value="/list/{userid}", method =
        RequestMethod.GET)
public String listRentals(@PathVariable("userid") String userid,
            final ModelMap model) {
    List<Rental> rentanList = rentalService.listRentalsOfUser(
                    userid);
    model.addAttribute("rentalList", rentalList);
    return "list";
}
```

Uygulama tarafından web adresin bir parçasının parametre olarak algılanabilmesi için bu adresin hangi kısmının parametre olduğunun @RequestMapping anotasyonu ile tanımlanması gerekmektedir. Kod 10.24 de yer alan örnekte /list/{userid} ile bu tanımlama yapılmaktadır.

@RequestMapping(value="/list/{userid}" şeklinde bir tanımlama yapıldığı taktirde @PathVariable anotasyonunun parametre ismini taşıması gerekli değildir. Bu sebepten dolayı kod 10.24 de yer alan listRentals(@PathVariable("userid") String userid, final ModelMap model) metot tanımlaması listRentals(@PathVariable String userid, final ModelMap model) olarak yazılabilir.

```
Kod 10.25 - RentalController

@RequestMapping(value="/user/{userid}/rental/{rentalid}", method =
            RequestMethod.GET)
public String listRentals(@PathVariable("userid") String userid,
                    @PathVariable("rentalid") String rentalid,
            final ModelMap model) {
    ...
}
```

Kod 10.25 de görüldüğü gibi metot parametrelerini oluştururken birden fazla @PathVariable kullanmak mümkündür. Örneğin 50 numaralı müşterinin 1 numarasını taşıyan araç kiralama işlemine ulaşmak için /user/50/rental/1

şeklinde bir web adresi kullanılabilir. listRentals() metodu bünyesinde userid (50) ve rentalid (1) değişkenleri ile web adresinde yer alan parametre değerlerine erişmek mümkündür.

@PathVariable ile int, long, Date gibi herhangi bir veri tipi kullanılabilir. Spring otomatik olarak veri tipi dönüşümünü sağlamaktadır.

Spring MVC Tarafından Tüketilebilecek Veri Türleri

Bir Spring MVC uygulaması String, XML, JSON, Byte gibi değişik türdeki verileri işleyecek şekilde yapılandırılabilir. Uygulamanın kullanılış tarzına ve kullanıcı tipine göre işlenen veri türü değişiklik gösterebilir. İncelediğimiz araç kiralama formu örneğinde kullandığımız veri türü String veri tipinde idi. Bu veriler HttpServletRequest sınıfı aracılığı ile uygulamaya taşındı.

Seçilen veri türünü tespit etmek için uygulama HttpServletRequest nesnesinde yer alan Content-Type başlık (request header) parametresinin değerini inceler. Form örneğinde bu parametrenin sahip olduğu değer application/x-www-form-urlencoded şeklindedir. XML veri türü için bu değer application/xml ya da text/xml, Json için application/json şeklindedir. Görüldüğü gibi veri türünü kullanıcı (client) tayin etmektedir.

Bir Spring MVC uygulamasında @RequestMapping anotasyonu kullanılarak işlenmek istenen veri türü tayin edilmektedir. Bunun yanı sıra RequestBody anotasyonu yardımı ile kullanıcı tarafından uygulamaya gönderilen verinin bir metot parametresine eşlenme işlemi gerçekleştirilir. Kod 10.26 da yer alan örnekte JSON olarak gönderilen veri @RequestBody anotasyonu ile bir Rental nesnesine dönüştürülmektedir. Sadece Content-Type parametresinin application/json değerini taşıdığı durumlarda kod 10.26 da yer alan addRental() metodu devreye girmektedir. Bunun kontrolü @RequestMapping anotasyonunun consumes elementi ile yapılmaktadır.

```
Kod 10.26 - RentalController

@RequestMapping(value="/rental", method =
        RequestMethod.POST consumes="application/json")
public String addRental(final @RequestBody Rental rental,
            final ModelMap model) {
    ...
}
```

HttpServletRequest nesnesinde yer alan verilerin istenilen türde bir alan nenesine dönüştürülebilmesi için HttpMessageConverter sınıfı kullanılmaktadır. Bu sınıf kullanıcı verilerinin bir alan nesnesine, alan nesnelerinin de kullanıcıya gönderilen cevaba (HttpServletResponse) dönüştürülmesini mümkün kılmaktadır. Kullanılabilecek bazı HttpMessageConverters implementasyonları şöyledir:

- ***ByteArrayHttpMessageConverter*** - byte array dönüşümü
- ***StringHttpMessageConverter*** - string dönüşümü
- ***FormHttpMessageConverter*** - form <-> MultiValueMap<String, String> dönüşümü
- ***MarshallingHttpMessageConverter*** - org.springframework.oxm paketi yardımı ile Java<->XML dönüşümü

RequestBody tarafından gerekli dönüşümün yapılabilmesi için konfigürasyon dosyasında RequestMappingHandlerAdapter sınıfının Spring bean olarak tanımlanması gerekmektedir. Kod 10.27 de yer alan örnekte String ve XML bazlı veri türlerinin dönüşümü için gerekli RequestMappingHandlerAdapter konfigürasyonu yer almaktadır.

```
Kod 10.27 - applicationContext.xml

<bean
     class="org.springframework.web.servlet.mvc.method.
                annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
          <util:list id="beanList">
            <ref bean="stringConverter"/>
            <ref bean="xmlConverter"/>
          </util:list>
    </property>
</bean>

<bean id="stringConverter"
   class="org.springframework.http.converter.
            StringHttpMessageConverter"/>

<bean id="xmlConverter"
   class="org.springframework.http.converter.xml.
        MarshallingHttpMessageConverter">
      <property name="marshaller" ref="castorMarshaller" />
      <property name="unmarshaller" ref="castorMarshaller" />
</bean>
```

```
<bean id="castorMarshaller" class="org.springframework.oxm.castor.
            CastorMarshaller"/>
```

Spring MVC Tarafından Oluşturulabilecek Veri Türleri

Bir Spring MVC uygulaması bir önceki bölümde ismi geçen veri türlerini tüketebildiği gibi, kullanıcılarına bu veri türlerinden oluşan cevaplar da sunabilir. Kod 10.28 de yer alan örnekte kullanıcıya bir Rental nesnesi Json formatında sunulmaktadır. Sunulan veri türünü belirlemek için @RequestMapping anotasyonunun produces elementi kullanılmaktadır.

```
Kod 10.28 - RentalController

@RequestMapping(value="/user/{userid}/rental/{rentalid}",
          method = RequestMethod.GET produces="application/json")
public Rental listRentals(@PathVariable("userid") String userid,
                      @PathVariable("rentalid") String rentalid,
                                      final ModelMap model) {
    ...
}
```

İç ve Dış Yönlendirme

JSP sayfalarına (view) yönlendirme işlemi controller sınıflarının bir JSP sayfasının ismini geri vermesi ile sağlanmaktadır. Kod 10.24 yer alan örnekte metodun geri verdiği değer list şeklindedir. Spring MVC InternalResourceViewResolver yardımı ile list.jsp ismini taşıyan JSP sayfasını lokalize ederek akış kontrolünü bu sayfaya bırakmaktadır. Bu yönlendirme Servlet API'sinde yer alan RequestDispatcher.forward() ya da RequestDispatcher.include() kullanılarak yapılmaktadır. Bu işlem iç yönlendirme (internal redirect) olarak isimlendirilmektedir, çünkü yönlendirme uygulama bünyesinde yapılmaktadır.

Bazı şartlar altında dış yönlendirme (external direct) gerekli olabilir. Örneğin bir controller sınıfından başka bir controller sınıfına iç yönlendirme yapıldığı zaman, birinci controller sınıfına kullanıcı tarafından gönderilmiş tüm veriler ikinci controller sınıfına da yönlendirilir. Bu beklentilerin dışında bir veri akışı olduğunda ikinci controller sınıfı için kafa karıştırıcı olabilir. Bunu engellemek

için ikinci controller sınıfına dışsal yönlendirme yapılabilir.

Kod 10.29 da yer alan örnekte form için gerekli işlem yapıldıktan sonra redirect:rental/done ile bu adresten sorumlu controller sınıfı için dış yönlendirme yapılmaktadır.

```
Kod 10.29 - RentalController

@RequestMapping(method = RequestMethod.POST)
public String processSubmit(final ModelMap model,
        @ModelAttribute(RENTAL_FORM) @Valid final Rental rental,
        final BindingResult result) {
    if (result.hasErrors()) {
        buildCarList(model);
        return RENTAL_VIEW;
    } else {
        processRental(rental);
        return "redirect:rental/done";
    }
}
```

rental/done adresinden DoneController sınıfı sorumlu olduğundan, done() metodu bünyesinde done.jsp sayfasına iç yönlendirme yapılmaktadır. Dış ve iç yönlendirme gerçekleştikten sonra kullanıcının web tarayıcısının adres çubuğunda http://localhost/rental/done adresi yer alır.

```
Kod 10.30 - DoneController

@Controller
public class DoneController {

    @RequestMapping(value = "/rental/done")
    public String done(final ModelMap model) {
        return "done";
    }
}
```

Dış yönlendirme için ayrıca RedirectView sınıfı kullanılabilir. Kod 10.30.1 de yer alan örnekte /rental/list sayfasına dış yönlendirme yapılmaktadır. DispatcherServlet RedirectView ile karşılaştığı durumlarda HttpServletResponse.sendRedirect() aracılığı ile ismi geçen sayfaya dış yönlendirme yapar.

```
Kod 10.30.1 - RentalController
```

```
@RequestMapping(method = RequestMethod.GET)
public RedirectView redirect(final ModelMap model)
            throws IOException {
        RedirectView redirectView =
            new RedirectView("rental/list");
        redirectView.addStaticAttribute("errorMessage",
                    "error1");
        return redirectView;
}
```

addStaticAttribute() metodu ile RedirectView nesnesine eklenen tüm parametreler dış yönlendirme yapılan adresin parametreleri hale gelir. Kod 10.30.1 de yer alan örnekte dış yönlendirme adresi /rental/list?errorMessage?error1 şeklindedir. /rental/list adresinden sorumlu olan controller sınıfı metodu @RequestParam anotasyonu ile bu parametrelere erişebilir. Kod 10.30.2 de yer alan getList() metodu rental/list adresinden sorumlu olduğu için @RequestParam("errorMessage") aracılığı ile errorMessage parametresini metot parametresi olarak tanımlamaktadır. Böylece dış yönlendirme ile gönderilen errorMessage parametresi getList() bünyesinde kullanılır hale gelmektedir.

```
Kod 10.30.2 - RentalController

@RequestMapping(value = "rental/list")
public String getList(
                @RequestParam("errorMessage") String
                            errorMessage, Model model) {
    model.addAttribute("errorMessage", errorMessage);
    return "list";
}
```

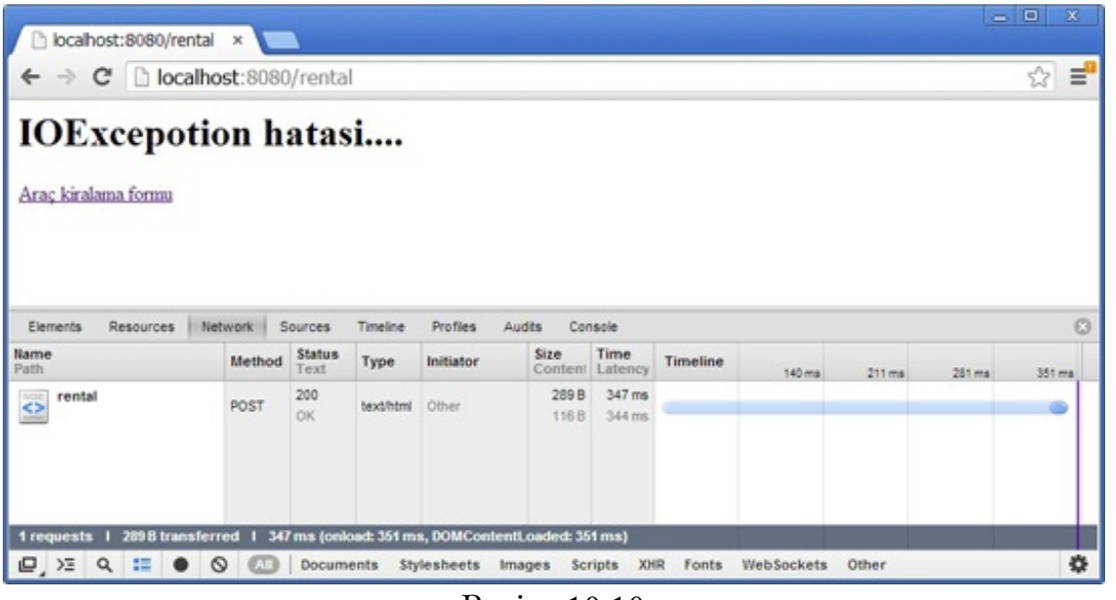
Hata Yönetimi

Bir Spring MVC uygulaması bünyesinde işlem esnasında çeşitli hatalar (exception) oluşabilir. SimpleMappingExceptionResolver sınıfı kullanılarak her hata için bir gösterim sayfası tanımlanabilir.

```
Kod 10.31 - applicationContext.xml

<bean id="exceptionResolver"
      class="org.springframework.web.servlet.handler.
                        SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
      <map>
```

```
            <entry key="java.io.IOException"
                   value="io-exception" />
            <entry key="java.lang.Exception"
                   value="generic-error" />
        </map>
    </property>
</bean>
```

Kod 10.31 de yer alan örnekte IOException oluşması durumunda io-exception (resim 10.10) view elementine yönlendirme yapılmaktadır. Oluşan diğer hatalar için yönlendirme adresi generic-error şeklinde olacaktır.



Resim 10.10

Resim 10.10 da görüldüğü gibi SimpleMappingExceptionResolver oluşan hata türüne göre gerekli sayfaya yönlendirme yapmaktadır, lakin HTTP statü kodu oluşan hata türünü yansıtmamaktadır. Statü kodunun 200 değerine sahip olması, kullanıcı isteğinin uygulama tarafından hatasız cevaplandığı anlamına gelmektedir. Ama biliyoruz ki bir IOException hatası oluşmuştur.

Bazı kullanıcılar (client) HTTP statü kodlarını inceleyerek oluşan hatalardan haberdar olma isteğinde olabilir. Bu durumda uygulama sunucusunun belli bir hata için belli bir sayfaya yönlendirme yapması yeterli olmayacaktır. Oluşan hata türüne göre kullanıcıya gönderilen cevap içinde HTTP statü kodunun tanımlanmış olması gerekir. Oluşan hata türüne göre HTTP statü kodunu atamak için DefaultHandlerExceptionResolver sınıfı kullanılmaktadır.

DefaultHandlerExceptionResolver sınıfı uygulama bünyesinde oluşan hata türüne göre statü kodunu atar. Bu sınıf mvc isim alanında yer aldığı için

konfigürasyon dosyasında tanımlanmasına gerek yoktur ve mvc isim alanının yüklenmesiyle aktif hale gelir. Aşağıda yer alan listede bazı hata türleri ve DefaultHandlerExceptionResolver tarafından atanan statü kodları yer almaktadır.

```
Liste 10.1:

BindException - 400 (Bad Request)
ConversionNotSupportedException - 500 (Internal Server Error)
HttpMediaTypeNotAcceptableException - 406 (Not Acceptable)
HttpMediaTypeNotSupportedException - 415 (Unsupported Media Type)
HttpMessageNotReadableException - 400 (Bad Request)
HttpMessageNotWritableException - 500 (Internal Server Error)
HttpRequestMethodNotSupportedException - 405 (Method Not Allowed)
MethodArgumentNotValidException - 400 (Bad Request)
MissingServletRequestParameterException - 400 (Bad Request)
MissingServletRequestPartException - 400 (Bad Request)
NoSuchRequestHandlingMethodException - 404 (Not Found)
TypeMismatchException - 400 (Bad Request)
```

DefaultHandlerExceptionResolver sınıfı hata durumunda sadece statü kodunu atamakla yetinir. Statü kodu haricinde kullanıcıya herhangi bir bilgi gönderilmez. Kullanıcının oluşan hatadan detaylı olarak haberdar olabilmesi için sadece statü kodunun atanması yeterli değildir. Oluşan hatanın çıktısı (stacktrace) kullanıcıya hata mesajı (error response) olarak gönderilmelidir. Bu amaçla @ExceptionHandler anotasyonu kullanılabilir.

Kod 10.32 yer alan örnekte processSubmit bünyesinde IOException tipinde bir hata oluşması durumunda handleIOException() metodu devreye girerek bir RedirectView nesnesi oluşturur. Bu nesne bünyesinde dış yönlendirme yapılacak sayfanın adresi (rental/ioexception) ve hata mesajı (errorMessage) yer almaktadır. Spring MVC /rental/ioexception için gerekli yönlendirmeyi gerçekleştirir. /rental/ioexception sayfası için bir istek gerçekleştiği için bu istek tekrar RentalController sınıfına yönlendirilir, çünkü RentalController bünyesinde @RequestMapping anotasyonu kullanılarak ioexception adresinden sorumlu errorRedirectPage() isimli bir metot tanımlanmıştır. Bu metot bünyesinde model nesnesine istek parametresi (request parameter) olarak gelen hata mesajı (errorMessage) yerleştirilir ve io-exception yani io-exception.jsp sayfasına hata gösterimi için iç yönlendirme yapılır.

```
Kod 10.32 - RentalController

@Controller
```

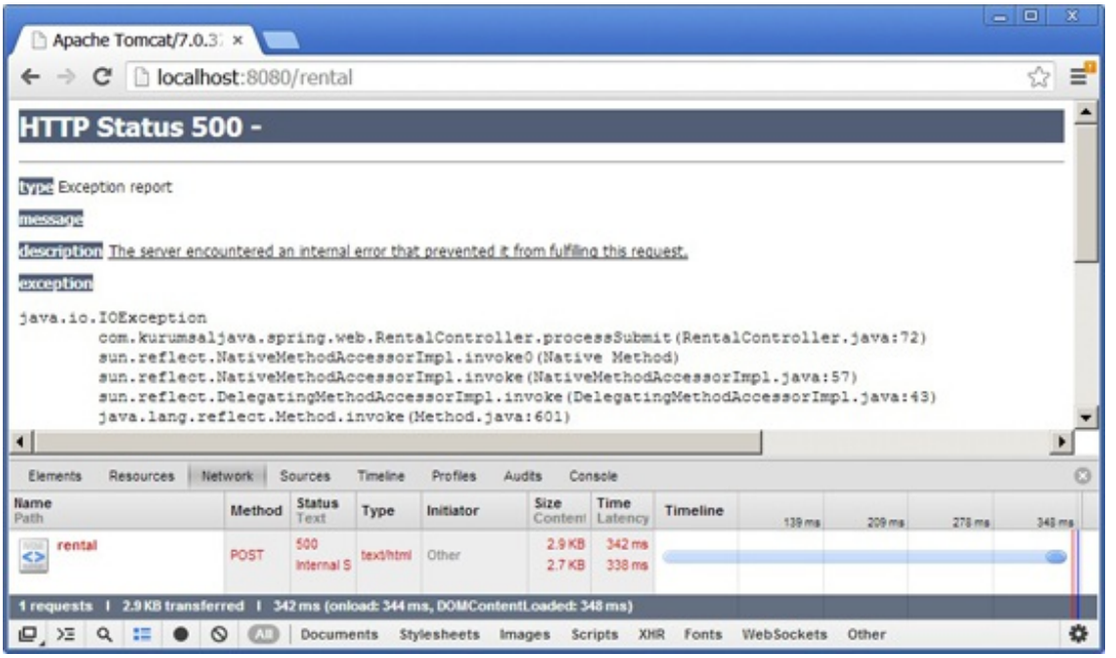
```
@RequestMapping("/rental")
public class RentalController {

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(final ModelMap model,
        @ModelAttribute(RENTAL_FORM) @Valid final Rental rental,
            final BindingResult result) {
        ...
    }

    @ExceptionHandler(IOException.class)
    public RedirectView handleIOException(IOException ex)
            throws IOException {
        RedirectView redirectView =
            new RedirectView("rental/ioexception");
        redirectView.addStaticAttribute("errorMessage",
                ex.getMessage());
        return redirectView;
    }

    @RequestMapping(value = "ioexception")
    public String errorRedirectPage(
                    @RequestParam("errorMessage")
                       String errorMessage, Model model) {
        model.addAttribute("errorMessage", errorMessage);
        return "io-exception";
    }
}
```

@ExceptionHandler anotasyonuna ek olarak HTTP statü kodu atanmak istendiğinde @ResponseStatus anotasyonu kullanılabilir. Kod 10.32.1 de yer alan örnekte IOException türünde bir hatanın oluşması durumunda rental/ioexception adresine dış yönlendirme yapılmakta ve HTTP statü kodu 500 (INTERNAL_SERVER_ERROR) olarak atanmaktadır.

```
Kod 10.32.1 - RentalController

@ExceptionHandler(IOException.class)
@ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
public RedirectView handleIOException(IOException ex)
            throws IOException {
    RedirectView redirectView =
            new RedirectView("rental/ioexception");
    redirectView.addStaticAttribute("errorMessage",
            ex.getMessage());
    return redirectView;
}
```

Genel Hata Sayfası Konfigürasyonu

Uygulama tarafından aktif olarak herhangi bir hata yönetimi uygulanmıyorsa, oluşan bir hata kullanıcıya resim 10.11 deki gibi yansıyacaktır.



Resim 10.11

Resim 10.11 de yer alan hata sayfası geliştiriciler için önemli bilgiler ihtiva etmekle birlikte son kullanıcılar için kafa karıştırıcı türdedir. Son kullanıcılar için genel bir hata sayfasının tanımlanması ve oluşan hata hakkında bu hata sayfasında bilgi verilmesi kullanıcının daha az irite olmasını sağlayacaktır.

Oluşan herhangi bir hataya işaret etmek için error-page elementi kullanılarak web.xml bünyesinde genel bir hata sayfası tanımlanabilir. Uygulama sunucusu statü kodunun bir hata değerini ihtiva etmesi durumunda error-page ile tanımlı olan sayfaya yönlendirme yapacaktır. Kod 10.33 de genel bir hata sayfasının tanımlanması yer almaktadır.

```
Kod 10.33 - web.xml

<error-page>
    <location>/error</location>
</error-page>
```

Kod 10.34 de yer alan ErrorController sınıfı /error adresinden sorumludur ve uygulama sunucusunun /error adresine yönlendirme yapmasıyla birlikte aktif

hale gelir. handle() metodu bünyesinde hata verilerini ihtiva eden bir model nesnesi oluşturulduktan sonra, bu verilerin gösterimi amacıyla errorPage.jsp sayfasına iç yönlendirme yapılır. @ResponseBody anotasyonununu on ikinci bölümde inceleyeceğiz. @ResponseBody handle() metodunun geriye verdiği değerin HTTP cevap gövdesine (response body) eklenmesini sağlamaktadır.

```
Kod 10.34 - ErrorController

@Controller
public class ErrorController {

    @RequestMapping(value="/error")
    @ResponseBody
    public String handle(ModelMap model,
            HttpServletRequest request) {
        Map<String, Object> map =
            new HashMap<String, Object>();
        map.put("status",
            request.getAttribute("
                    javax.servlet.error.status_code"));
        map.put("reason", request.getAttribute("
                    javax.servlet.error.message"));
        model.put("errorMap", map);
        return "errorPage";
    }
}
```

Spring MVC uygulamalarında hata yönetimi için daha detaylı bilgiyi kitabın on ikinci, REST uygulamalarında hata yönetimi bölümünde bulabilirsiniz. REST uygulamaları Spring MVC çatısı kullanılarak geliştirildiğinden, kullanılan hata yönetimi mekanizmaları aynıdır.

14. Bölüm

Sürekli Entegrasyon

Giriş

Sürekli entegrasyon (Continuous Integration = CI) kod üzerinde yapılan her değişikliğin ardından tüm sistemin çalışır durumda olduğunu ve yapılan değişikliğin sistemin bazı bölümlerinde kırılmalara yol açmadığını tespit etmek için kullanılan yöntemdir. Kırılmaları tespit edebilmek için testlere ihtiyaç duyulmaktadır. Bu testler yapılan değişikliğin neticesi olarak yeni bir yapı (build) hazırlandıktan sonra otomatik olarak çalıştırılır. Yapılan değişiklik yeni yapının bir parçası olduğu için testlerde oluşan hatalar yapılan değişikliğin sistemi kırdığı anlamına gelmektedir. Bu durumdan tüm programcılar haberdar edilerek, hatanın bir an önce giderilmesi ve testlerin her zaman olumlu sonuç vermesi sağlanır. Sürekli entegrasyon ile programcılar tarafından kod üzerinde yapılan çalışmalar neticesinde her zaman çalışır bir sürümün oluşması sağlanmış olur.

Martin Fowler bu konuda yazdığı makalesinde sürekli entegrasyonu şu şekilde tanımlıyor:

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significAntly reduced integration problems and allows a team to develop cohesive software more rapidly.

Şu şekilde tercüme edilebilir:

Sürekli Entegrasyon programcı ekibi tarafından yapılan değişikliklerin sık aralıklarla sisteme entegre edilmesi için kullanılan yazılım geliştirme yöntemidir. Her programcı günde en az bir kere yaptığı değişiklikleri entegre eder. Bu sayede gün boyunca birden fazla entegrasyon gerçekleşir. Otomatik sürüm oluşturulması ve mevcut testler yardımı ile entegrasyon kontrol edilir. Birçok ekip tarafından sürekli entegrasyon metodu ile entegrasyon sorunlarının azaldığı tespit edilmiştir. Sürekli entegrasyon ile bir ekip hızlı bir şekilde kendi içinde bütün ve çalışır programlar yazabilmektedirler.

Çevik süreçlerde sürekli entegrasyon önemli bir rol oynamaktadır. Sürekli entegrasyon aracılığıyla proje bünyesinde geri bildirim (feedback) kuvvetlendirilir. Programcılar projenin içinde bulunduğu durum hakkında her

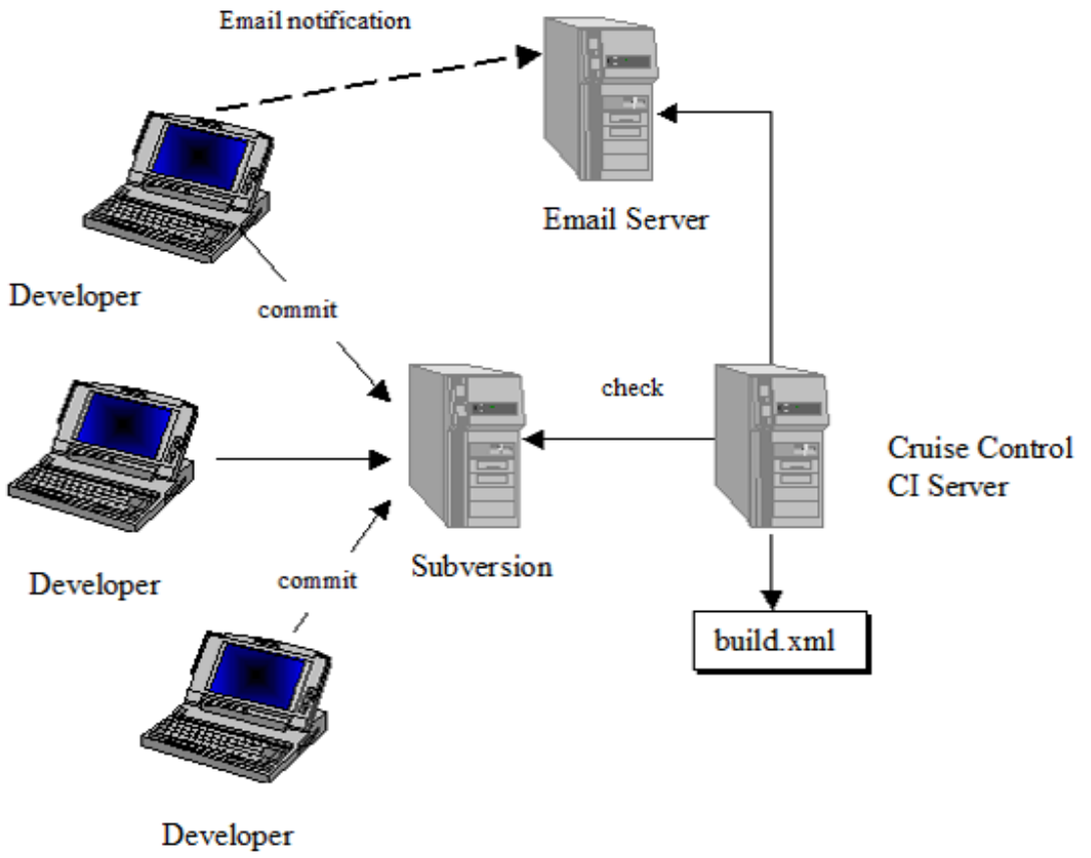
gün gerçekleşen entegrasyon sayesinde haberdar olurlar. Olumsuz netice veren bir entegrasyon program hatalarının oluştuğu mesajını verir ve bu programcılar için bir geri bildirimdir. Hata hemen giderilerek, yeniden entegrasyon süreci başlatılır. Bu süreç proje bitimine kadar geçerliliğini yitirmez.

Sürekli Entegrasyon Nasıl Çalışır?

Sürekli entegrasyon yöntemini uygulayabilmek için teknik altyapının oluşturulması gerekmektedir. Bunun için gerekli araçlar:

1. Sürekli Entegrasyonu otomatik olarak gerçekleştiren bir sunucu, örneğin Cruise Control ya da Jenkins
2. Versiyon yönetim sistemi, örneğin Subversion
3. Geri bildirim için kullanılmak üzere bir email sunucusu
4. Otomatik yapılandırma aracı, örneğin Ant ya da Maven
5. JUnit Testleri

İlk etapta sürekli entegrasyonun nasıl çalıştığına göz atalım.



Resim 14.1 Sürekli entegrasyon için gerekli altyapı

Sürekli entegrasyonu benimsemiş bir ekip, aşağıdaki şekilde çalışır:

1. Her programcı sabah mesaiye başladıktan sonra versiyon kontrol sisteminden projenin en son halini alır (update).
2. Projeyi yapılandırmak (build) için Ant ya da Maven kullanılır. Programcı yaptığı değişiklikleri kontrol etmek için kendi bilgisayarında yapı oluşturma programını çalıştırır. Olumlu sonuç aldığı taktirde değişiklikleri versiyon control sistemine gönderir (commit). Olumsuz netice alması durumunda hataları giderir ve tekrar yapı oluşturma programını çalıştırır. Bu işlemi olumlu sonuç alana kadar tekrarlar. Entegrasyonu sağlayabilmek için yazdığı kodun çalışır durumda olması gerekmektedir. Hiçbir programcı çalışmayan kodu versiyon kontrol sistemine eklemez.
3. Commit işleminden sonra sürekli entegrasyonu gerçekleştiren sunucu versiyon kontrol sisteminden en son değişiklikleri alarak, yapı oluşturma programını (örneğin Ant) çalıştırır. Bu program ile tüm proje yapılandırılır ve testler çalıştırılır. Hata oluşması durumunda sürekli entegrasyon sunucusu tüm programcılara email göndererek hatayı bildirir (feedback). Hata oluşmaması durumunda entegrasyon olumlu sonuç vermiştir ve sürekli entegrasyon sunucusu daha sonra göreceğimiz gibi yeşil ışık yakarak, durumu dolaylı olarak programcılara bildirir.
4. Sürekli entegrasyon sunucusu versiyon kontrol sistemini belirli aralıklarla kontrol etmeye devam eder. Bir değişiklik olduğunu tespit ettiği zaman otomatik olarak tekrar yapı oluşturma programını çalıştırarak tüm projeyi yapılandırır ve sonuçları beyan eder.

Bu süreçte yapılandırma (build) önemli bir rol oynamaktadır. Yapılandırma işlemi esnasında tüm Java sınıfları derlenir ve testler çalıştırılır. Bu işlem için Ant gibi otomatik yapı (build) oluşturma araçlarından faydalanılır. Yapılandırma işlemi esnasında kodun kalitesini ölçmek için JDepend, PMD, FindBugs, EclEmma ve CheckStyle gibi araçlardan faydalanılabilir. Bu işlemler Ant tarafından otomatik olarak yapılabileceği için programcı için ek yük oluşmamaktadır. Bu araçların çevik süreçlerde nasıl kullanıldığını detaylı olarak bir sonraki bölümde inceleyeceğiz.

Sürekli Entegrasyon ve Geri Bildirim

Sürekli entegrasyonun önemli bacaklarından birisini de geri bildirim (feedback) mekanizması oluşturur. Entegrasyon ve oluşan hatalardan haberdar olmayan bir programcı için sürekli entegrasyon ne kadar anlam taşıyabilir? Bu sorunun

cevabını beraber arayalım.

Entegrasyon için tüm ekip beraber çalışmak zorundadır, çünkü entegrasyon birden fazla program parçasının bir araya getirilip, bir bütün oluşturulması işlemidir. Bu süreç içinde hatalar oluşabilir. Entegrasyonun başarılı olabilmesi için oluşan hataların giderilmesi gerekmektedir. Bu işlem tüm hatalar ortadan kaldırılana kadar devam eder. Sonuç olarak çalışan ve entegre edilmiş bir sistem oluşur. Entegrasyon boyunca oluşan hatalar programcılar için bir nevi geri bildirimdir, çünkü sadece bu sayede sistemin ne durumda olduğunu anlayıp, hataları düzeltebilirler. Buradan başarılı bir entegrasyon için bir geri bildirim mekanizmasına ihtiyaç duyulduğu sonucunu çıkartabiliriz.

Sürekli entegrasyon otomatize edilmiş bir süreçtir. Cruise Control gibi sürekli entegrasyon için kullanılan bir araç, email aracılığıyla oluşan hataları ve yapı sonuçlarını programcılara bildirir. Bu geri bildirim sayesinde programcılar sürekli entegrasyon işleminin hangi safhada olduğunu anlayabilirler. Email haricinde başka geri bildirim mekanizmaları kullanılabilir. Aşağıda yer alan resimlerde geri bildirim tamamen görsel bir şekilde sağlanmıştır.

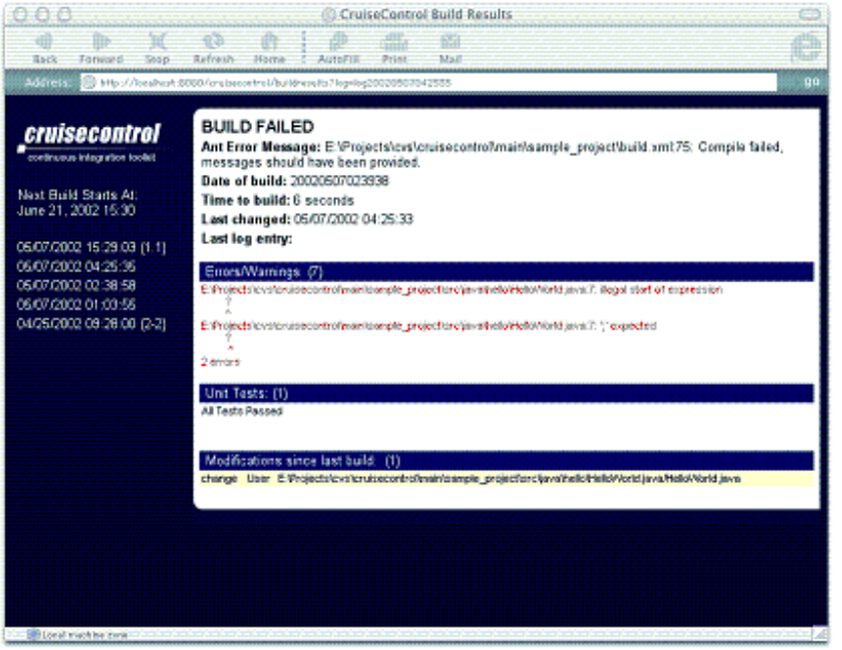
Resim 14.2 Değişik yapı statülerini gösteren lambalar

Resim 14.3 Yapının ne durumda olduğunu gösteren araç

Cruise Control

Cruise Control sürekli entegrasyon yapılabilecek altyapıyı ihtiva eden bir araçtır. Cruise Control sürekli entegre edilmek istenen projeler için konfigüre edildikten sonra Subversion gibi bir versiyon kontrol sisteminden projeler bünyesinde yapılan değişiklikleri tespit ederek, Ant skriptleri ile projeyi yapılandırır ve tanımlanan testleri çalıştırır.

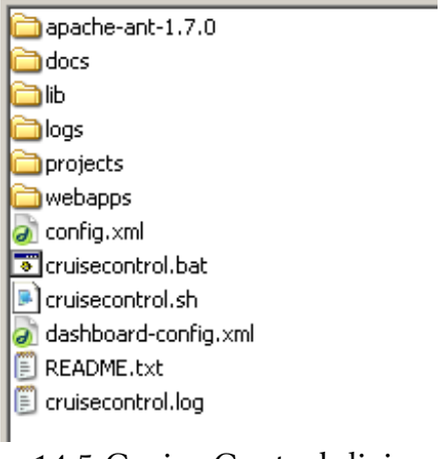
Cruise Control ihtiva ettiği pluginler aracılığıyla gereksinimler doğrultusunda konfigüre edilebilir. Cruise Control email notifikasyonu, Ant ve değişik tipte versiyon kontrol sunucuları entegrasyonu için pluginler ihtiva eder. Sunduğu web arayüzü ile projelerin statüleri takip edilebilir.



Resim 14.4 Cruise Control web arayüzü

Cruise Control Kurulumu

Güncel Cruise Control sürümünü bu adresten temin edebilirsiniz. Edindiğimiz zip dosyasını herhangi bir dizine açıyoruz.



Resim 14.5 Cruise Control dizin yapısı

Ana Cruise Control dizini içinde work isminde yeni bir dizin oluşturuyoruz. Bu dizin projemiz için gerekli konfigürasyonları ihtiva edecektir.

Resim 14.6 work dizin yapısı

work dizini içinde resim 14.6 da yer alan alt dizinleri oluşturuyoruz

- ***work/artifacts*** - Projeyi yapılandırma işlemi esnasında oluşan her türlü çıktı Cruise Control tarafından bu dizine yerleştirilir.
- ***work/checkout*** - Subversion versiyon kontrol sisteminde bulunan proje bu dizine yüklenir (checkout).
- ***work/logs*** - Yapılandırma esnasında yapılan işlemleri ihtiva eden raporlar bu dizine yerleştirilir.

work dizininde config.xml isminde bir konfigurasyon dosyası oluşturmamız gerekiyor. Bu dosyanın içeriği şu şekilde olmalıdır:

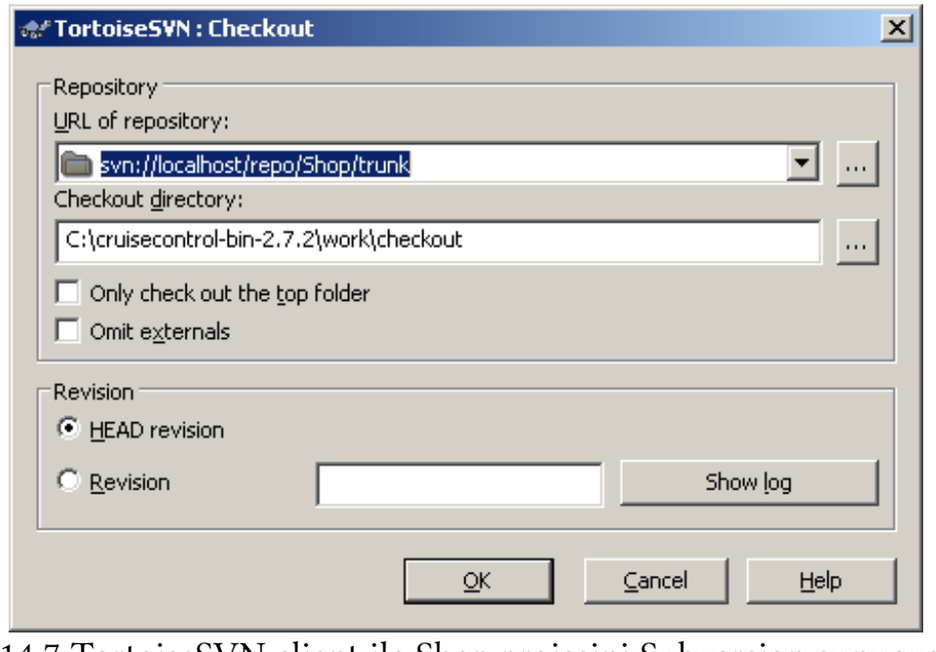
Kod 14.1 Cruise Control config.xml

```
<cruisecontrol>
  <project name="Shop" buildafterfailed="true">

    <bootstrappers>
      <currentbuildstatusbootstrapper
                file="logs/Shop/buildstatus.txt"/>
    </bootstrappers>

    <modificationset quietperiod="60">
      <svn localworkingcopy="checkout/Shop"/>
    </modificationset>

    <schedule interval="60">
      <ant Antscript="C:\cruisecontrol-bin-2.7.2\
                  apache-Ant-1.7.0\bin\Ant.bat"
           buildfile="build-Shop.xml"
           target="build"
           uselogger="true"
           usedebug="false"/>
    </schedule>
```

```
    <log dir="logs/Shop"/>


    <publishers>
      <currentbuildstatuspublisher
                file="logs/Shop/buildstatus.txt"/>

      <artifactspublisher
       dir="checkout/Shop/build/junit-report"
       dest="artifacts/Shop"/>
    </publishers>

  </project>

</cruisecontrol>
```

Cruise Control sunucusunu çalıştırmadan önce work/check dizinine Subversion sunucusunda olan projeyi indirmemiz (checkout) gerekiyor. Bunun için Subversion Client ile Subversion sunucusuna bağlanarak Shop projesinin bir kopyasını ediniyoruz.

Bu işlemin ardından work/checkout/Shop dizininde proje bünyesindeki dosyalar yer alır. config.xml dosyasını checkout/Shop dizini kullanacak şekilde konfigüre ediyoruz.



Resim 14.7 TortoiseSVN client ile Shop projesini Subversion sunucusundan alıyoruz (checkout)

Cruise Control konfigüre ettiğimiz projeyi yapılandırmak (build) için projenin

build.xml Ant skriptinden faydalanır. Proje bünyesindeki değişikliklerin Cruise Control tarafından kontrol edilebilmesi için aracı bir Ant skript oluşturuyoruz. Bunun bir örneğini kod 14.2 de görmekteyiz. config.xml bünyesinde schedule elementi ile kullandığımız build-Shop.xml önce svn (exec executable=”svn”) up (update) komutuyla work/checkouts dizininde bulunan Shop projesini update eder. Daha sonra kontrolü Shop/build.xml skriptine devrederek, projenin yapılandırılmasını sağlar.

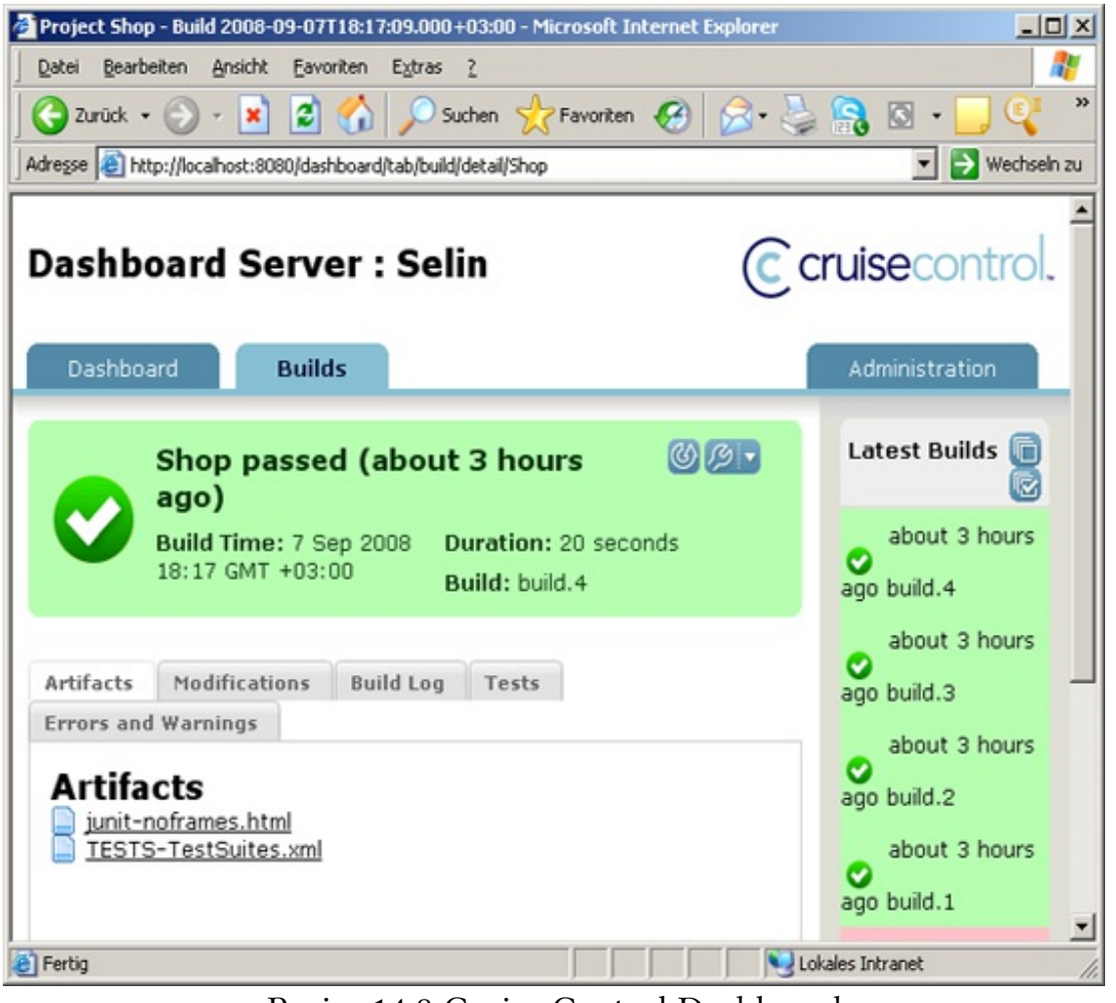
```
Kod 14.2   build-Shop.xml

<project name="build-Shop" default="build" basedir="checkout/Shop">
    <target name="build">

       <exec executable="svn">
           <arg line="up"/>
        </exec>

        <ant antfile="build.xml" target="run-junit"/>

     </target>
</project>
```

Eğer projenin build.xml skripti tarafından svn update işlemi yapılıyorsa, kod 14.2 de yer alan skripte ihtiyaç yoktur.

Bu işlemlerin ardında work dizinine giderek Cruise Control sunucusunu çalıştırabiliriz.

```
C:\cruisecontrol-bin-2.7.2\work>
        C:\cruisecontrol-bin-2.7.2\cruisecontrol.bat
```

Proje için oluşturduğumuz config.xml dosyasının kullanımını sağlamak için cruisecontrol.bat programının work dizini içinden çalıştırılması gerekmektedir.



Resim 14.8 Cruise Control Dashboard

http://localhost:8080/dashboard adresinden Cruise Control sunucu arayüzüne ulaşabiliriz.

Cruise Control her 60 saniyede bir Subversion sunucusunda bulunan Shop projesini kontrol edecek şekilde konfigüre edildi. Programcılar tarafından yapılan her değişiklik, Cruise Control tarafından bu zaman diliminde tespit edilir ve proje yeniden yapılandırılır. Resim 14.8 de Cruise Control tarafından başarıyla tamamlanan yapılar sağ kolonda yer almaktadır (build.4, build.3 vs.)

Email ile Geri Bildirim

Program hatalarından dolayı Cruise Control projeyi yapılandıramayabilir. Cruise Control Dashbaord arayüzü aracılığıyla projenin en son durumunu takip edilebiliriz. Kırılan yapılarda programcıları daha hızlı uyarmak için Cruise Control programcılara uyarı emaili gönderecek şekilde konfigüre edilebilir.

```
Kod 14.3  Cruise Control email notifikasyon
```

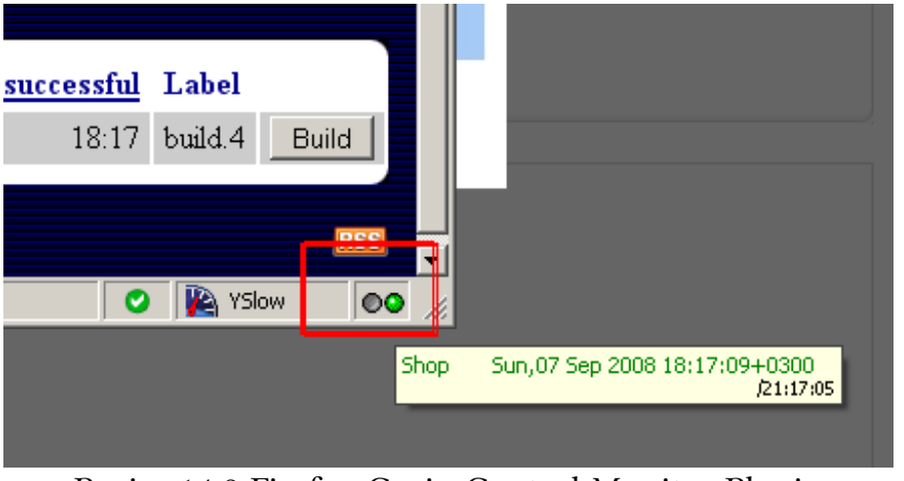
```
<email mailhost="localhost"
       returnaddress="cruise@mydomain.com"
       buildresultsurl="http://localhost/cc/buildresults/Shop"
       skipusers="true" spamwhilebroken="true">
    <always address="dev1@mydomain.com"/>
    <always address="dev2@mydomain.com"/>
    <failure address="failed-builds@mydomain.com"/>
</email>
```

Email notifikasyonu aktive etmek için kod 14.3 de bulunan satırları config.xml de bulunan publishers segmentine eklememiz yeterli olacaktır

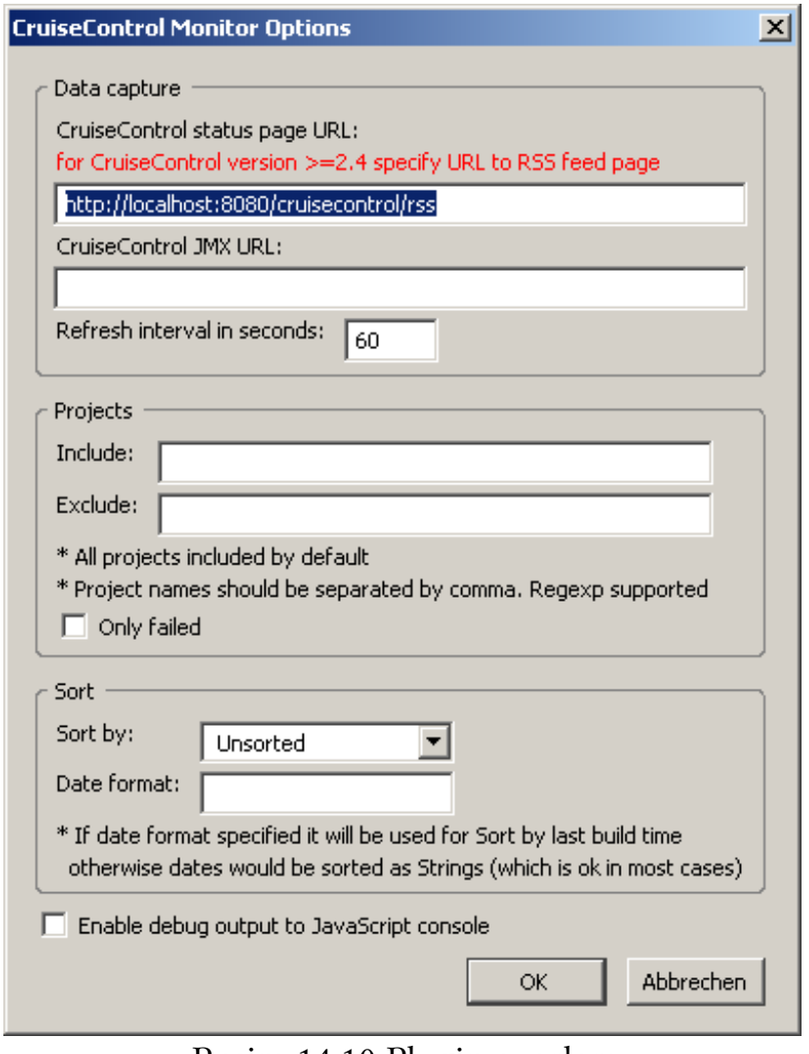
Firefox CruiseControl Monitor Plugin

https://addons.mozilla.org/en-US/firefox/addon/896 adresinde Firefox için ilginç bir plugin keşfettim. CruiseControl Monitor plugini aracılığıyla bir Cruise Control sunucusunda olup bitenleri Firefox web tarayıcısının sağ alt köşesindeki panelden takip edebiliriz. Burada bir kırmızı birde yeşil iki lamba bulunmaktadır. Cruise Control sunucusundaki projelerin statüsüne göre kırmızı ya da yeşil lamba yanarak, bizi projeler hakkında bilgilendirir.

Böylece devamlı Cruise Control Dashboard arayüzüne bakmak zorunda kalmadan, statü panelin üzerinden Cruise Control sunucusunda dolaylı olarak takip edebiliriz.



Resim 14.9 Firefox CruiseControl Monitor Plugin



Resim 14.10 Plugin ayarları

JUnit Testleri ve Sürekli Entegrasyon

Sürekli entegrasyon yapabilmek için entegre edilen sistem parçalarının test edilebilmesi gerekir. JUnit testleri olmayan bir projenin Cruise Control sunucusu tarafından sürekli entegre edilmesi bir anlam ifade etmez, çünkü entegrasyon neticesinin geri bildirimi yoktur. Cruise Control sadece sınıfları derlemekte yetinir ki bu da aslında Cruise Control gibi bir aracın kullanımı için yeterli bir argüman değildir.

Kitabın onuncu bölümünde implemente ettiğimiz login modülü için birim, entegrasyon ve onay/kabul tarzı test sınıfları oluşturmuştuk Bu testlerden birim tarzı olanlar en hızlı şekilde koşturulabilen testlerdir, çünkü bu testlerin dış dünyaya olan bağımlılıklarını mock nesneler kullanarak simüle ettik. Birim testlerini koşturmak için build.xml bünyesinde run-junit isminde bir hedef (target) tanımlamıştık. Bu hedef tüm sınıfları önce derledikten sonra

tanımladığımız test sınıfını koşturur ve test sonuçlarını HTML raporu haline getirir.

Birden fazla test sınıfını koşturabilmek için bir test suite oluşturmamız gerekiyor. Bunun bir örnegi kod 14.4 de yer almaktadır.

```
Kod 14.4  AllTests.java - JUnit Test Suite

package test.shop.junit;

@RunWith(Suite.class)
@SuiteClasses({LoginControllerTest.class,LoginManagerImplTest.class,
        LoginDaoImplTest.class  })
public class AllTests{

}
```

@SuiteClasses anotasyonu bünyesinde üç değişik test sınıfını bir araya getirerek, bu testlerin ardarda koşturulmalarını sağlayabiliriz.

```
Kod 14.5  build.xml - run-junit target

<target name="run-junit" depends="compile">

    <junit fork="false" failureproperty="tests.failed"
        showoutput="true" printsummary="yes"
        haltonfailure="yes">
        <classpath refid="compile.classpath" />
        <classpath path="${build.dir.classes}" />

        <test name="${junit.testcase.class}"
            haltonfailure="yes" outfile="build/junit-result">
            <formatter type="xml" />
        </test>
    </junit>

    <fail if="tests.failed">
    ******************************************************
    ******************************************************
    JUnit testlerinde hata olustu. Lütfen kontrol et
    ******************************************************
    ******************************************************
     </fail>

    <junitreport todir="${junit.report.dir}">
        <fileset dir="${build.dir}">
            <include name="junit-result.xml" />
```

```
        </fileset>
        <report format="noframes"
                todir="${junit.report.dir}" />
    </junitreport>
</target>
```

Oluşturduğumuz test suite sınıfını Ant build.xml dosyasında tanımladığımız run-junit hedefi bünyesinde ${junit.testcase.class} değişkeni aracılığıyla kullanabiliriz. Bu değişkeni ant.properties dosyası içinde tanımlıyoruz:

```
#junit
junit.testcase.class=test.shop.junit.AllTests
junit.report.dir=${base.dir}/build/junit-report
```

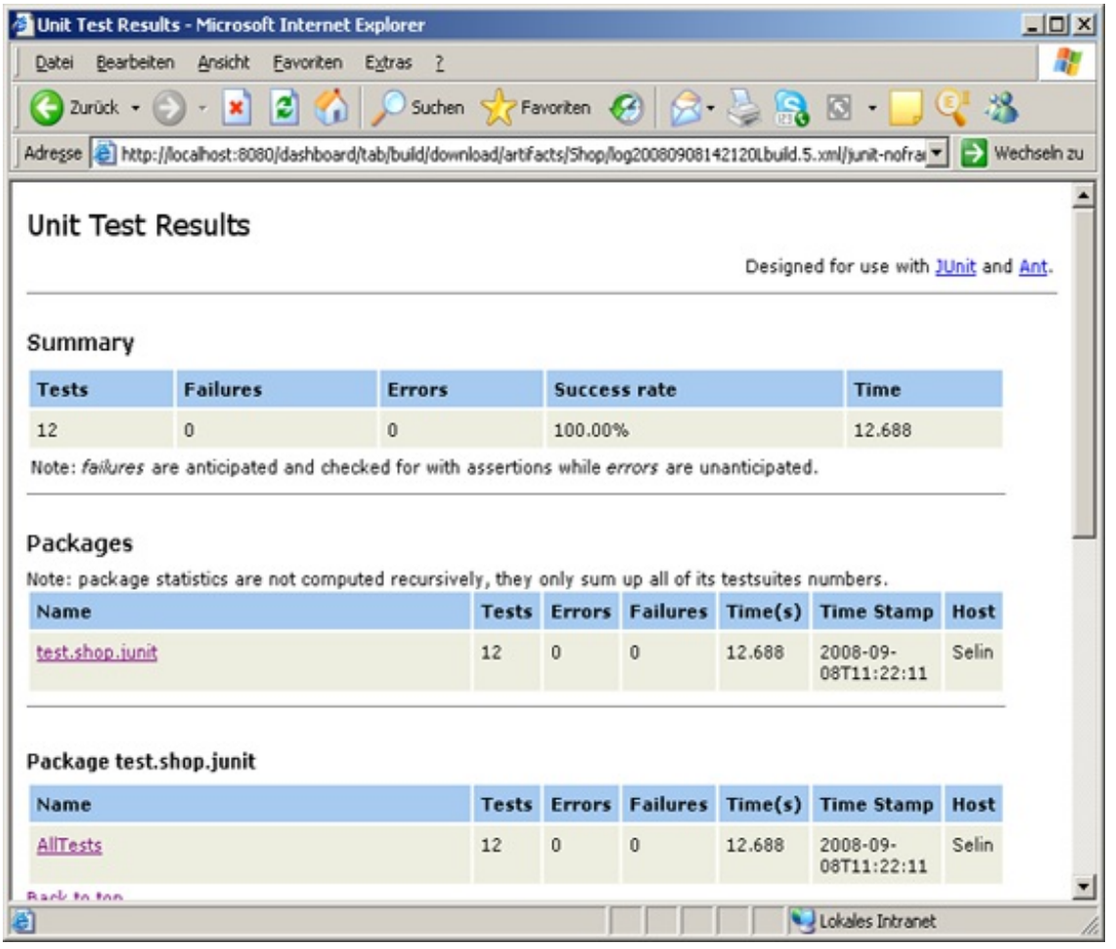
Cruise Control tarafından bu hedefin kullanılmasını istiyorsak, kod 14.2 de görüldüğü gibi

```
<ant antfile="build.xml" target="run-junit"/>
```

build.xml dosyasını ve run-junit hedefini antfile komutu ile tanımlamamız gerekiyor.

Resim 14.11 Cruise Control Artifacts sayfası

Eger run-junit hedefi Cruise Control tarafından testler kırılmadan koşturulabilirse, güncel yapının (build) Artifacts panelinde oluşturulan JUnit raporlarına ulaşabiliriz.



Resim 14.12 JUnit test raporu

Artifaklar projenin Ant ile yapılandırılması esnasında oluşan çıktılardır. Bunlar html, xml ya da başka bir formatta loglar, raporlar ya da başka türlü yapılandırma sonuçlarını ihtiva eden dosyalar olabilir.

Oluşan artifakları Cruise Control Dashboard üzerinden erişilebilir hale getirmek için config.xml (kod 14.1) publishers segmentine aşağıda yer alan artifactspublisher elementini yerleştirmemiz gerekiyor.

```
<artifactspublisher
        dir="checkout/Shop/build/junit-report"
                dest="artifacts/Shop"/>
```

Cruise Control artifactspublisher ile her build ardından Shop/build/junit-report dizininde bulunan dosyaları (JUnit rapor) Artifacts panelinde (resim 14.11) gösterir.

Entegrasyon Testleri ve Sürekli Entegrasyon

Entegrasyon testlerinin koşturulması altyapı bağımlılıklarından dolayı JUnit testlerine nazaran daha zaman alıcıdır. Yapı gereği entegrasyon testleri entegrasyonu test ettikleri için bu testler entegre edilen komponentlerin çalışır durumda olmalarını gerekli kılar. Örneğin veri tabanını test eden bir entegrasyon testi için test öncesi veri tabanının çalışır durumda olması gerekir. Veri tabanı üzerinde işlem yapan test veri tabanı sunucusuna bağlanma, işlem yapma ve test için gerekli fikstürün oluşturulması için birim testlerine nazaran daha zaman fazla zaman harcar.

Cruise Control bünyesinde entegrasyon testlerinin koşturulabilir olması için kullanılan Ant skriptinin entegrasyon için gerekli altyapıyı oluşturabilecek yapıda olması gerekiyor. Onuncu bölümde oluşturduğumuz entegrasyon testleri bir veri tabanına ve Spring konfigürasyonuna ihtiyaç duymaktadır. Bu yüzden test öncesi veri tabanı sunucusunun (HSQLDB) çalışır hale getirilmesi gerekiyor. Entegrasyon testleri için oluşturduğumuz bir Ant hedefin (target) nasıl yapılandırılması gerektiğini şimdi yakından inceleyelim:

Kod 14.6 build.xml - run-integration target

```
<target name="run-integration" depends="compile, hbm2ddl">

    <junit
        fork="false"
        failureproperty="tests.failed"
        showoutput="true"
        printsummary="yes"
        haltonfailure="yes">

        <classpath refid="compile.classpath" />
        <classpath path="${build.dir.classes}" />

        <test
            name="${integration.testcase.class}"
            haltonfailure="yes"
            outfile="build/integration-result">
            <formatter type="xml" />
        </test>
    </junit>

    <fail if="tests.failed">
    ******************************************************
    ******************************************************
    JUnit testlerinde hata olustu. Lütfen kontrol et
    ******************************************************
    ******************************************************
```

```
    </fail>

    <junitreport
        todir="${integration.report.dir}">
        <fileset dir="${build.dir}">
            <include name="integration-result.xml" />
        </fileset>
        <report
            format="noframes"
            todir="${integration.report.dir}" />
    </junitreport>

    <Antcall target="hsqldb-stop"/>

</target>
```

run-integration hedefi yapı olarak run-junit hedefi ile hemen hemen aynı. Sadece kullanılan test sınıfı farklı ve bu run-integration hedefinin bağımlılık listesinde compile hedefi yanı sıra birde hbm2ddl ismini taşıyan ikinci bir hedef daha yer almaktadır.

Gerekli değişkenleri ant.properties dosyasında tanımladıktan sonra, hbm2ddl hedefi nedir, onu inceleyelim.

Kod 14.7 ant.properties

```
#entegrasyon junit
integration.testcase.class=test.shop.integration.AllTests
integration.report.dir=${base.dir}/build/integration-report
```

Onuncu bölümde oluşturduğumuz entegrasyon testleri veri tabanında olması gereken bir veri bazını gerekli kılar. Bu verilerin her test öncesi veri tabananına yerleştirilmesi gerekmektedir. Bu işlem için DBUnit i kullanmıştık. Veri tabanında olması gereken verileri dbunit-dataset.xml dosyasında tanımlayarak, DBUnit aracılığıyla test öncesi veri tabanına yerleştirilmesini sağladık. Entegrasyon testlerinin çalışabilmesi için bu verilerin test öncesi veri tabanında olması gerekiyor, aksi taktirde testler olumsuz sonuç verecektir. Test sınıfları otomatik olarak her test öncesi DBUnit aracılığıyla gerekli verileri veri tabanına yüklerler. Verilerin veri tabanı tablolarına yüklenebilmesi için gerekli şemaların (table structure) daha önce veri tabanında oluşturulmuş olması gerekir. Aksi taktirde gerekli tablolar bulunamadığı için veriler veri tabanına aktarılamaz. Gerekli veri tabanı tablolarını oluşturmak için hbm2ddl hedefini oluşturduk.

Kod 14.8 build.xml - hbm2ddl

```
<taskdef
    name="hibernatetool"
    classname="org.hibernate.tool.Ant.HibernateToolTask"
    classpathref="compile.classpath" />

<target name="hbm2ddl" depends="clean, hsqldb-start">
    <hibernatetool destdir="${build.dir}">
        <classpath>
            <path location="${base.web.web-inf.classes}" />
        </classpath>
        <annotationconfiguration
            configurationfile="${base.dir}/
                        properties/hibernate.cfg.xml" />
        <hbm2ddl
            drop="true"
            create="true"
            export="true"
            outputfilename="shop.sql"
            delimiter=";"
            format="true" />
    </hibernatetool>
</target>
```

Bildiğiniz gibi login bölümünün implementasyonunda Hibernate ORM çatısını kullanmıştık. Üyelik bilgilerini Customer isimli bir sınıfta tuttuk. Kullandığımız anotasyonlar aracılığıyla bu sınıf bir Hibernate managed entity object haline geldi, yani bu sınıfın nesneleri veri tabanının customer isimli tablosunda tutulabilecek özellikte. Customer sınıfının bu özelliğinden faydalanarak customer tablosunun yapısını HibernateToolTask sınıfı yardımıyla otomatik olarak oluşturabiliriz. Bu işlemi gerçekleştirmek için hbm2ddl isminde yeni bir hedef tanımlıyoruz. HibernateToolTask sınıfı hibernate-tools.jar dosyasında bulunmaktadır. Bu sınıfın bulunabilmesi için adı geçen Jar dosyasının lib dizinine eklenmesi gerekiyor.

İlk olarak hibernatetool isminde taskdef elementi ile yeni bir task tanımlıyoruz. Ant hibernatetool isimli taskı tanımadığı için taskdef elementi ile bu yeni taskın tanımlanması gerekiyor.

Akabinde hbm2ddl isminde yeni bir hedef tanımlıyoruz. Bu hedef bünyesinde hibernate.cfg.xml dosyasında bulunan Hibernate ayarları kullanılarak ve derlenen sınıflar içinde Hibernate anotasyonlarını taşıyan sınıflar taranarak gerekli veri tabanı tabloları oluşturulur. Bu işlemin yapılabilmesi için HSQLDB veri tabanı sisteminin çalışır durumda olması gerekir. Bu sebeple depends

elementinde clean ve hsqldb-start hedefleri yer almaktadır. Buna göre hbm2ddl hedefi koşmadan önce clean, akabinde hsqldb-start hedefleri devreye girer. Böylece HSQLDB çalışır hale getirilir ve hbm2ddl ile gerekli tablo yapısı oluşturulur.

```
Kod 14.9  shop.sql - oluşturulan ddl skript
drop table customer if exists;

create table customer (
    id bigint generated by default as identity (start with 1),
    email varchar(255) not null,
    password varchar(255) not null,
    primary key (id)
)
```

Tekrar run-integration hedefine göz attığımızda, veri tabanı tablolarının hbm2ddl hedefi aracılığıyla her test öncesi yeniden oluşturulduğunu görmekteyiz. Entegrasyon testleri ihtiyaç duydukları verileri DBUnit ile test öncesi veri tabanına aktarırlar. Test sona erdikten sonra hsqldb-stop hedefi ile HSQLDB veri tabanını tekrar deaktive ediyoruz.

Bu kadar harika bir otomasyonu siz hangi yöntemlerle sağlayabilirdiniz? Çevik yöntemlerle tanışmadan önce böyle bir otomasyonun mümkün olabileceğinin bile aklımdan geçmesi mümkün değildi. Bu yeni yöntemlerle çalışma imkanı bulduğum için kendimi mutlu sayıyorum.

Onay/Kabul Testleri ve Sürekli Entegrasyon

Entegrasyon testleri gibi onay/kabul testlerinde altyapı bağımlılıkları vardır. Onay/kabul testleri çalışan bir veri tabanı yanı sıra, Tomcat gibi uygulamanın deploy edildiği bir uygulama sunucusuna ihtiyaç duyarlar. Onuncu bölümde oluşturduğumuz onay/kabul testleri için bir sonraki listede yer alan altyapı komponentlerinin çalışır durumda olması gerekir:

- Tomcat uygulama sunucusu
- HSQLDB veri tabanı
- Selenium Remote Control sunucusu

Oluşturmamız gereken Ant hedefi test öncesi Tomcat, HSQLDB ve Selenium sunucusunu çalıştırmak ve test bitiminde bu komponentleri durdurabilecek özelliklere sahip olmak durumunda. Bunu nasıl yapabileceğimizi inceleyelim:

Kod 14.10 build.xml - run-acceptance hedefi

```
<target name="run-acceptance" depends="compile, dbunit">

    <parallel>
        <Antcall target="start-selenium-server">
        </Antcall>
        <sequential>
            <echo
                taskname="waitfor"
                message="Wait for proxy server launch" />
            <waitfor
                maxwait="2"
                maxwaitunit="minute"
                checkevery="100">
                <http
                    url="http://localhost:4444/selenium-server
                                /driver/?cmd=testComplete" />
            </waitfor>
        </sequential>
    </parallel>

     <Antcall target="tomcat-start"/>

    <junit fork="false"
        failureproperty="tests.failed"
        showoutput="true"
        printsummary="yes"
        haltonfailure="yes">

        <classpath refid="compile.classpath" />
        <classpath path="${build.dir.classes}" />

        <test
            name="${acceptance.testcase.class}"
            haltonfailure="yes"
            outfile="build/acceptance-result">
            <formatter type="xml" />
        </test>
    </junit>


    <fail if="tests.failed">
    ******************************************************
    ******************************************************
    JUnit testlerinde hata olustu. Lütfen kontrol et
    ******************************************************
    ******************************************************
```

```
    </fail>

    <junitreport
        todir="${acceptance.report.dir}">
        <fileset dir="${build.dir}">
            <include name="acceptance-result.xml" />
        </fileset>
        <report
            format="noframes"
            todir="${acceptance.report.dir}" />
    </junitreport>

    <Antcall target="hsqldb-stop"/>
    <Antcall target="tomcat-stop"/>
    <Antcall target="stop-selenium-server"/>

</target>
```

Onay/kabul testlerini koşturmak için run-acceptance isminde yeni bir hedef tanımlıyoruz. Bu hedef bünyesinde kullandığımız diğer hedefler şöyledir:

- dbunit
- tomcat-start
- tomcat-stop
- start-selenium-server
- stop-selenium-server

Bu hedeflerin görevlerinin ne olduğunu yakından inceleyelim:

```
Kod 14.11  build.xml - dbunit hedefi

<taskdef
        name="dbunit"
        classname="org.dbunit.Ant.DbUnitTask"
        classpath="${base.lib}/dbunit-2.2.jar"/>


<target name="dbunit" depends="hbm2ddl">
    <dbunit
        datatypefactory="org.dbunit.ext.hsqldb.HsqldbDataTypeFactory"
        driver="org.hsqldb.jdbcDriver"
        url="jdbc:hsqldb:hsql://localhost:9006/shop"
        userid="sa"
        password="">

        <classpath>
            <pathelement
```

```
                location="${base.lib}/hsqldb.jar" />
        </classpath>
        <operation
            type="INSERT"
            src="properties/dbunit-dataset.xml" />
    </dbunit>

</target>
```

Onay/kabul testlerinin entegrasyon testlerinde olduğu gibi veri tabanında bir veri bazına ihtiyaçları bulunmaktadır. Bu verilerin test öncesi DBUnit ile veri tabanına aktarılması gerekmektedir. Entegrasyon testleri bu işlemi otomatik olarak setUp() metodunda gerçekleştirmişti. Lakin onay/kabul testlerinin böyle bir özelliği bulunmamaktadır. Bu yüzden onay/kabul testlerini koşturmadan önce dbunit hedefi ile dbunit-dataset.xml içinde tanımlanmış olan verileri veri tabanına aktarıyoruz.

dbunit hedefi daha önce tanıştığımız hbm2ddl hedefine bağımlıdır. hbm2dll hedefi bünyesinde HSQLDB sunucusu aktif edilir ve veri tabanı şeması oluşturulur.

Kod 14.12 build.xml – tomcat-start, tomcat-stop hedefleri

```
<target name="tomcat-stop">
    <java
        jar="${tomcat.home}/bin/bootstrap.jar"
        fork="true"
        spawn="true">
        <jvmarg
            value="-Dcatalina.home=${tomcat.home}" />
        <arg
            line="stop" />
    </java>
</target>
```

Tomcat sunucusu start, stop edebilmek için kod 14.11 de yer alan hedefleri tanımlıyoruz.

```
Kod 14.13  build.xml

<target name="start-selenium-server">
    <java
        jar="lib/selenium-server.jar"
        fork="true"
        spawn="true">
```

```
        <arg line="-timeout 30" />
    </java>
</target>



<target name="stop-selenium-server">
    <get
        taskname="selenium-shutdown"
                src="http://localhost:4444/
                selenium-server/driver/?cmd=shutDown"
        dest="result.txt"
        ignoreerrors="true" />
    <echo
        taskname="selenium-shutdown"
        message="DGF Errors during shutdown are expected" />
</target>
```

Onay/kabul testlerini Selenium API si ile geliştirdiğimiz için Selenium Remote Control sunucusunun çalışır durumda olması gerekmektedir. Selenium RC sunucusunun start ve stop işlemleri için kod 14.13 de tanımladığımız hedefleri kullanıyoruz.

```
Kod 14.14  Ant run-acceptance console çıktısı

Buildfile: C:\workspace\Shop\build.xml
clean:
    [mkdir] Created dir: C:\workspace\Shop\build
    [mkdir] Created dir: C:\workspace\Shop\build\classes
    [mkdir] Created dir: C:\workspace\Shop\build\junit-report
    [mkdir] Created dir: C:\workspace\Shop\build\integration-report
    [mkdir] Created dir: C:\workspace\Shop\build\acceptance-report
copy_properties:
     [copy] Copying 5 files to C:\workspace\Shop\build\classes
compile:
    [javac] Compiling 10 source files to C:\workspace\Shop\build\
                                       classes
    [javac] Note: C:\workspace\Shop\src\shop\presentation\login\
                    controller\LoginController.java uses
                        unchecked or unsafe operations.
    [javac] Note: Recompile with -Xlint:unchecked for details.
    [javac] Compiling 11 source files to C:\workspace\Shop\build\classes
hsqldb-start:
     [java] The args attribute is deprecated. Please use nested
                        arg elements.
hbm2ddl:
[hibernatetool]     drop table customer if exists;
```

```
[hibernatetool] 18:19:00,343 DEBUG SchemaExport:
[hibernatetool]     drop table customer if exists;
[hibernatetool]
[hibernatetool]     create table customer (
[hibernatetool]         id bigint generated by default as
                                identity (start with 1),
[hibernatetool]         email varchar(255) not null,
[hibernatetool]         password varchar(255) not null,
[hibernatetool]         primary key (id)
[hibernatetool]     );
[hibernatetool] 18:19:00,343 DEBUG SchemaExport:
[hibernatetool]     create table customer (
[hibernatetool]         id bigint generated by default as identity
                    (start with 1),
[hibernatetool]         email varchar(255) not null,
[hibernatetool]         password varchar(255) not null,
[hibernatetool]         primary key (id)
[hibernatetool]     );
[hibernatetool] 18:19:00,359  INFO SchemaExport: schema export
                        complete
[hibernatetool] 18:19:00,359 DEBUG DriverManagerConnectionProvider:
                        returningconnection to pool, pool size: 1
[hibernatetool] 18:19:00,359  INFO DriverManagerConnectionProvider:
                        cleaning up
                        connection pool:
                        jdbc:hsqldb:hsql://localhost:9006/shop
dbunit:
   [dbunit] Executing operation: INSERT
   [dbunit]           on   file: C:\workspace\Shop\properties\
                                dbunit-dataset.xml
   [dbunit]           with format: null
run-acceptance:
  [waitfor] Wait for proxy server launch

start-selenium-server:

tomcat-stop:

tomcat-start:
    [junit] Running test.shop.acceptance.AllTests
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed:
                10,406 sec
[junitreport] Processing C:\workspace\Shop\build\acceptance-report
            \TESTS-TestSuites.xml to C:\workspace\Shop\build\
                    acceptance-report\
            junit-noframes.html
[junitreport] Loading stylesheet jar:file:/C:/_development/
                    eclipse_3_3_1/eclipse/plugins
                /org.apache.Ant_1.7.0.v200706080842/lib/Ant-junit.jar!
```

```
                            /org/apache/tools
                 /Ant/taskdefs/optional/junit/xsl/junit-noframes.xsl
18:19:14,609  INFO DriverManagerConnectionProvider:
                         cleaning up connection pool:
                 jdbc:hsqldb:hsql://localhost:9006/shop
[junitreport] Transform time: 547ms

hsqldb-stop:

tomcat-stop:
stop-selenium-server:

[selenium-shutdown] Getting: http://localhost:4444/selenium-server/
                    driver/?cmd=shutDown
[selenium-shutdown] To: C:\workspace\Shop\result.txt
[selenium-shutdown] DGF Errors during shutdown are expected
BUILD SUCCESSFUL
Total time: 24 seconds
```

run-acceptance hedefini koşturduğumuz zaman kod 14.14 de yer alan ekran çıktısı oluşacaktır. Bu onay/kabul testlerinin tamamen otomatize edilmiş bir şekilde çalıştığı anlamına gelmektedir ki tam test otomasyonu projelerde takip edilmesi gereken ana amaçlardan birisi olmalıdır.

Oluşturduğumuz tüm birim, entegrasyon ve onay/kabul testlerini bir test suite bünyesinde toplamak istersek, kod 14.14 de ki gibi bir test sınıfı oluşturabiliriz.

```
Kod 14.15  AllTests.java - test suite of test suites

package test.shop;

@RunWith(Suite.class)
@SuiteClasses({test.shop.junit.AllTests.class,
        test.shop.integration.AllTests.class,
         test.shop.acceptance.AllTests.class })
public class AllTests{
}

Kod 14.16  build.xml - AllTests hedefi

<target
    name="run-alltets"
    depends="run-junit, run-integration, run-acceptance"/>
```

Yeni bir hedef (kod 14.16) oluşturarak, sürekli entegrasyon sürecinde mevcut olan tüm testlerin koşturulmasını sağlıyabiliriz. Yeni hedefi (run-alltests) Cruise

Control a tanıtmak için build-Shop.xml dosyasında değişiklik yapmamız gerekiyor:

Kod 14.17 build-Shop.xml (Cruise Control)

```
<project name="build-Shop" default="build" basedir="checkout/Shop">
    <target name="build">

       <exec executable="svn">
           <arg line="up"/>
        </exec>

        <ant antfile="build.xml" target="run-alltests"/>

     </target>
</project>
```

15. Bölüm

Yazılım Metrikleri

Giriş

Bir yazılım metriği yazılım sisteminin herhangi bir niteliğinin (property) değerlendirilmesi için kullanılan ölçektir. Yazılımcı olarak elimizden gelenin en iyisini yapmayı isteriz. Peki ortaya koyduklarımızın iyi olduğunu nasıl ölçebiliriz? Yaptıklarımızın iyi olduğunu düşünmemiz sübjektif bir yaklaşımdır. Onların iyi olduğunu ispatlamamız gerekiyor. Yazılım metrikleri yapılan işin kalitesinin ölçümünde bize yol gösterirler.

Projelerde kullanılan genel metrikler şöyledir:

- Kod satır adedi (source lines of code)
- Kod kompleksite oranı (cyclomatic complexity)
- Satır başına düşen hata adedi (bugs per line of code)
- Kullanılan kod oranı (code coverage)
- Sınıf ve interface sınıfların adedi (number of classes and interfaces)
- Paket metrikleri (software package metrics)
- Uyumluluk (cohesion)
- Bağımlılık (coupling)

Kullanılan bu metrikler yazılım sistemin karakteristik özelliklerini ortaya koyarlar. Örneğin bir yazılım sisteminin bakılabilirliği (maintainability) kod kompleksitesiyle (cyclomatic complexity) direk orantılıdır. Kod kompleksitesi yüksek olan bir yazılım sisteminin bakımı ve geliştirilmesi o oranda zorlaşır.

Değişik türdeki metrikleri tespit edebilmek için çevik projelerde çoğunluğu açık kaynaklı (open source) olan araçlardan faydalanılır. Kitabın bu bölümünde yakından inceleyeceğimiz bu araçlar şöyledir:

- CheckStyle
- FindBugs
- JDepend
- PMD
- Emma

CheckStyle

CheckStyle öncelikli olarak uygulanan kod standartlarını kontrol etmek için geliştirilen bir araçtır. Zaman içinde yapılan eklemelerle CheckStyle daha kapsamlı yazılım metriklerin elde edilmesi için kullanılan bir araç olmuştur.

Bir XML dosyasında kullanılmak istenen CheckStyle modülleri tanımlanır. Bunun bir örneğini kod 15.1 de görmekteyiz.

```
Kod 15.1   checkstyle.xml

<module name="Checker">
    <module name="PackageHtml"/>
    <module name="TreeWalker">
        <module name="AvoidStarImport"/>
        <module name="ConstantName"/>
        <module name="EmptyBlock"/>
    </module>
</module>
```

Örneğin kod kompleksitesini ölçmek için checkstyle.xml dosyasında

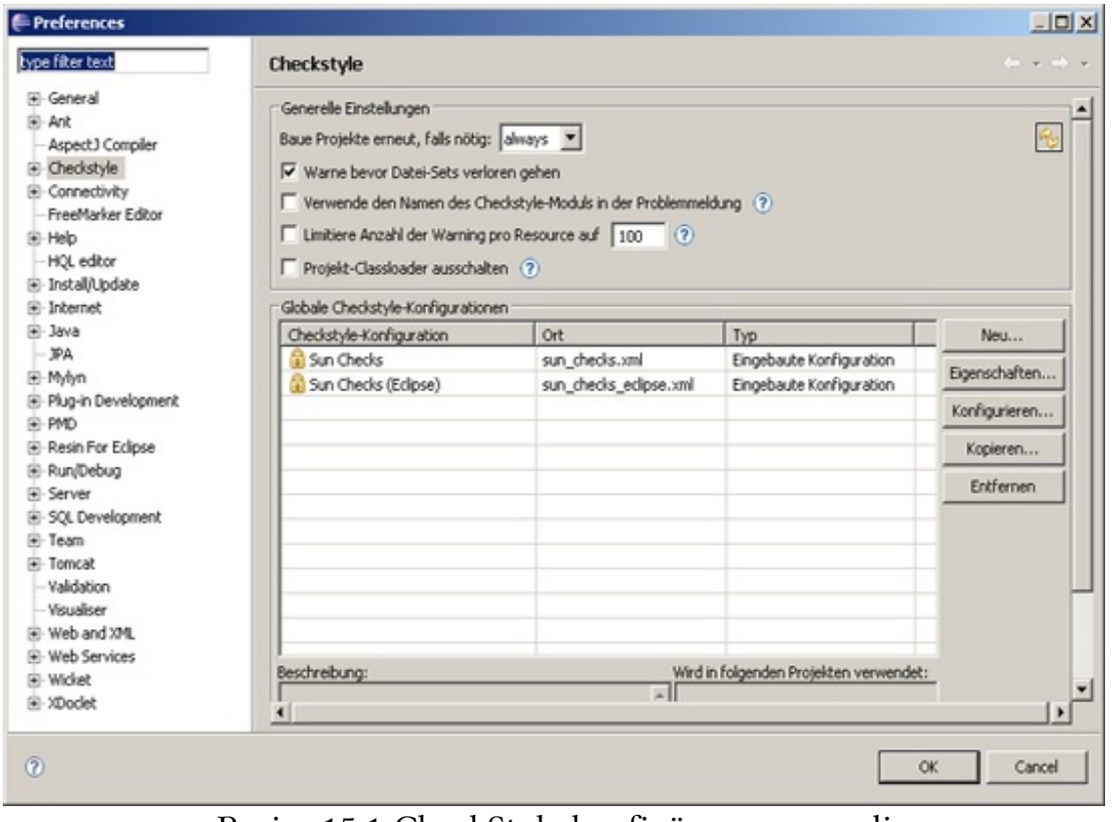
```
<module name="CyclomaticComplexity"/>
```

tanımlaması yapılır. Modül listesini CheckStyle websitesinden edinebilirsiniz.

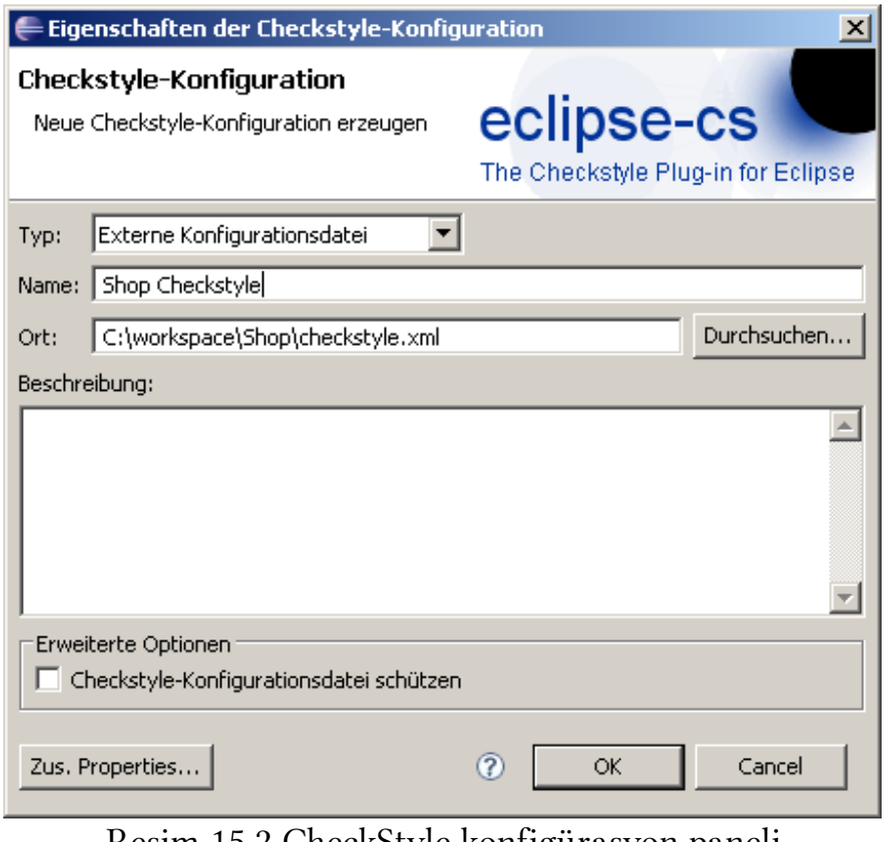
CheckStyle Eclipse Plugin

CheckStyle plugin olarak Eclipse e entegre edilebilir. Bu plugini http://eclipse cs.sourceforge.net adresinden edinebilirsiniz.

Plugin kurulumu yapıldıktan sonra proje için tanımlanan ve CheckStyle modüllerini ihtiva eden XML konfigürasyon dosyasının (checkstyle.xml) plugine tanıtılması gerekmektedir.



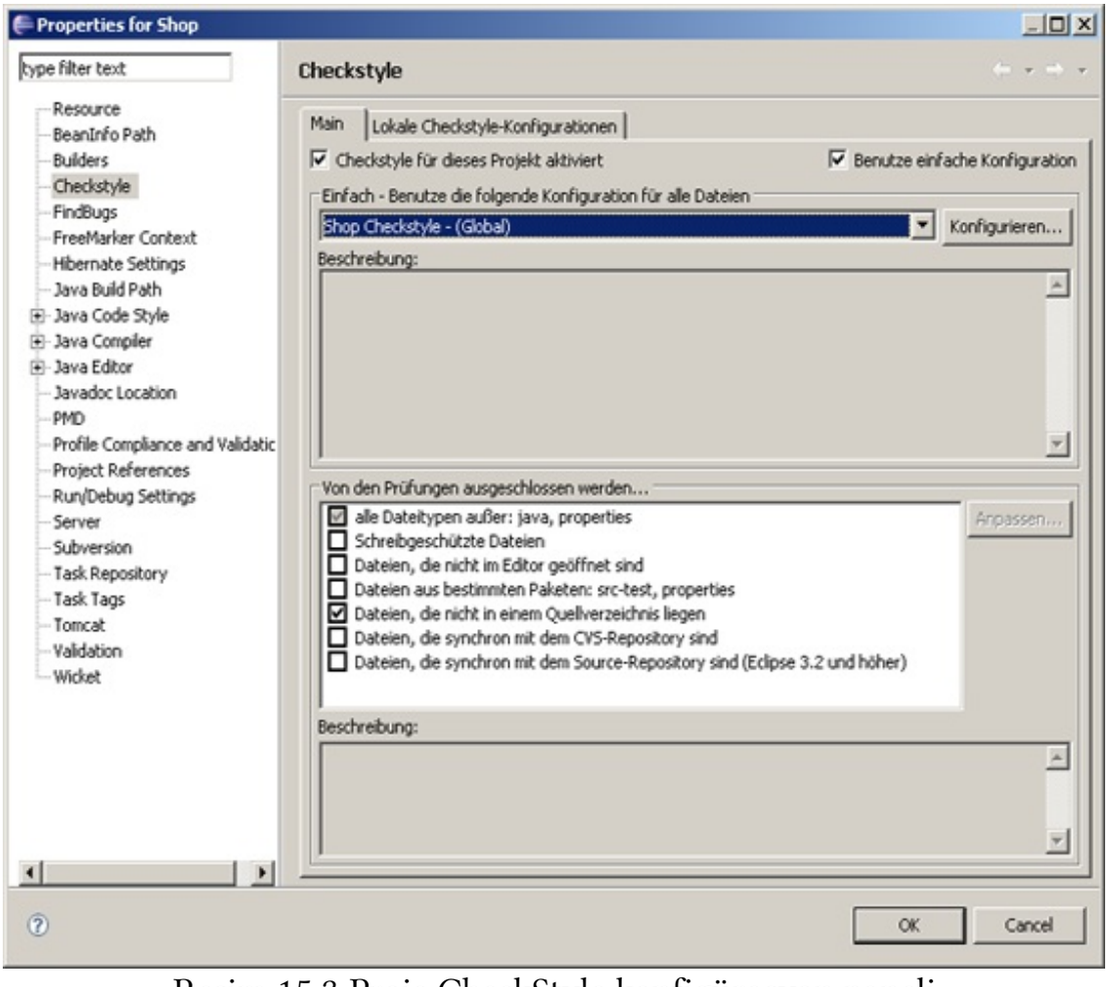
Resim 15.1 CheckStyle konfigürasyon paneli



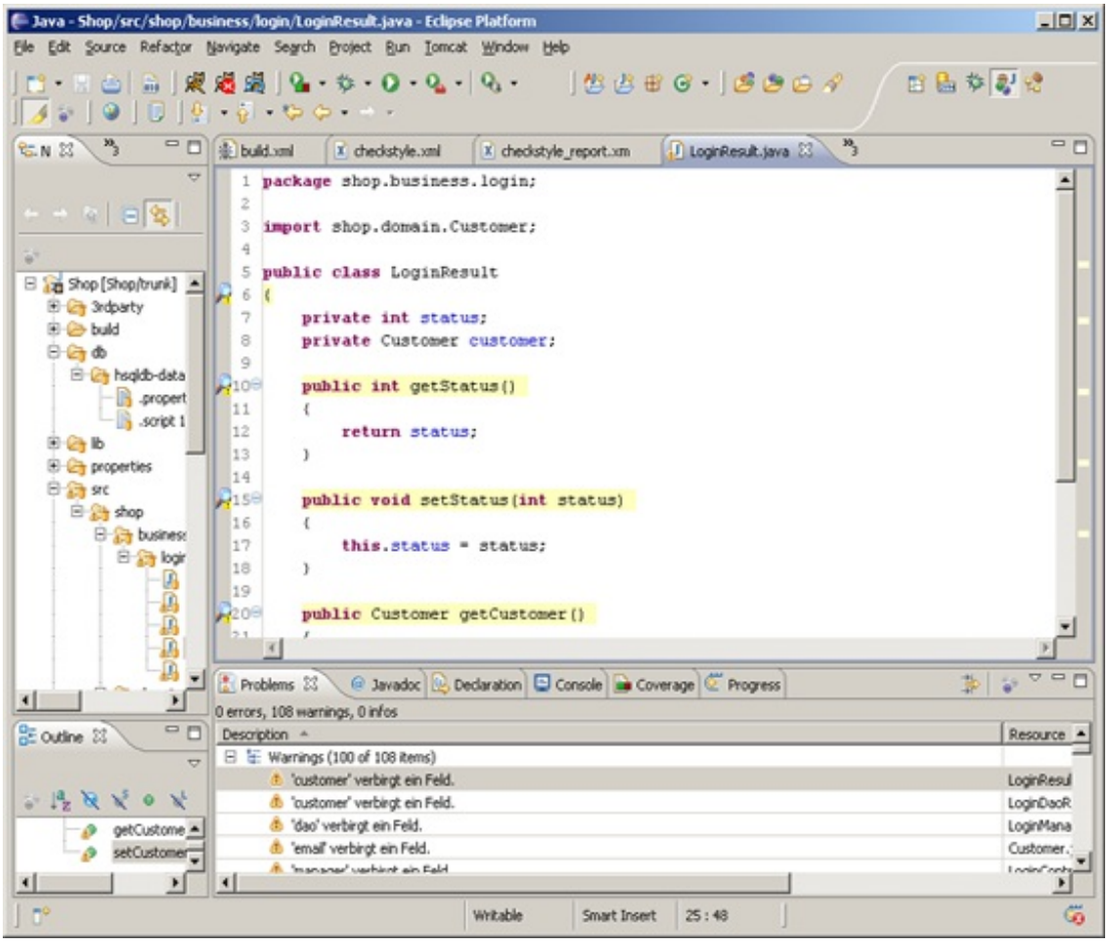
Resim 15.2 CheckStyle konfigürasyon paneli

Resim 15.2 de görüldüğü gibi proje için kullanmak istediğimiz checkstyle.xml dosyasını seçerek, yeni bir CheckStyle konfigürasyonu oluşturuyoruz.. Bir

sonraki adımda tanımladığımız bu konfigürasyon ile CheckStyle ı proje bünyesinde aktive etmemiz gerekiyor.



Resim 15.3 Proje CheckStyle konfigürasyon paneli



Resim 15.4 Proje için aktif hale getirilen CheckStyle problems panelinde tespit ettiği hataları gösterir

CheckStyle ve Sürekli Entegrasyon

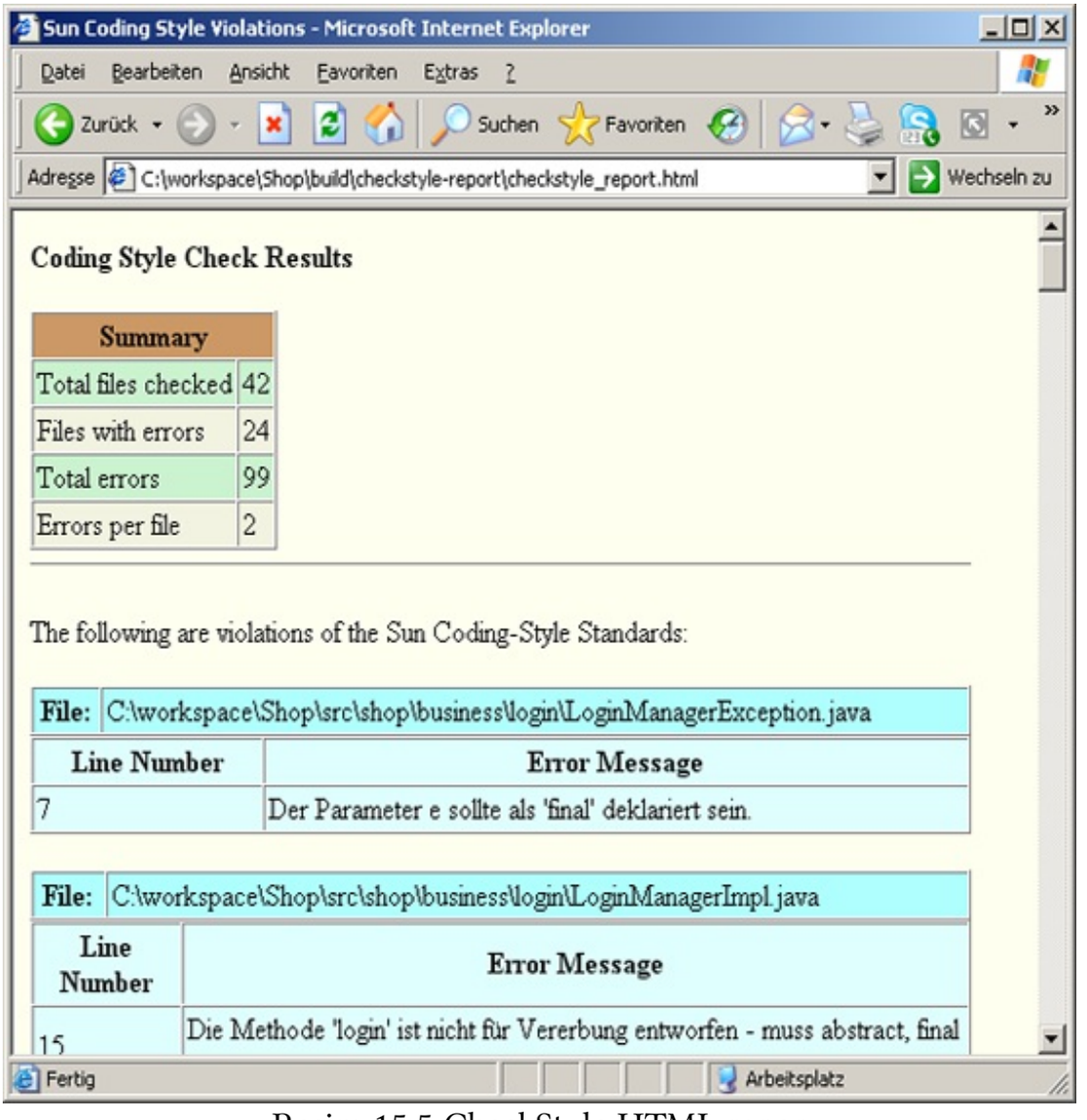
Çevik süreçte kullandığımız sürekli entegrasyon sayesinde sistemin değişik bölümlerinin her değişikliğin ardından entegre edilmesini sağlayabiliriz. Bu birim testlerine bağımlı olan bir süreçtir. Entegrasyon sonunda entegrasyon sonuçlarını ihtiva eden artifaklar oluşur, örneğin JUnit raporları. Bu artifakları inceleyerek entegrasyonun kalitesi hakkında fikir sahibi olabiliriz.

Sürekli entegrasyon proje açısından önemli bir süreç ise, CheckStyle gibi kod standartlarını ve diğer metrikleri kontrol edici bir mekanizmanın bu sürece dahil edilmesinde fayda vardır. Bunu sağlayabilmek için CheckStyle ı Ant build.xml skriptimiz ile kontrol edebilmemiz gerekmektedir.

```
Kod 15.2   Ant build.xml

<taskdef
    resource="checkstyletask.properties"
```

```
    classpath="${checkstyle.home}/checkstyle-all-4.3.jar" />


<target
    name="checkstyle" depends="clean">

    <checkstyle
        config="checkstyle.xml"
        failureProperty="checkstyle.failure"
        failOnViolation="false">
            <formatter type="xml"
            tofile="${checkstyle.report.dir}/checkstyle_report.xml" />
        <fileset
            dir="${base.src}"
            includes="**/*.java" />
        <fileset
            dir="${base.src.test}"
            includes="**/*.java" />
    </checkstyle>

    <xslt
        in="${checkstyle.report.dir}/checkstyle_report.xml"
        out="${checkstyle.report.dir}/checkstyle_report.html"
        style="${checkstyle.home}/contrib/checkstyle-simple.xsl" />
</target>
```

Kod 15.2 de CheckStyle Ant entegrasyonu için gerekli taskdef ve target yer almaktadır. CheckStyle tespit ettiği anomalik durumları checkstyle_report.xml dosyasına kaydeder. Daha sonra xslt taskı ile bu XML dosyasını kullanarak HTML tabanlı bir rapor üretebiliriz. Bunun bir örneğini bir sonraki resimde görmekteyiz.



Resim 15.5 CheckStyle HTML rapor

JDepend

JDepend ile Java sınıfları baz alınarak tasarım kalite metrikleri oluşturulur. Bunlar:

- Sınıf ve interface sınıfların adedi
- Afferent Couplings (Ca): Paket içinde bulunan sınıflara bağımlı olan diğer paketlerin adedi
- Efferent Couplings (Ce): Paket içinde bulunan sınıfların bağımlı olduğu diğer paketlerin adedi
- Abstractness (A): Bir paket içindeki soyut sınıfların paket içindeki tüm sınıflara olan oranı
- Instability (I): Ca metriğinin Ca+Ce toplamına olan oranı (I = Ce / (Ce+Ca)

)

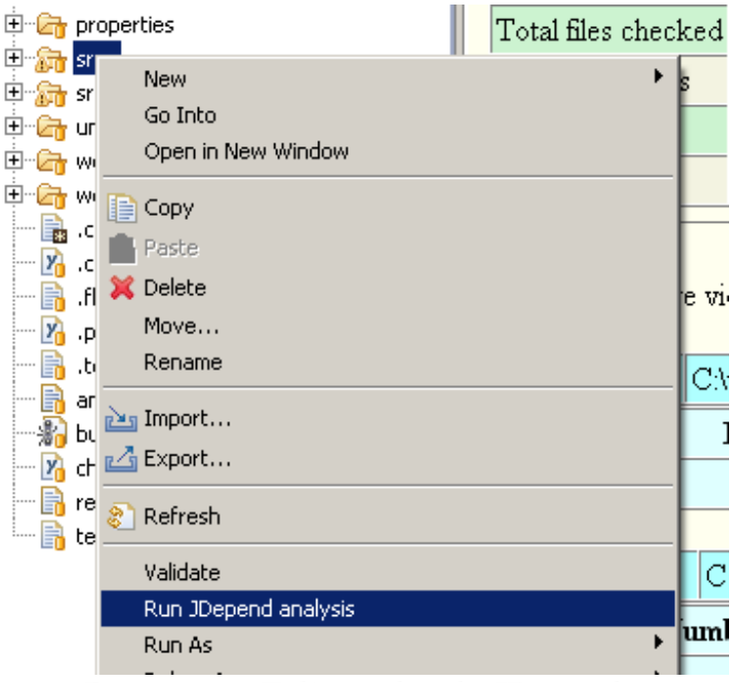
- Distance from the Main Sequence (D): Main Sequence e olan uzaklık

Bu metrikleri kitabın yedinci bölümünde SDP (Stable Dependencies Principle) ve SAP (Stable Abstractions Principle) paket (package) tasarım prensipleri olarak tanımıştık. Bu yüzden bu bölümde detaya girmeden JDepend in nasıl kullanılabileceğini aktarmak istiyorum. Ama JDepend tarafında oluşturulan metriklerin iyi anlaşılabilmesi için yedinci bölümde ele aldığımız tasarım prensiplerinin iyi anlaşılması gerekmektedir.

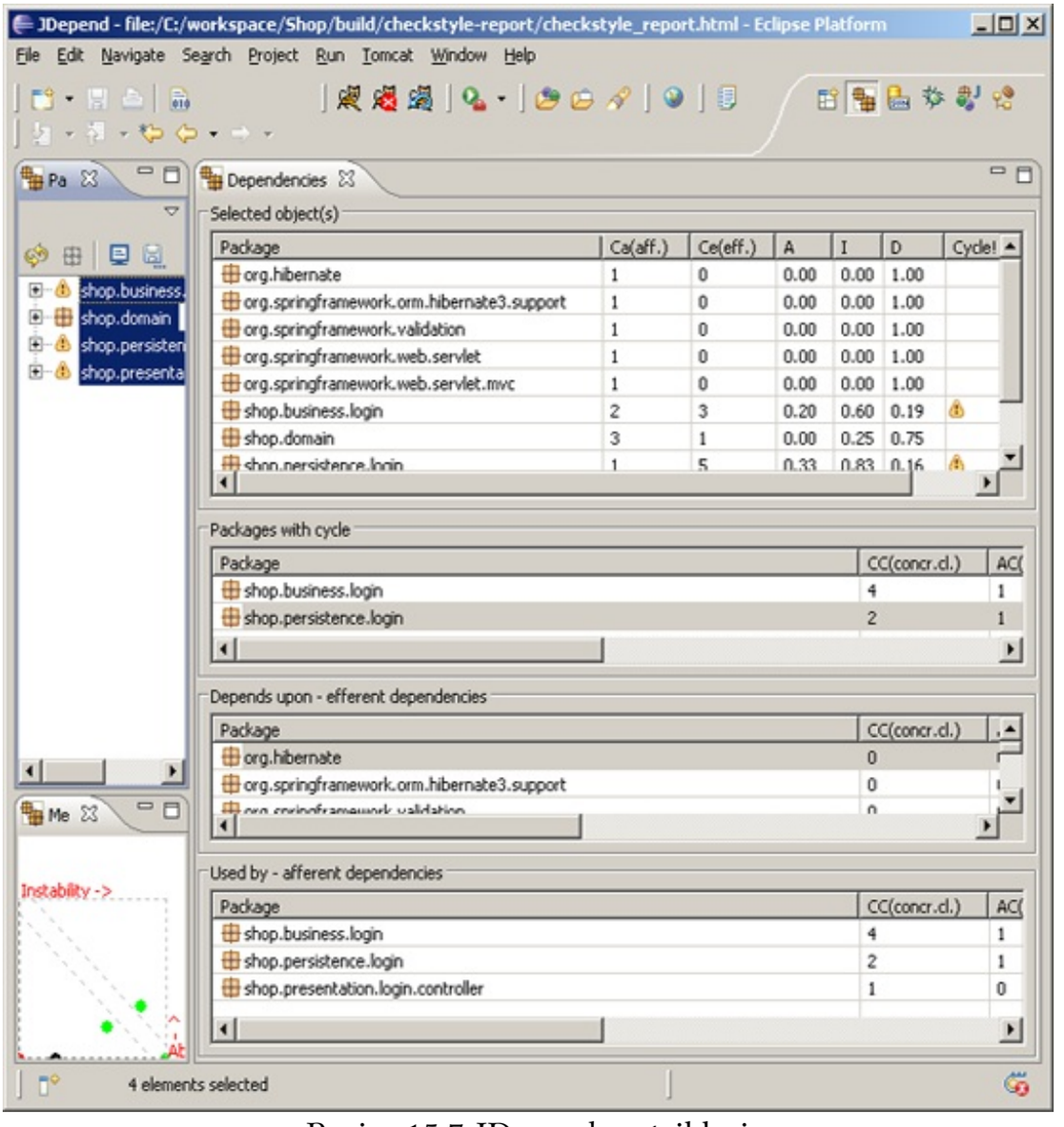
JDepend Eclipse Plugin

JDepend plugin olarak Eclipse e entegre edilebilir. Bu plugini http://andrei.gmxhome.de/jdepend4eclipse adresinden edinebilirsiniz.

Eclipse altında ismi geçen metrikleri oluşturmak için kodların yer aldığı bir dizin (src) seçerek, sağ tuşa tıklıyoruz. Oluşan menüde Run JDepend analysis isminde bir alt menü yer almaktadır.



Resim 15.6 Eclipse altında JDepend



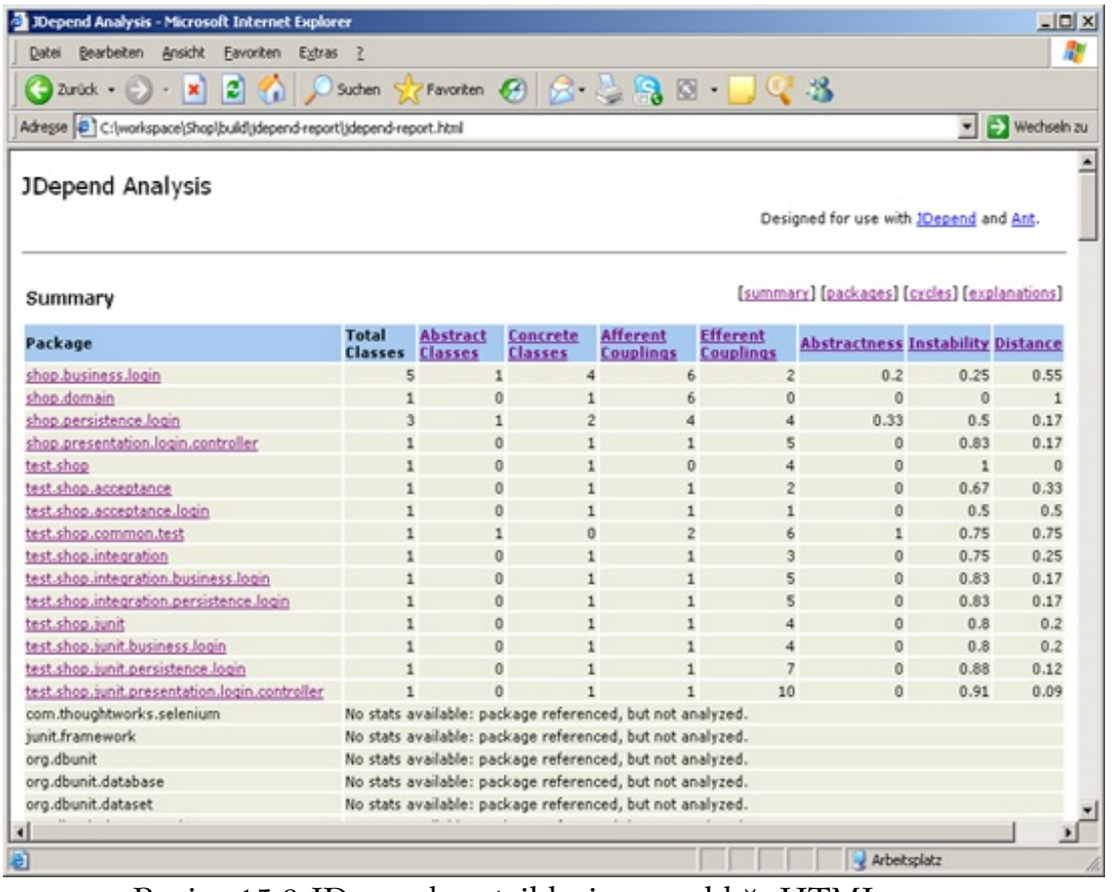
Resim 15.7 JDepend metrikleri

JDepend ve Sürekli Entegrasyon

JDepend aracını sürekli entegrasyona entegre etmek için kod 15.3 de kullanılan Ant hedefi kullanılabilir.

```
Kod 15.3   Ant build.xml

<taskdef
    name="jdepend"
        classname="org.apache.tools.ant.taskdefs.optional.
                    jdepend.JDependTask"
    classpath="${jdepend.home}/lib/jdepend-2.9.jar" />

<target name="jdepend" depends="clean, compile">
```

```
    <jdepend
        format="xml"

        outputfile="${jdepend.report.dir}/jdepend-report.xml">
        <exclude name="java.*" />
        <exclude name="javax.*" />
        <classespath>
            <pathelement location="${build.dir.classes}" />
        </classespath>
        <classpath location="compile.classpath" />
    </jdepend>
    <xslt
        in="${jdepend.report.dir}/jdepend-report.xml"
        out="${jdepend.report.dir}/jdepend-report.html"
        style="${ant.home}/etc/jdepend.xsl" />
</target>
```



Resim 15.8 JDepend metriklerin yer aldığı HTML rapor

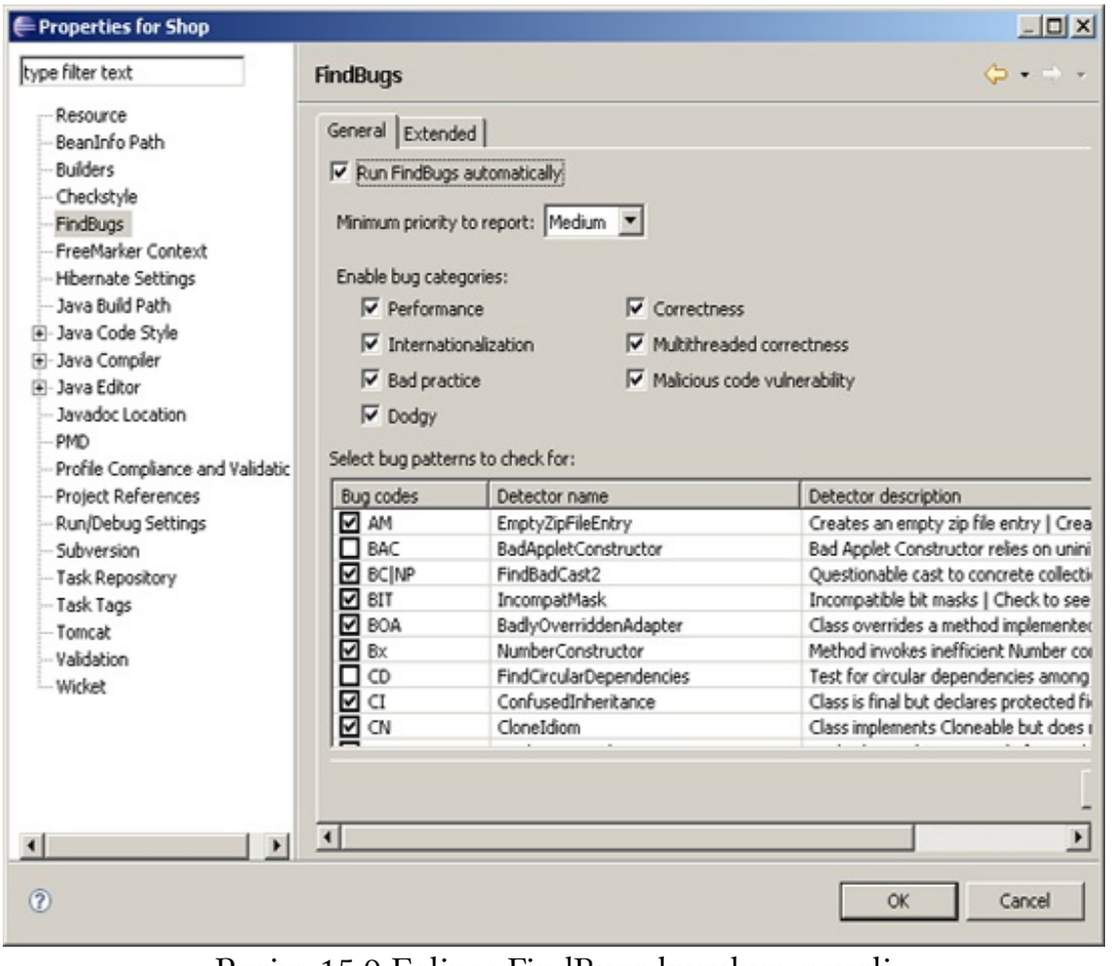
FindBugs

FindBugs mevcut kodu analize ederek, hataları tespit etmek için kullanılan bir araçtır.

FindBugs Eclipse Plugin

FindBugs plugin olarak Eclipse e entegre edilebilir. Bu plugini http://findbugs.sourceforge.net adresinden edinebilirsiniz.

Plugin kurulumu yapıldıktan sonra proje için aktive edilmesi gerekmektedir (resim 15.9).



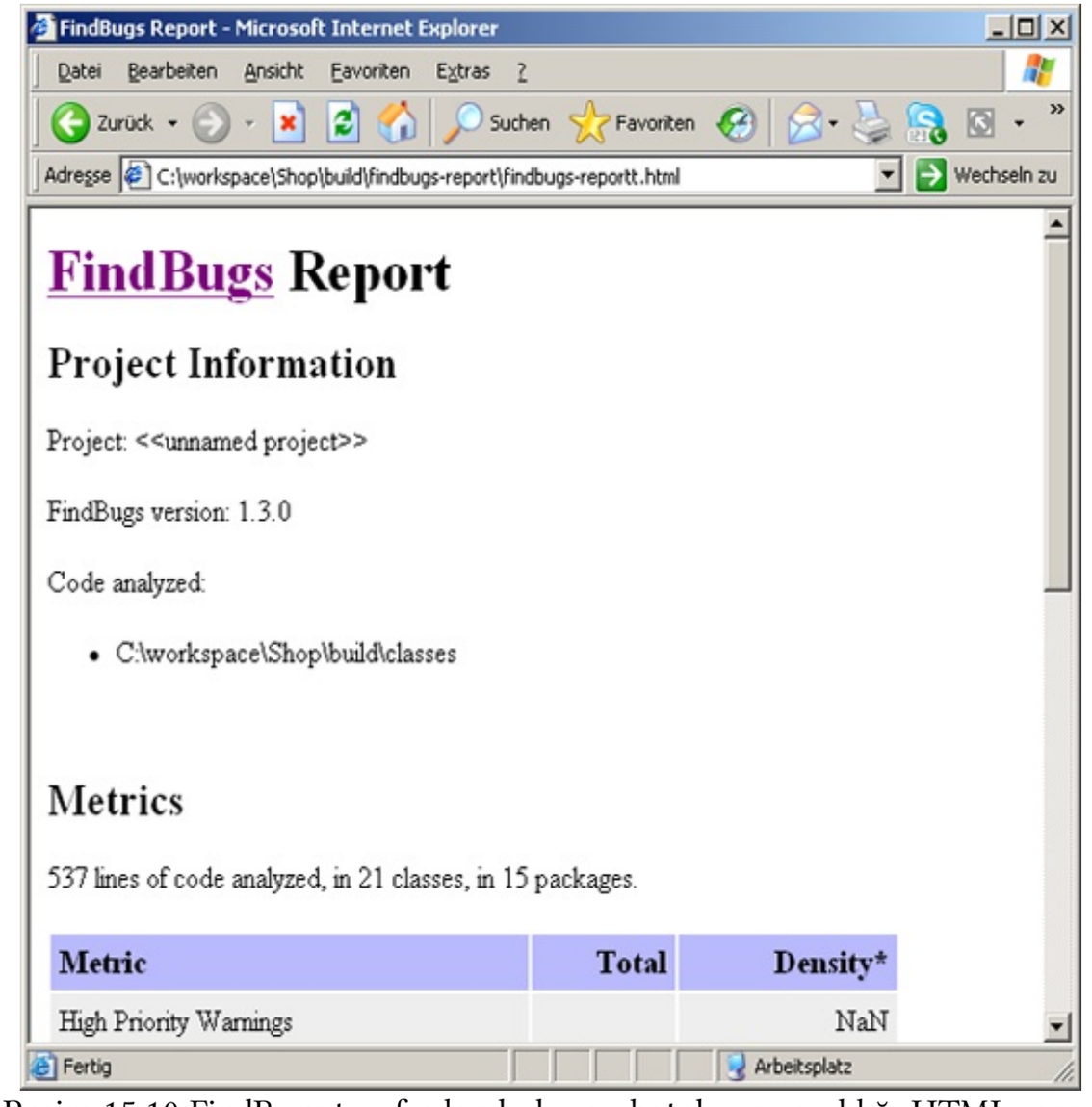
Resim 15.9 Eclipse FindBugs kurulum paneli

FindBugs ve Sürekli Entegrasyon

FindBugs aracını sürekli entegrasyona entegre etmek için kod 15.4 de kullanılan Ant hedefi kullanılabilir.

```
Kod 15.4   Ant build.xml

<taskdef
    name="findbugs"
    classname="edu.umd.cs.findbugs.anttask.FindBugsTask"
```

```
    classpath="${findbug.home}/lib/findbugs-ant.jar" />

<target name="findbugs" depends="clean, compile">
    <findbugs
        home="${findbug.home}"
        output="xml" outputFile="${findbug.report.dir}/findbugs.xml"
        jvmargs="-Xmx512m">


        <auxClasspath>
            <fileset dir="${base.web}/WEB-INF/lib">
                <include name="*.jar" />
            </fileset>
            <fileset dir="${base.dir}/lib">
                <include name="*.jar" />
            </fileset>
        </auxClasspath>

        <sourcePath>
            <fileset dir="${base.src}">
                <include name="**/*.java" />
            </fileset>
        </sourcePath>
        <class location="${build.dir.classes}" />
    </findbugs>

    <xslt
        in="${findbug.report.dir}/findbugs.xml"
        style="${findbug.home}/src/xsl/default.xsl"
        out="${findbug.report.dir}/findbugs-reportt.html" />
</target>
```



Resim 15.10 FindBugs tarafından bulunan hataların yeraldığı HTML rapor

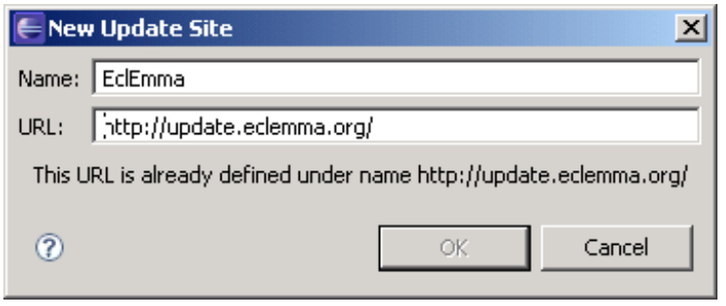
Emma

Emma JUnit testleri tarafından işlem gören kod bölümlerini tespit etmek için kullanılan bir araçtır. Diğer bir adı code coverage olan bu işlem ile testlerin kapsama alanı tespit edilir. Testler aracılığıyla hangi kod satırlarının işlem gördüğünü görebilmek, gereksiz ve ölü kod satırlarının lokalize edilebilmesi için önemli bir işlemdir.

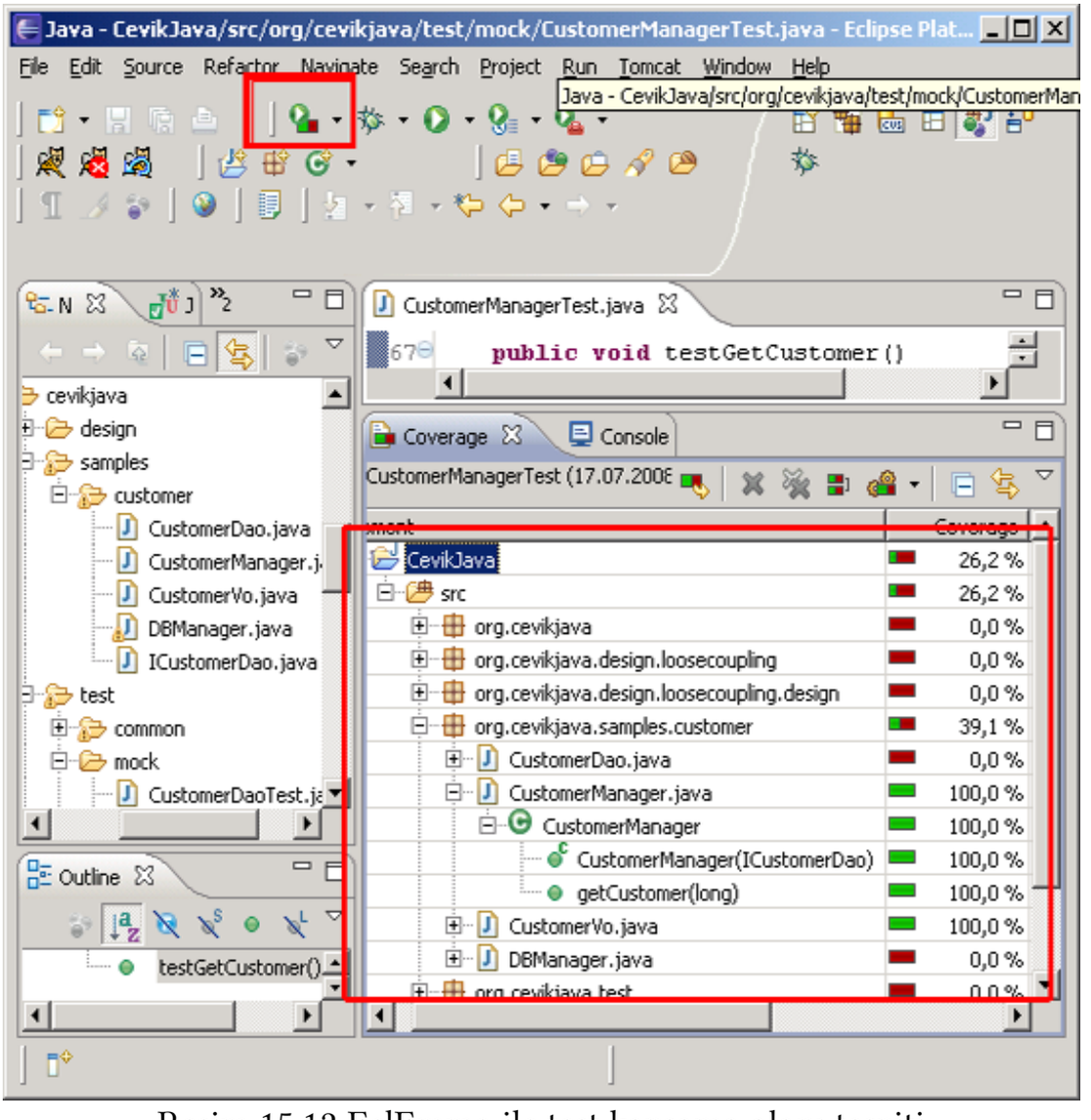
EclEmma Eclipse Plugin

EclEmma plugini ile daha önce tanışmıştık. Test kapsama alanını Eclipse bünyesinde EclEmma programıyla tespit etmek mümkündür. Bir Eclipse plugini

olan EclEmma nın kurulumu resim 15.11 de görüldüğü gibi yapılabilir.



Resim 15.11 EclEmma installasyonu Eclipse altında Help>Software Updates > Find and Install bölümünden yapılabilir.



Resim 15.12 EclEmma ile test kapsama alanı tespiti

Plugin kurulduktan sonra üst panelde resim 15.12 de görüldüğü gibi yeni bir buton oluştu (kare içinde). Bu buton aracılığıyla herhangi bir JUnit test sınıfını seçerek, çalıştırabiliriz. EclEmma arka planda hangi sınıf ve metotların

kullanıldığını paket ve sınıf bazında tespit ederek alt panelde görüntüler. Resim 15.12 de yer alan örnekte CustomerManagerTest sınıfı kullanılmıştır. Bu testin proje (CevikJava) genelindeki eriştiği kapsama alanı %26.2 dir. org.cevikjava.samples.customer paketi için test kapsama alanının %39.1 olduğunu görüyoruz. Bu paket içinde Cusstomermanager ve CustomerVo sınıfları %100 test kapsama alanına girmiştir. Bu demektir ki CustomerManagerTest sınıfı CustomerManager ve CustomerVo sınıfında bulunan her satır kodun çalışmasına sebep olmuş ve %100 test kapsamı seviyesine ulaşılmıştır. EclEmma bunun yanı sıra hangi metotların hangi oranda test kapsama alanında olduğunu gösterir. Örneğin CustomerManager sınıfında bulunan CustomerManager() konstruktör metodu ve getCustomer() metodu %100 işlem görmüştür.

EclEmma test sonucunda test edilen sınıfların işlem gören satırlarını yeşil, işlem görmeyen satırlarını kırmızı renkte gösterir. Bunun bir örneğini Resim 15.13 de görmekteyiz.

```
1  package org.cevikjava.samples.customer;
2
3  /**
4   * Müsteri yönetimini yapan
5   * sinif.
6   *
7   * @author Oezcan Acar
8   *
9   */
10 public class CustomerManager
11 {
12     private ICustomerDao dao;
13
14     public CustomerManager(ICustomerDao  dao)
15     {
16         this.dao =  dao;
17     }
18
19     public CustomerVo getCustomer(long id)
20     {
21
22         CustomerVo vo = this.dao.getCustomer(id);
23         System.out.println("Firstname: "
24                 + vo.getFirstname());
25         System.out.println("Nme: " + vo.getName());
26         return vo;
27     }
28 }
29
```

Resim 15.13 Test kapsama alanı EclEmma tarafından renkli olarak gösterilir.

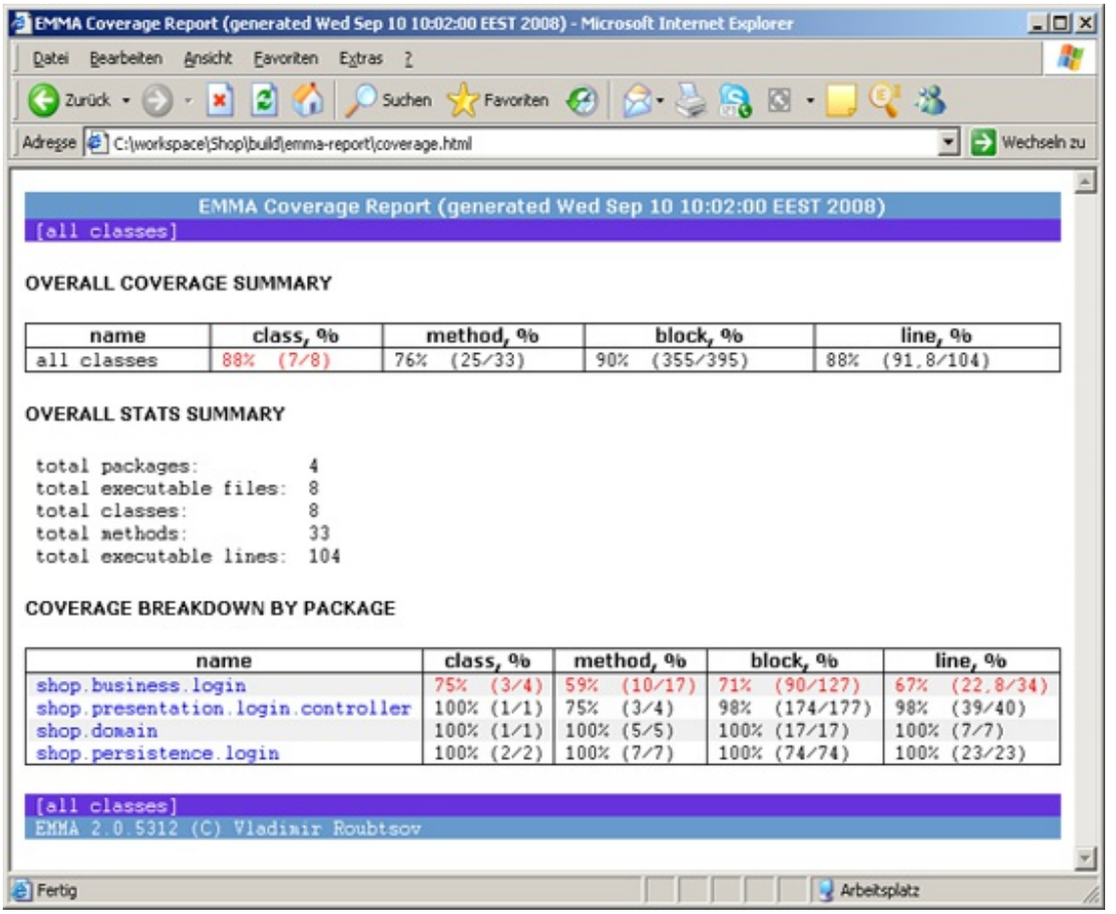
Emma ve Sürekli Entegrasyon

Emma aracını sürekli entegrasyona entegre etmek için kod 15.5 de kullanılan Ant hedefi kullanılabilir.

Kod 15.5 Ant build.xml

```
<property name="emma.dir" location="${basedir}" />
<property name="junit.dir" location="${build.dir.classes}" />

<target name="emma" depends="emma.tool"/>

<target name="emma.tool" depends="clean, compile">

    <available
        file="${basedir}/lib/emma.jar"
        property="emma.available"/>
    <fail
        unless="emma.available"
        message="EMMA not available!"/>

    <property name="emma.lib.dir" value="${basedir}/lib/" />
    <path id="emma.lib.dir" >
        <pathelement location="${emma.lib.dir}/emma.jar" />
        <pathelement location="${emma.lib.dir}/emma_ant.jar" />
    </path>

    <taskdef
        resource="emma_ant.properties"
        classpathref="emma.lib.dir" />

    <emma enabled="true" verbosity="verbose">
        <instr instrpath="${build.dir.classes}/shop"
             destdir="${build.dir.classes}"
             metadatafile="${emma.dir}/metadata.emma"
             merge="true" mode="overwrite">
        </instr>
    </emma>


    <junit
        fork="false"
        failureproperty="tests.failed"
        showoutput="true"
        printsummary="yes"
        haltonfailure="yes">
```

```
        <classpath refid="compile.classpath" />
        <classpath path="${build.dir.classes}" />
        <classpath path="${emma.lib.dir}" />
        <sysproperty
            key="emma.coverage.out.file"
            value="${emma.dir}/coverage.emma" />
        <sysproperty
            key="emma.coverage.out.merge"
            value="true" />

        <test
            name="${junit.testcase.class}"
            haltonfailure="yes"
            outfile="build/junit-result">
            <formatter type="xml" />
        </test>

    </junit>

    <fail if="tests.failed">
      ****************************************************
      ****************************************************
      One or more tests failed. Check the output...
      ****************************************************
      ****************************************************
      </fail>


    <junitreport todir="${junit.report.dir}">
        <fileset dir="${build.dir}">
            <include name="junit-result.xml" />
        </fileset>
        <report format="noframes" todir="${junit.report.dir}" />
    </junitreport>


    <emma enabled="true" verbosity="verbose">
        <report sourcepath="${base.src}">
            <fileset dir="${emma.dir}">
                <include name="*.*" />
            </fileset>
            <html outfile="${emma.report.dir}/coverage.html" />
        </report>
    </emma>
</target>
```



Resim 15.14 Emma test kapsama alanı raporu

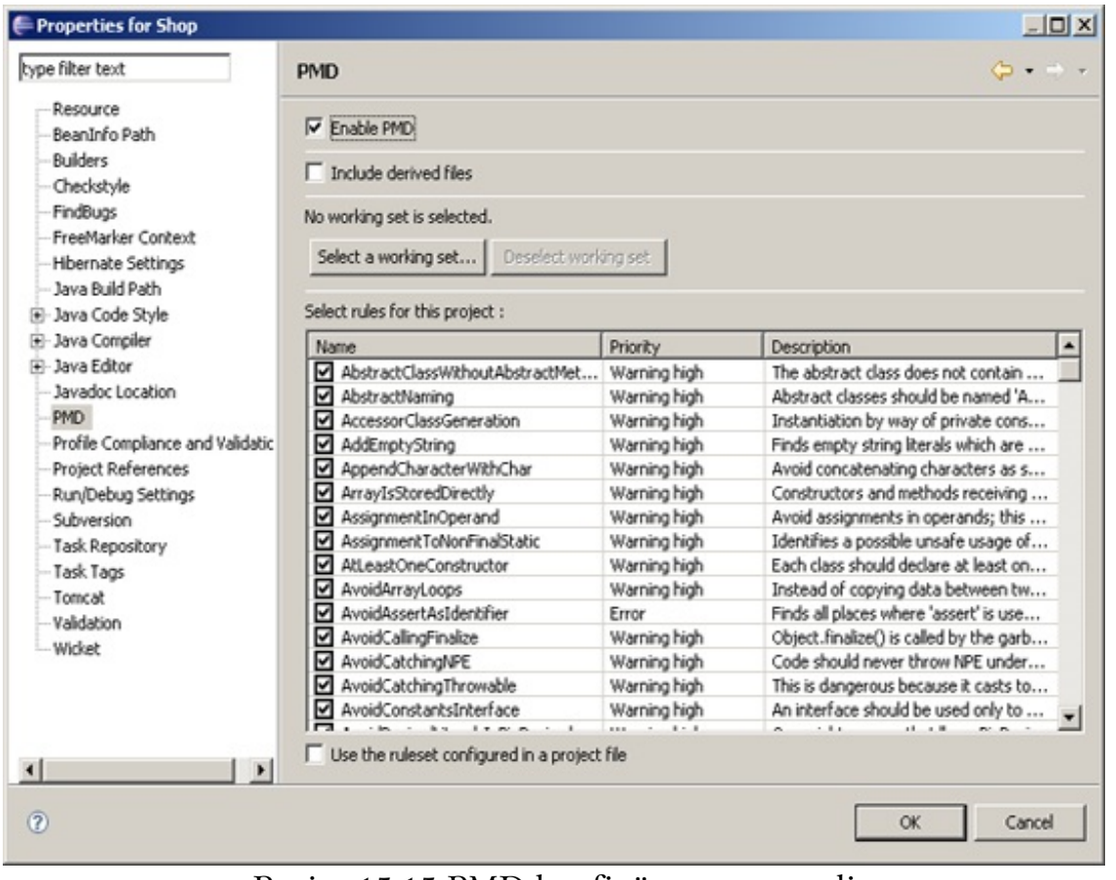
PMD

PMD kod içinde sorunlu ve hatalara yol açabilecek alanların tespiti için kullanılan bir araçtır. PMD ile tespit edilebilecek sorunlar şu şekildedir:

- İlk bakışta görünmeyen hataların (bug) tespit edilmesi
- Boş try/catch/finally/switch bloklarının lokalizasyonu
- Kullanılmayan lokal değişkenler, parametreler ve private olarak tanımlanmış metotların lokalizasyonu
- Optimal olmayan kodun tespiti – örneğin gereksiz String ve StringBuffer kullanımı
- Komplike kod yapılarının tespiti – örneğin gereksiz if blokları, for ve while döngüleri
- Tekrar eden kod – örneğin copy/paste usulüyle kopyalanmış kodun lokalizasyonu

PMD Eclipse Plugin

PMD plugin olaran Eclipe e entegre edilebilir. Gerekli bilgiyi http://pmd.sourceforge.net/integrations.html#eclipse adresinden edinebilirsiniz.



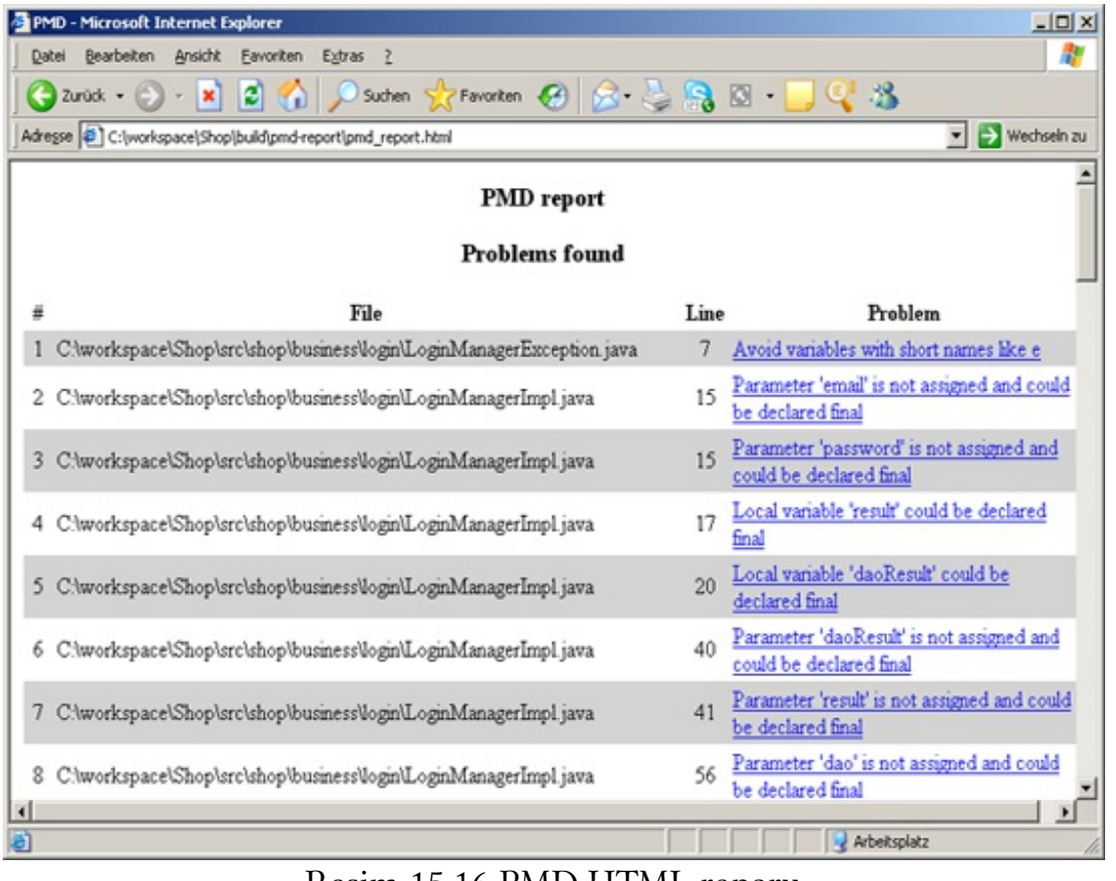
Resim 15.15 PMD konfigürasyon paneli

PMD ve Sürekli Entegrasyon

PMD aracını sürekli entegrasyona entegre etmek için kod 15.6 de kullanılan Ant hedefi kullanılabilir.

```
Kod 15.6   Ant build.xml

<target name="pmd" depends="clean, compile">
    <taskdef
        name="pmd"
        classname="net.sourceforge.pmd.ant.PMDTask"
        classpath="${pmd.home}/lib/pmd-4.1.jar" />

    <pmd
  rulesetfiles="basic,braces,clone,codesize,controversial,
    coupling,design,finalizers,imports,j2ee,javabeans,
    junit,logging-jakarta-commons,logging-java,migrating,
                    naming,optimizations,
```

```
    scratchpad,strictexception,strings,sunsecure,typeresolution,
                    unusedcode">
        <formatter
            type="html"
            toFile="${pmd.report.dir}/pmd_report.html"
            toConsole="true" />

        <fileset
            dir="${base.src}">
            <include name="**/*.java" />
        </fileset>
    </pmd>
</target>
```

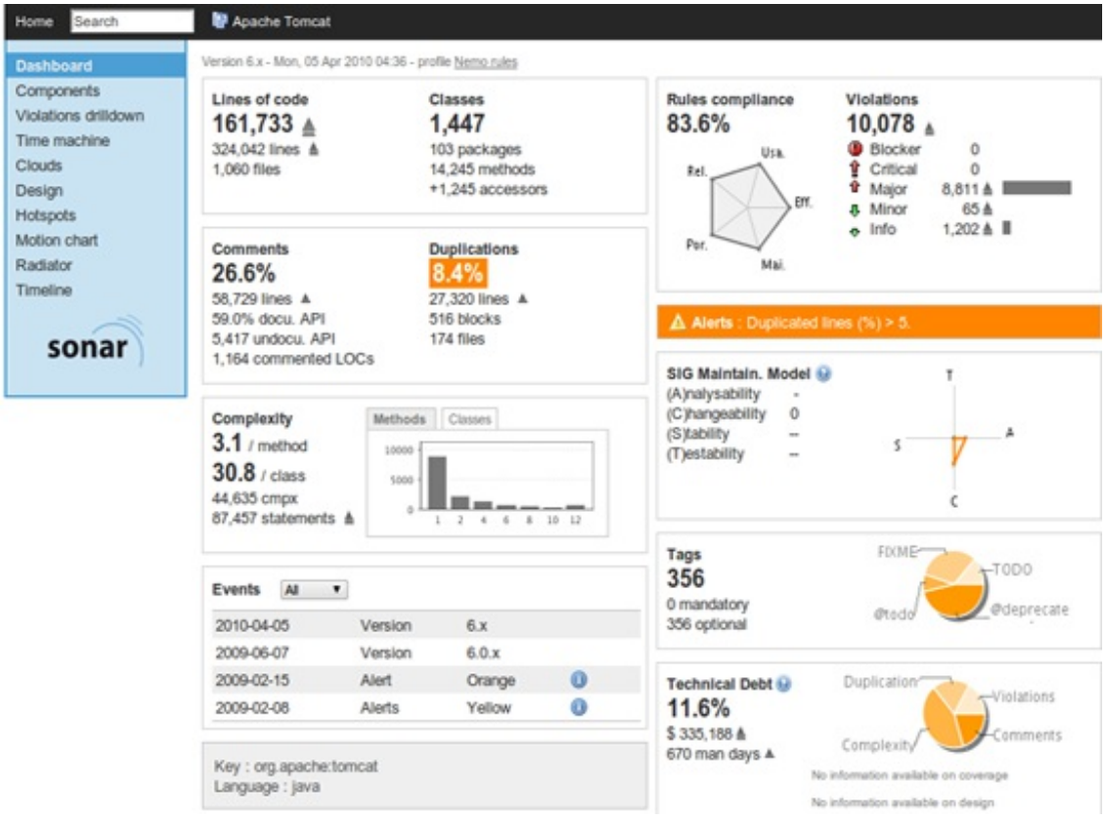


Resim 15.16 PMD HTML raporu

Sonar

Sürekli entegrasyon sürecinde yazılım metriklerinin tespit edilmesinin önemini belirttim. Bu metrikleri elde etmek için kullanılabilecek araçları bu bölümde tanımış olduk. Bu araçların hepsini tek, tek entegre etmek zaman alıcı ve hatalara sebep veren bir süreç olabilir.

Bahsettiğim metrikleri elde etmenin daha kolay bir yolu bulunmaktadır. Sonar ismini taşıyan web tabanlı bir uygulama ile uygulama için geçerli tüm metrikleri her entegrasyon sonunda görebiliriz.



Resim 15.17 Sonar Uygulaması

Bu aracın kurulumu ve kullanımı hakkında bilgiyi http://www.sonarqube.org/ adresinden edinebilirsiniz.

16. Bölüm

Subversion ile Versiyon Kontrolü

Bir Kaosun Hikayesi ...

Anıl o gün işe gitmek için arabasına bindiğinde aklı hala sabaha kadar düşünüp cevap bulmadığı sorudaydı. Boğaz köprüsüne kadar geldiğinin farkına bile varmamış, İstanbul'un o tipik trafiğine takılıp kalmıştı. Anıl 29 yaşında, iyi denebilecek seviyede bir programcıydı. Birçok KOBI'de programcı olarak tek kişilik projelerde çalışmış ve tecrübe edinmişti. Tek başına çalışmaya alışkın olduğu için yazdığı programları 8 GB hafızalı USB Stick üzerinde iş yerinden eve, evden işe taşır ve böylece birden fazla bilgisayar üzerinde iş yerinde ve evde zamanı olduğunda çalışma firsatı bulurdu.

İş yerinde mesai saati bitimi hızlıca tüm projeyi USB Stick üzerinde çekerek kopyalar, kopyalama işlemi 3-5 dakika içinde tamamlandıktan sonra USB sticki cebine atarak, evin yolunu tutardı. Birçok programcı gibi eve geldiğinde hemen bilgisayarın başına oturur ve önce maillerini kontrol eder, daha sonra USB stick üzerindeki programı bilgisayarına aktarır ve iş yerinde çözemediği problem üzeride çalışmaya devam ederdi. Tabii eşi bu durumdan pek hoşnut değildi. Yazdığı programları her zaman yanında bulundurmak ona güven verirdi. Herhangi bir bilgisayar üzerinde işine devam edebileceğini bilmek bilgisayar mühendisi olarak onun öz güvenini artırırdı.

Anıl hala birçok programcının beraber çalışmak zorunda olduğu bir projede yer alamamış ve bu yüzden bir takım içinde nasıl çalışılacağı hakkında bilgiye sahip olamamıştı. Diğer mühendis arkadaşlarıyla görüşmelerinde versiyon kontrolü, dal (branch) ve release yönetimi gibi terimler duymuş, lakin bu konuda detaylı araştırma ve uygulama yapma firsatı bulamamıştı. Son zamanlarda yazılım sektöründe meydana gelen metot değişiklikleri Anıl'ı tedirgin etmeye başlamıştı. Oda her insan gibi değişikliklere karşı içgüdüsel bir karşı tavır almıştı, ama bilgisayar mühendisi olarak er ya da geç bu yeni metotlarla tanışmak ve çalışmak zorunda olduğunu biliyordu.

İki hafta önce patronu onun yanına gelerek, çok büyük bir ihaleyi aldıklarını ve iyi bir programcı ekibi kurarak, yazılım sürecini başlatmaları gerektiğini söylemişti. Anıl'ın yeni görevi yeni programcı ekibi için elemanların işe alınması ve takım yöneticiliği olacaktı. Bu durum Anıl'ı tedirgin etmişti. Bir takım içinde kod paylaşımı USB sticklerle mümkün olacak mıydı? Yoksa kodun merkezi bir sunucu üzerinde bulundurulması mı gerekecekti? İşte bu ve bunun gibi sorular sabaha kadar beynini meşgul etmişti.

Boğaz köprüsünün üzerinde geldiğinde boğazın mavi suları içini biraz

rahatlatmış ve tedirginliğini azaltmıştı. Bugün yeni takım arkadaşlarıyla tanışacak ve yeni projenin temeli atılacaktı. Trafiğin yoğunluğundan dolayı ofise geç kalmıştı. Toplantı odasına girdiğinde patronu ve 3 yeni takım arkadaşını onu bekler durumda buldu. Tanışma faslının ardından proje hakkında konuşuldu. Patron kısa bir zaman sonra tüm ekibe başarılar dileyerek, kendi işinin başına döndü. Gelişmelerden Anıl aracılığıyla haberdar olmak istediğini ve kısa bir zaman içinde çalışır bir prototip oluşturulması için hemen çalışmalara başlanmasını istedi.

Anıl ve yeni çalışma arkadaşları Aydın, Müge ve Mustafa kolları sıvayarak, işe başladılar. USB stick ile kod paylaşımının mümkün olmadığını bildikleri için kodun merkezi bir sunucu üzerinde bulundurulmasına karar verdiler. Sonuç itibariyle herkesin görevi belli idi ve kod paylaşımında belli kurallar takip edildiği sürece bir sorun çıkması beklenmiyordu. Takım mensubu her programcı akşam evine gitmeden önce yaptığı program değişikliklerini sunucuya kopyalanacak ve böylece kendi kodunu takım içinde paylaşmış olacaktı. Bu yöntemin ne kadar kötü sonuçlar doğuracağını daha sonra çok iyi anlayacaklardı.

İlk iki hafta herkes kendi modülü üzerinde çalışmalarını sürdürdü ve oluşturduğu kodu sunucuya yükledi. Programcılar her sabah en güncel kodu sunucudan edinerek, işlerine devam ettiler. Çalışmalar hızlı bir şekilde devam etti ve ikinci haftanın sonunda ilk prototip çalışır hale geldi.

Anıl patronunun yanına giderek, ilk sürümün tamamlandığını bildirdi. Anıl'ın patronu ve kalite kontrol bölümünden Veli bey prototip üzerinde incelemelerde bulundular. Prototip birçok modüllü ihtiva etmese de, çalışır durumda olan bir program parçasıydı. Patron prototipin kendisinde iyi bir intiba bıraktığını söyleyerek, çalışmalara devam edilmesi direktifini verdi. Anıl için bu çalışma metodlarının ne kadar iyi olduğunun bir ispatıydı. Kendisi ve çalışma arkadaslarıyla gurur duydu ve işinin başına döndü.

Üçüncü haftanın başında Müge rahatsızlanmış ve işe gelememişti. Bir hafta boyunca dinlenmesi ve iyileşmesi gerekiyordu. Yapılan proje planına ayak uydurulabilmesi için Müge'nin görevlerinin ekip içindeki diğer programcılara dağıtılması gerekiyordu. Kısa bir kriz toplantısının ardından Müge'nin üzerinde çalıştığı modüller Mustafa ve Aydın arasında paylaştırıldı. Bu durum Mustafa ve Aydın'ı tedirgin etmişti. İki hafta boyunca ekip içinde neredeyse kimse birbiriyle beraber çalışmadan kendi modülleri üzerinde çalışmıştı. Açıkcası Mustafa ve Aydın Müge'nin ne yaptığı hakkında bir bilgiye sahip değildiler. O

gün Müge'nin kodlarını inceleyerek geçti.

Mustafa ve Aydın kendi modülleri yanı sıra Müge'den devraldıkları modül üzerinde ortak çalışmaya başladılar. Her programcı mesai bitiminde yaptığı tüm değişiklikleri sunucuya aktarıyor ve mesai başlangıcında tüm projeyi sunucudan alıyordu. Zaman içinde Mustafa ve Aydın farkında olmadan aynı metodlar üzerinde değişiklik yapmaya başladılar. Bunun nasıl bir sonuç doğuracağı ortadaydı: en son programcının sunucuya kopyaladığı programlar diğer programcılar tarafından yapılan değişiklikleri yok ediyordu. Tabi bu durumun ortaya çıkması pek uzun bir zaman almadı. Mustafa ertesi gün çalıştığı en son sınıf kodunda oluşturduğu yeni metodu bulamadı. Bunun üzerine Aydın'ın yanına giderek sınıf üzerinde bir değişiklik yapıp, yapmadığını sordu. Aydın sınıf üzerinde bir değişıklık yapmamıştı, ama dün mesai bitiminde kodları en son sunucuya kopyalayan Aydın oldugu için Mustafa'nın yaptığı değişiklikler yok olmuştu. Bir version kontrol sistemi kullanmadıkları için herhangi bir dokümanın belirli bir tarihteki durumunu elde etmeleri de mümkün değildi.

Bu hikayenin sonucunu büyük bir ihtimalle kestirebiliyorsunuz. Hayır! Arkadaşlar işlerinden olmadılar, ama patrondan büyük bir azar işittikten sonra versiyon kontrol sistemi hakkında verilen bir seminerin yolunu tuttular :)

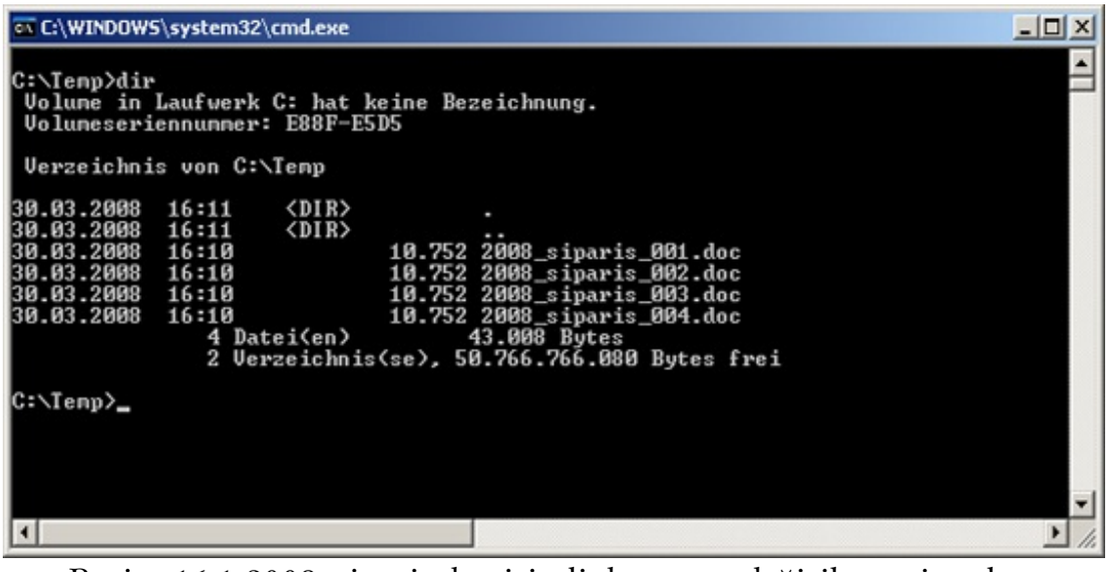
Doküman varsa versiyon vardır...

İnsanoğlu okuma, yazmayı icat etmeden önce mağara duvarlarına resimler yaparak düşüncelerini şekillendirmeye başladı. Araştırmalara göre ilk yazının Sümer'liler tarafından İsa'dan önce 3500 civarında icat edildiği söylenmektedir. O devrin insanları yazı benzeri işaretler kullanarak ilk dokümanları oluşturmuş ve bu dokümanları iletişim aracı olarak kullanmışlardır. Günümüzde latin alfabesinde yer alan kelimeleri kullanarak bilgisayarda dijital dokümanlar oluşturuyoruz, arkadaşlarımıza eposta gönderiyoruz ya da elimize kağıt ve kalem alarak mektup yazıyoruz. Bu işlemlerin sonucunda dijital olan ya da olmayan bir doküman oluşuyor.

Her doküman oluşturulduğu ilk saniyeden itibaren bir versiyon ihtiva eder. Bir versiyon dokümanın geleceğe doğru olan yolculuğunda durak yaptığı istasyonlardan birisidir. Zaman içinde doküman değişikliğe uğrar. Her değişikliğin ardından dokümanın yeni bir versiyonu oluşur. Dokümanın herhangi iki versiyonu arasındaki fark, bu iki versiyonun oluşumu için geçen

zaman diliminde doküman üzerinde yapılan tüm değişiklikleri ihtiva eder.

Bir dokümanın versiyonları arşivlenmediği sürece o doküman üzerinde yapılan değişiklikler takip edilemez.



Resim 16.1 2008_siparis.doc isimli dosyanın değişik versiyonları

Resimde görüldüğü gibi 2008 senesi için yapılan siparişler 2008_siparis_xxx.doc isminde bir doküman içinde tutulmaktadır. Her yeni sipariş için dokümanın yeni bir versiyonu oluşturulmuş ve böylece bu dokümanın tarihçesi arşivlenmiştir. 2008_siparis_004.doc en son versiyondur ve bu Doküman bünyesinde meydana gelen tüm değişiklikleri ihtiva etmektedir. Dokümanın değişik versiyonları arşivlendiği için değişik versiyonlar arasında kıyaslama (compare) yapılarak meydana gelen değişiklikler takip (track) edilebilir.

Versiyon kontrolü nedir?

Bir dokümanın oluşum sürecini ve değişik versiyonların takibi ve arşivlenmesi için kullanılan metot ve sistemlere versiyon kontrolü adı verilir. Genelde yazılım sektöründe projelerin yönetimi için versiyon kontrol sistemleri kullanılır. Birden fazla programcının kod paylaşımı ve yapılan değişiklerin takibi için bir versiyon kontrol sisteminin kullanımı kaçınılmazdır. Daha sonra yakından inceleyeceğimiz gibi oluşturulan bir yazılım ürününün (program) değişik versiyonlarının oluşturulması ve bu versiyonlardaki hataların (bug) giderilmesi için kullanılan versiyon kontrol sistemi değişik araçlar ve yöntemler ihtiva etmektedir. Bu metotlar kullanılarak yazılım süreci desteklenir.

Bir versiyon kontrol sistemini depo gibi düşünebiliriz. Oluşturulduğumuz her

doküman bu depoya gönderilir. Depo dokümanların ve değişik versiyonlarının nasıl yönetileceğini bilir. Programcı olarak deponun içinde neler olup, bittiğini bilmek zorunda değiliz. Sadece bir dokümana gerek duyduğumuz zaman depoya başvurarak, gerekli dokümanı ediniriz. Bu doküman üzerinde değişiklik yapabilir ve değiştirilen dokümanı tekrar depoya gönderebiliriz. Bu esnada dokümanın değişik versiyonları oluşur. Oluşan değişik versiyonlar üzerinde de kafa yormamıza gerek yoktur, çünkü depo bu versiyonların yönetimi üstlenir. Depoya danışarak, bir dokümanın tarihçesini edinebiliriz. Depo hangi dokümanın kim tarafından ne zaman değiştirildiğini her zaman bilir. Kayıtlarda iz bırakmadan bir doküman üzerinde değişiklik yapmak mümkün değildir.

Versiyon kontrol sistemlerinde kullanılan depolara repository ismi verilir. Çoğu versiyon kontrol sistemi bünyelerinde veri tabanları kullanarak repository oluşturmaktadırlar. Örneğin Subversion da Berkley DB veri tabanı kullanarak bir repository oluşturulabilir. Bunun yanı sıra versiyon kontrol sistemi tarafından normal sistem dosyaları (file repository) kullanılarak da repository ler oluşturulabilir. Bu gereksinimlere bağlı olarak yapılması gereken bir seçimdir.

Versiyon kontrol sisteminin merkezinde bir repository olduğu için bu repository nin güvenli bir bilgisayar üzerinde olmasına dikkat edilmelidir. Bu bilgisayar erişilemez durumda olduğunda ya da en kötü ihtimalle bozulduğunda tüm versiyon kontrol sistemi çalışmaz hale gelecektir. Mutlaka düzenli aralıklarla repository nin kopyası (backup) alınmalıdır.

Kod paylaşımını kolaylaştırmak için çoğu modern versiyon kontrol sistemleri bilgisayar ağı üzerinden kullanılabilir şekilde çalıştırılabilir (network enabled). Bu şekilde programcının bulunduğu yer önemini yitirmekte ve bilgisayar ağı mevcut olduğu sürece yerden bağımsız olarak versiyon kontrol sistemi kullanılabilmektedir. Programcı bilgisayar ağı üzerinden versiyon kontrol sisteminin barındırdığı repository ye bağlanarak, kod paylaşımını gerçekleştirebilir.

Peki bir programcı bilgisayar ağı üzerinden erişim olmadığı durumlarda çalışmasını nasıl devam ettirebilir? Modern versiyon kontrol sistemleri devre dışı mod (disconnected ya da offline mode) olarak tabir edebileceğimiz çalışma tarzını desteklemektedirler. Programcı iş yerinden ayrılmadan önce üzerinde çalışmak istediği projeyi repository den alır ve bilgisayarına yükler. Programcı böylece kendi bilgisayarında projenin bir kopyasına sahip olur (working copy). Gün içinde programcı proje üzerinde çalışarak bir takım değişiklikler yapar.

Programcı ertesi gün tekrar ofise gelerek, yaptığı değişiklikleri repository ile senkronize eder.

Bir versiyon kontrol sistemi seçilirken programcı ekibinin gereksinimleri göz önünde bulundurulmalıdır. Her versiyon kontrol sistemi istenilen esnekliği sağlayamayabilir. Edindiğim tecrübeler doğrultusunda Subversion versiyon kontrol sisteminin programcı ekibin gereksinimlerine büyük oranda cevap verebileceğini söyleyebilirim.

Bir Başarı Hikayesi ...

Cem 34 yaşında tecrübeli bir bilgisayar mühendisi idi. ODTÜ bilgisayar mühendisliği fakültesini bitirdikten ve askerden geldikten sonra Türkiye'nin en başarılı İnternet firmalarında çalışmış ve takım tecrübesi edinmişti. Bir projenin başarıya ulaşması için gerekli tüm metot ve araçlara hakimdi.

Her zaman olduğu gibi o günde erkenden ofise gelmiş ve çalışmaya başlamıştı. Cem ilk önce kendi bilgisayarında yer alan kodları merkezi versiyon kontrolü sistemi aracılığıyla güncelledi (update). Bir gün önceki mesai bitiminde üzerinde çalıştığı kodların çalışır durumda olduğunu kontrol ettikten sonra merkezi versiyon kontrolü sistemine eklemişti (commit). Takımda yer alan her programcı her sabah aynı işlemi yapardı. Bu bir alışkanlık haline gelmişti. Her programcı kendi bilgisayarında kodların bir kopyasına (working copy) sahipti. Programcılar sahip oldukları kodları her sabah güncelledikten ettikten sonra çalışmalarına devam ederlerdi.

Cem o sabah güncelleme işlemi sırasında takım arkadaşı olan Sevgi'nin bazı sınıflar üzerinde değişiklikler yaptığını fark etti. Sevgi Siparis sınıfını da değiştirmiş ve yeni değişkenler eklemişti. Cem'in üzerinde çalıştığı sınıflar Siparis sınıfı ile bağlantılı olduklarından, Sevgi'nin yanında giderek, yapılan değişikler hakkında bilgi almayı düşündü, lakin Sevgi o gün müşteri görüşmesindeydi. Cem tekrar bilgisayarının başına döndü. Merkezi versiyon kontrol sistemi proje bünyesinde oluşan her dokümanı ve bu dokümanların değişiklikler sonunda meydana gelen versiyonlarını bünyesinde barındırıyordu. Cem, Siparis sınıfında yapılan değişiklikler listesini aldığında, Sevgi tarafından eklenen en son yorumu gördü:

Siparis sınıfına, siparişin gönderileceği adres için gerekli değişkenleri ekledim.

Yapılan değişiklikler merkezi versiyon kontrol sistemine eklenmeden önce her programcı yaptığı değişiklikleri açıklayıcı bir yorum eklemek zorundaydı. Böylece bu yorumları okuyarak, ne gibi değişikliklerin yapıldığını anlamak kolaydı. Ayrıca merkezi versiyon kontrolü sisteminin bünyesinde barındırdığı bazı araçlar aracılığıyla bir dokümanın değişik versiyonlarını kıyaslamak ve yapılan değişiklikleri bu şekilde görmekte mümkündü. Her doküman kayıtsız şartsız merkezi versiyon kontrolü sisteminin himayesindeydi ve hiç kimse iz bırakmadan bir doküman üzerinde değişiklik yapamazdı. Kimin ne zaman ne yaptığı her zaman merkezi versiyon kontrolü sistemine danışılarak takip edilebilirdi.

Cem yeni Siparis sınıfını göz önünde bulundurarak, kendi kodu üzerinde gerekli değişiklikleri yaptı ve diğer takım arkadaşları ile paylaşmak için tüm değişikleri merkezi versiyon kontrolü sistemine ekledi. Bu işlemin ardından üzerinde çalıştığı tüm dokümanların yeni versiyonları merkezi versiyon kontrolü sisteminde yerini almıştı. Bu şekilde kodun güncellenmesi ve tekrar merkezi versiyon sistemi üzerinden paylaşımı tüm takımın efektif bir şekilde çalışmasını sağlıyordu.

Öğleden sonra Sevgi'den telefon geldi. Sevgi müşteride olduğunu ve orada kullanılan versiyonda bir hata bulduklarını söyledi. Bu hatanın giderilmesi gerekiyordu, çünkü müşteri için program kullanılmaz hale gelmiş ve hatalı hesaplar üretiyordu. Cem bu hatayı ofiste düzeltebilir ve müşteriye oluşturulan yeni versiyonu gönderebilirdi. Bunun için bilmesi gereken iki nokta vardı: 1.) program hatası nerede oluşuyordu? 2.) Müşteri programın hangi sürümünü kullanıyordu. Sevgi hatanın Fatura sınıfında olduğunu ve KDV nin yanlış hesaplandığını söyledi. İkinci sorunun cevabı ise v.3.5 idi, yani müşterinin kullandığı sürümün numarası 3.5 idi. Müşteri için bir sürüm oluşturulmadan önce merkezi versiyon kontrol sisteminde müşteri için bir etiket (tag) oluşturulurdu. Bu etiket v.3.5 ismini taşıyordu. Bu etiket ile programın o sürümünde bulunan tüm sınıf ve dosyalarına ulaşmak mümkündü.

Cem v.3.5 etiketini taşıyan sürümü kodları incelemek için versiyon kontrol sisteminden kendi bilgisayarına indirdi. Bir etikete sahip dosya üzerinde değişiklik yapmak mümkün değildi. Aynı etikete sahip dosyalardan herhangi birisi üzerinde değişiklik yapmak, etiketin belirli bir tarihte belirli bir amaç için oluşturulma ilkesine ters düşmekteydi. Yeni bir dalın (branch) oluşturulması gerekiyordu. Cem v.3.5 etiketini baz alarak yeni bir dal oluşturdu ve hatayı bu dal içinde giderdi. Programı test ettikten sonra v.3.6 isminde yeni bir etiket kullanarak, programın yeni sürümünü oluşturdu. Cem bu sürümü CD üzerinde

kurye ile müşteride olan Sevgi'ye gönderdi.

Bu hikaye devam eder, ama görüldüğü gibi arkadaşlar işlerine hakimler ve gözümüz arkada kalmadan onların yanından ayrılabiliriz :)

Çevik Süreçlerde Versiyon Kontrolü

Çevik süreçlerin olmazsa olmazlarından birisi versiyon sistemi kullanımıdır. İteratif ve inkrementel yapıya sahip olan çevik süreçlerde versiyon kontrol sistemler merkezi bir rol oynar. Bir çevik projede versiyon kontrol kullanımını gerekli kılan nedenler şöyledir:

- Her iterasyonun fiziksel olarak diğer iterasyonlardan ayırt edilebilmesi için o iterasyon bünyesinde oluşan ve değişikliğe uğrayan dosyaların etiketlenmesi gerekir. Her iterasyon sonunda müşteriye çalışır bir sistem sunulur. Sistem iterasyonun etiketini taşır. Böylece hangi sürümün müşteri tarafından kullanımda olduğu anlaşılır.
- Birden fazla iterasyonda oluşan sistem entegre edilerek, bir versiyon numarasıyla etiketlenir. Bu versiyon müşteriye kullanım için verilir. Versiyon numarası ile hangi sürümün kullanımda olduğu anlaşılır.
- XP tekniklerinden ortak sorumluluğu (collective ownership) uygulayabilmek için programcılar arası kod paylaşımını kolaylaştırmak gerekmektedir.
- XP tekniklerinden sürekli entegrasyonu (continuous integration) gerçekleştirebilmek için merkezi bir versiyon kontrol sistemine ihtiyaç duyulmaktadır.

Subversion

Subversion açık kaynaklı (open source) versiyon kontrol sistemidir. Bazı özellikleri şöyledir:

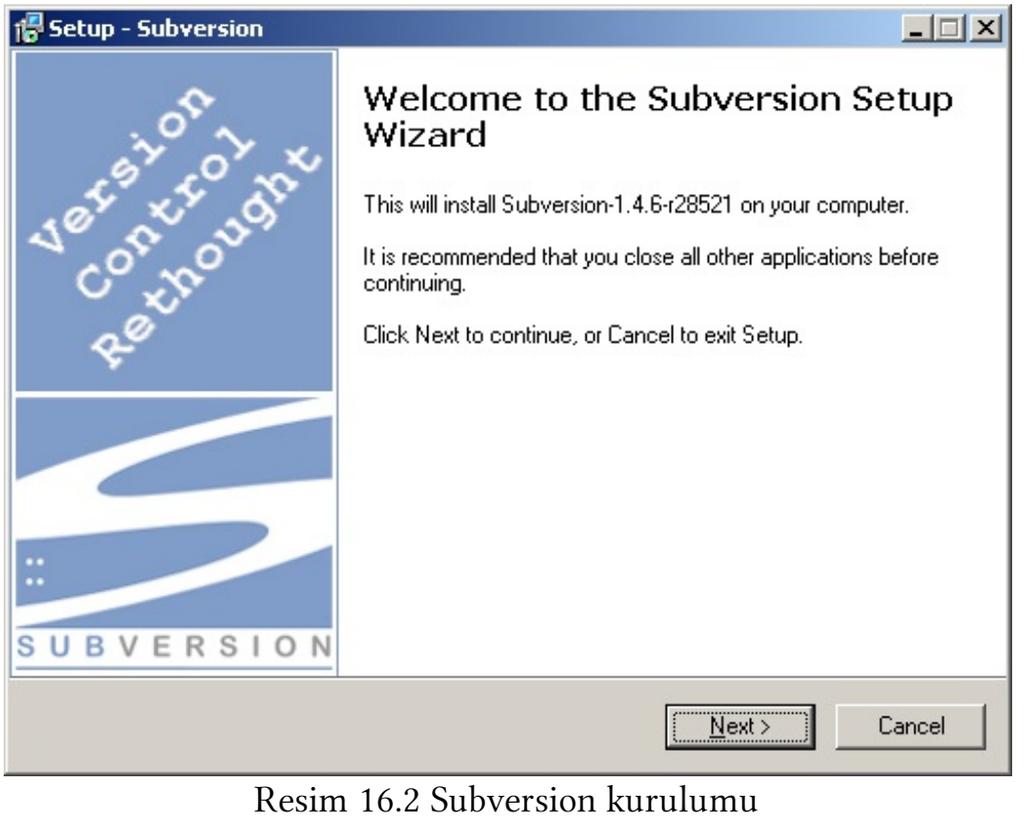
- Subversion CVS örnek alınarak yapılmıştır. Amaç CVS den daha iyi bir versiyon kontrol sistemi oluşturmaktı. Bu yüzden Subversion birçok CVS özelliğine sahiptir.
- Subversion dizinlerin de normal dosyalar gibi versiyonlarını oluşturur.
- Kopyalama, silme ve isim değiştirme işlemlerinde Subversion tarafından yeni versiyonlar oluşturulur.
- Subversion da yapılan işlemler ya hep ya hiç prensibiyle gerçekleşir, yani

commit ler atomiktir (atomic commits).
- Dal (branch) ve etiket (tag) oluşturulması copy işlemi kullanılarak gerçekleştirildiği için kısa sürer.
- File locking mekanizması kullanılarak dosyaların üzerinde değişiklik yapılması engellenebilir.
- Subversion bir Apache web sunucu üzerinde erişilebilir hale getirilebilir.
- Svnserve komutuyla Subversion versiyon kontrol sunucusu olarak görev yapabilir.

Bunun yanı sıra daha birçok özelliği olan Subversion çevik projelerde vazgeçilmez bir araç haline gelmiştir.

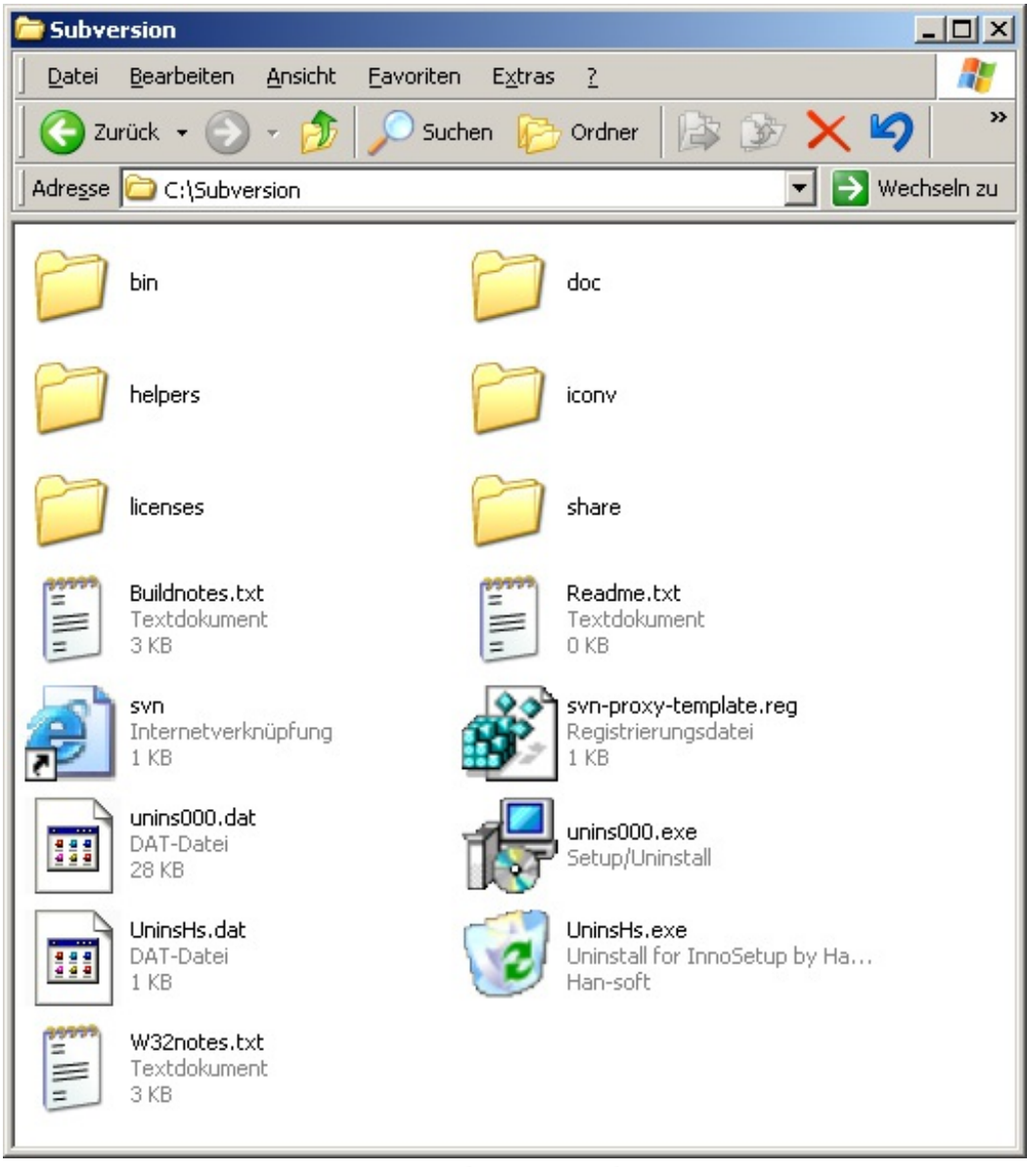
Subversion Windows Kurulumu

Bu bölümde Subversion ın Windows işletim sistem üzerinde kurulumunu inceleyeceğiz. Kullandığım sürüm 1.4.6 dır.



Resim 16.2 Subversion kurulumu

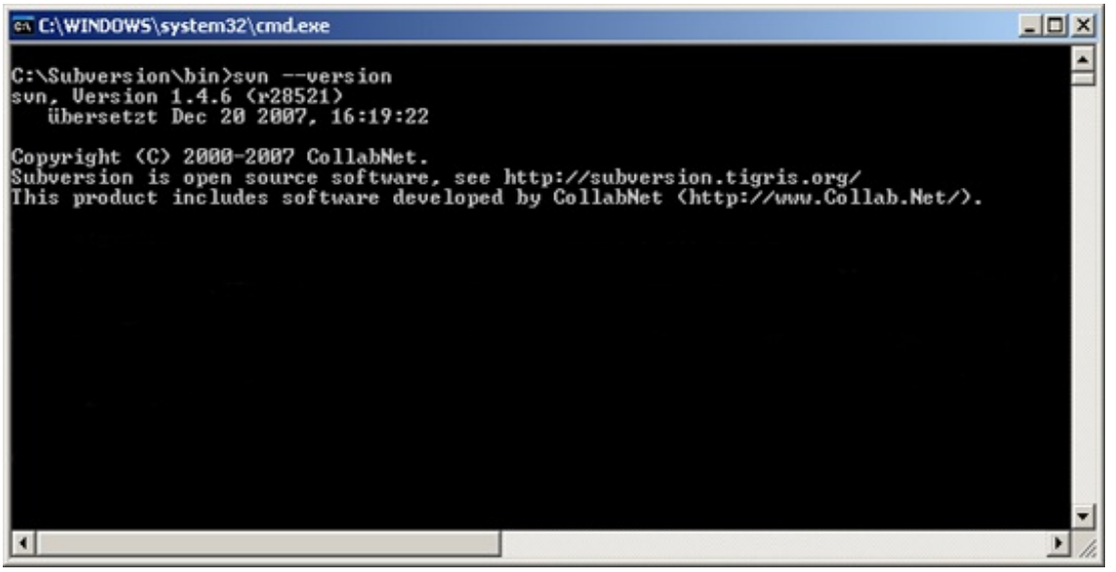
Kurulum tamamlandıktan sonra seçilen dizine bağlı olarak aşağıdaki dizin yapısı oluşacaktır.



Resim 16.3 Subversion dizin yapısı

Subversion ın grafiksel arayüzü yoktur. İşlemleri bin dizininde bulunan komutlar aracılığıyla bir Dos Shell altında gerçekleştirmemiz gerekiyor.

Bin dizininde yer alan svn komutu ile kurulumu kontrol edebiliriz:

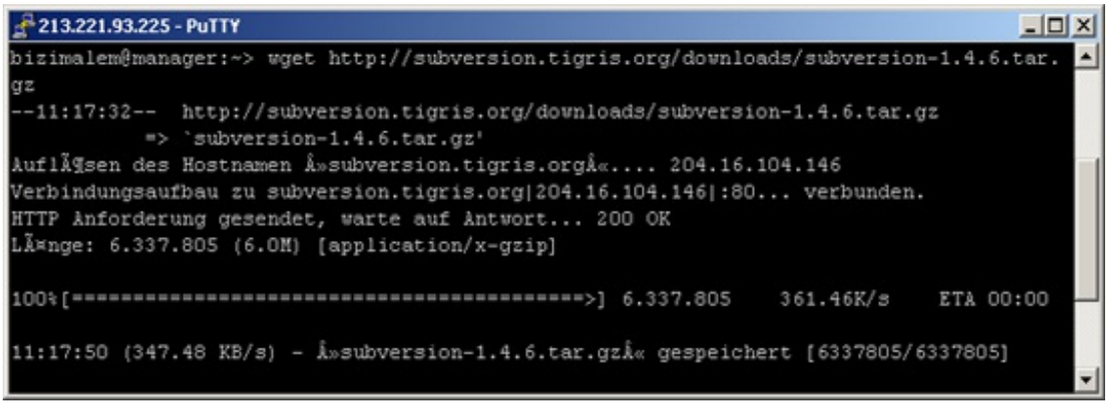


Resim 16.4 svn komutu

Subversion Linux / Unix Kurulumu

Subversion ı Red Hat, FreeBsd ve Solaris gibi değişik Linux / Unix işletim sistemlerinde kullanmak mümkündür. Çoğu zaman belli bir işletim sistemi için hazırlanmış kurulum paketini kullanılabilir.

Bu bölümde Subversion ın kaynak kodlarını kullanarak, nasıl kurabileceğimizi inceleyeceğiz. Eğer Linux / Unix işletim sistemli bilgisayarınızın internet bağlantısı varsa, wget komutu ile kaynak kodun yer aldığı paketi edinebilirsiniz:

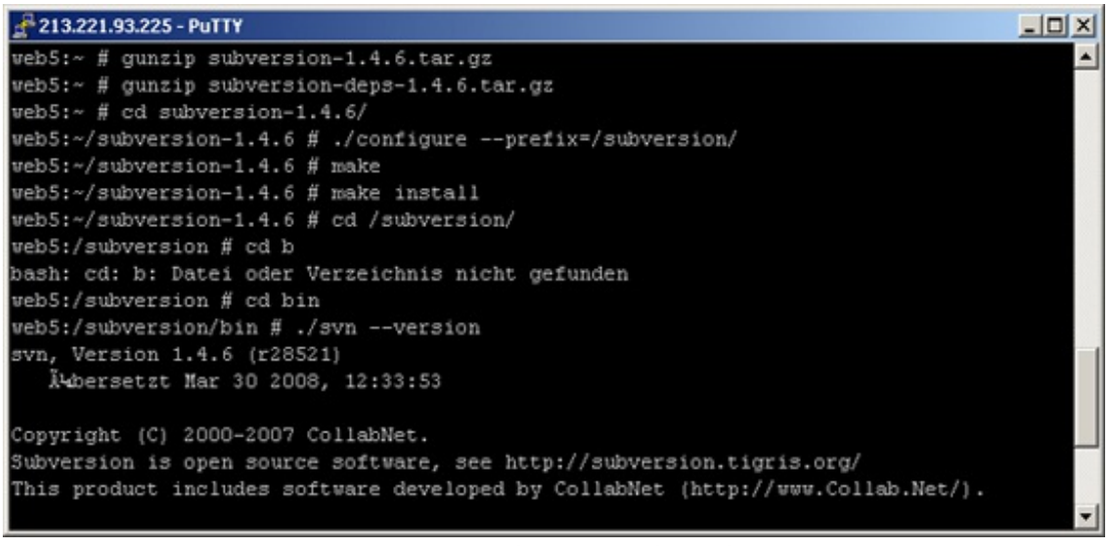


Resim 16.5 wget ile kurulum paketini bu adresten temin edinebilirsiniz

Subversion kurulumu için kaynak kodların bulunduğu iki paketin indirilmesi gerekmektedir:

1. subversion-1.4.6.tar.gz http://subversion.tigris.org/downloads/subversion-1.4.6.tar.gz
2. subversion-deps-1.4.6.tar.gz

http://subversion.tigris.org/downloads/subversion-deps-1.4.6.tar.gz



Resim 16.6 Subversion kurulumu

Kurulum için atılması gereken adımlar şöyledir:

- gunzip subversion-1.4.6.tar.gz
- gunzip subversion-deps-1.4.6.tar.gz
- ve subversion-1.4.6
- ./configure –prefix=/subversion/
- make
- make install

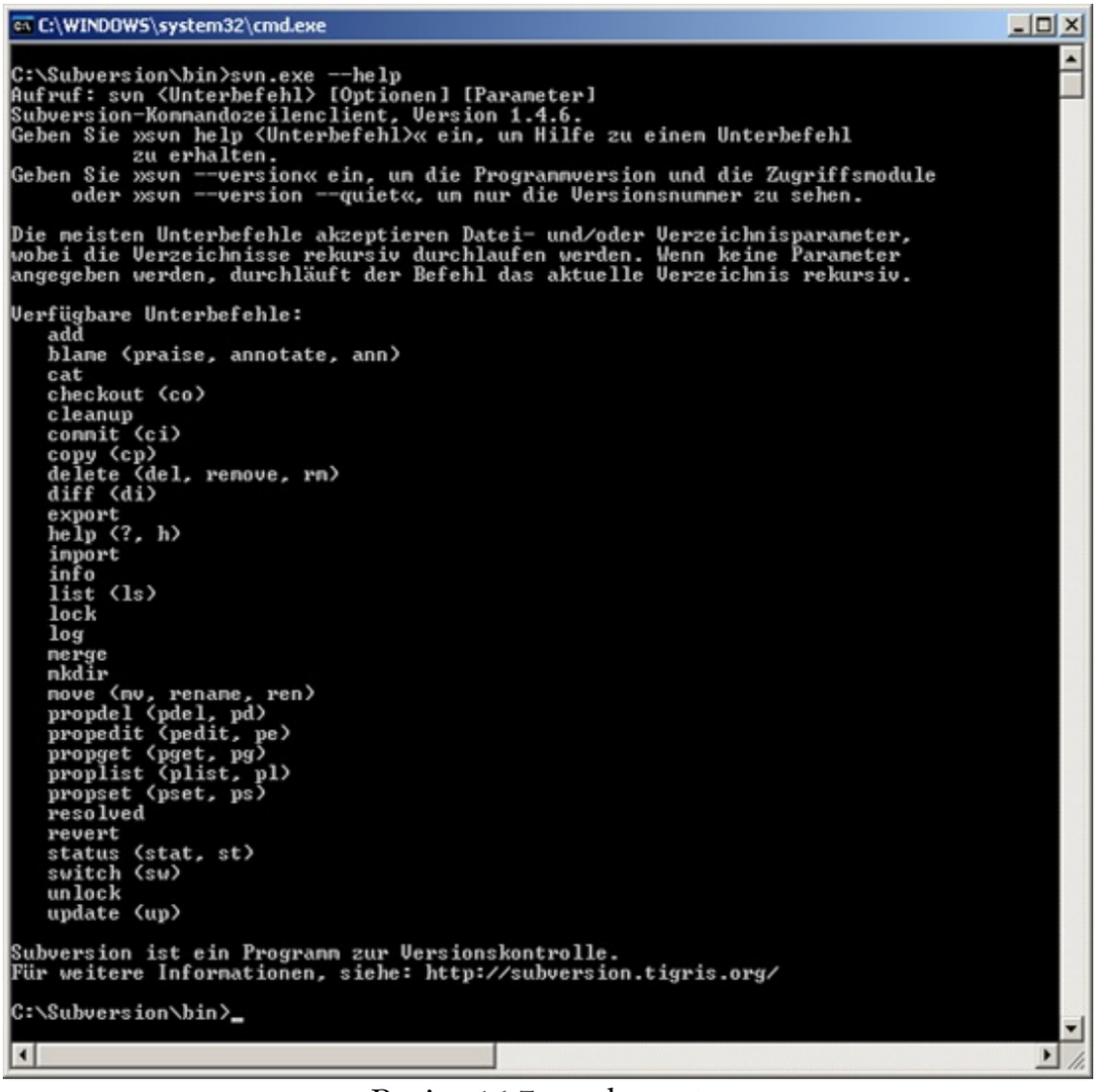
Bu işlemlerin ardından /subversion/bin dizininde Subversion komutları yer alır.

Subversion Komutları

Kurulum işlemi ardında bin dizininde aşağıda yer alan programlar bulunacaktır.

svn

Programcılar tarafından kullanılan client programdır. Repository de bulunan bir dokümana ulaşmak ya da bir doküman üzerinde yapılan değişiklikleri repository ye göndermek için kullanılır.



Resim 16.7 svn komutu

svnadmin

Repository lerin oluşturulması ve yönetiminde kullanılır. Bu program sadece Subversion sunucusu üzerinde çalıştırılabilir. Bilgisayar ağı üzerinden repository lerin yönetimi mümkün değildir.

svnlook

Repository lerin kontrolü için kullanılır. Svnadmin gibi sunucu üzerinde kullanılabilir. Bu program repository bilgilerini sadece okuyabilir, değiştiremez.

svnversion

Bu program ile üzerinde çalışılan dokümanların revizyon numarası öğrenilebilir.

svnserve

Kullanılan repository leri bilgisayar ağı üzerinde erişilir hale getirmek için svnserv programı kullanılır. Svnserve ile bir Subversion sunucusu kurulmuş olur. IP adresi üzerinden bu sunucuda bulunan repository lere erişilir.

svndumpfilter

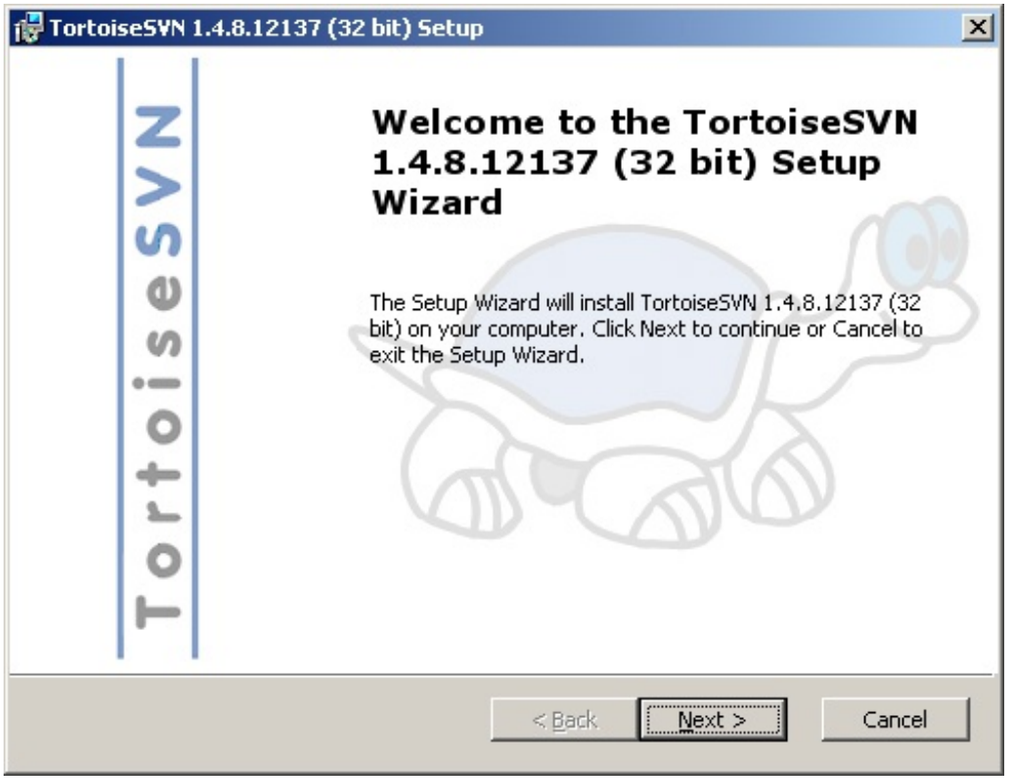
Bu filtre programı ile svnadmin komutuyla oluşturulmuş repository yığınları (dump) içinde belirli kriterlerde arama yapılabilir.

svnsync

Bu program aracılığıyla herhangi bir repository nin sadece okunabilir ama değiştirilemez bir kopyası oluşturulabilir. Oluşturulan kopya ana repository ile bu repository üzerinde yapılan değişiklikleri yansıtacak şekilde senkron tutulur.

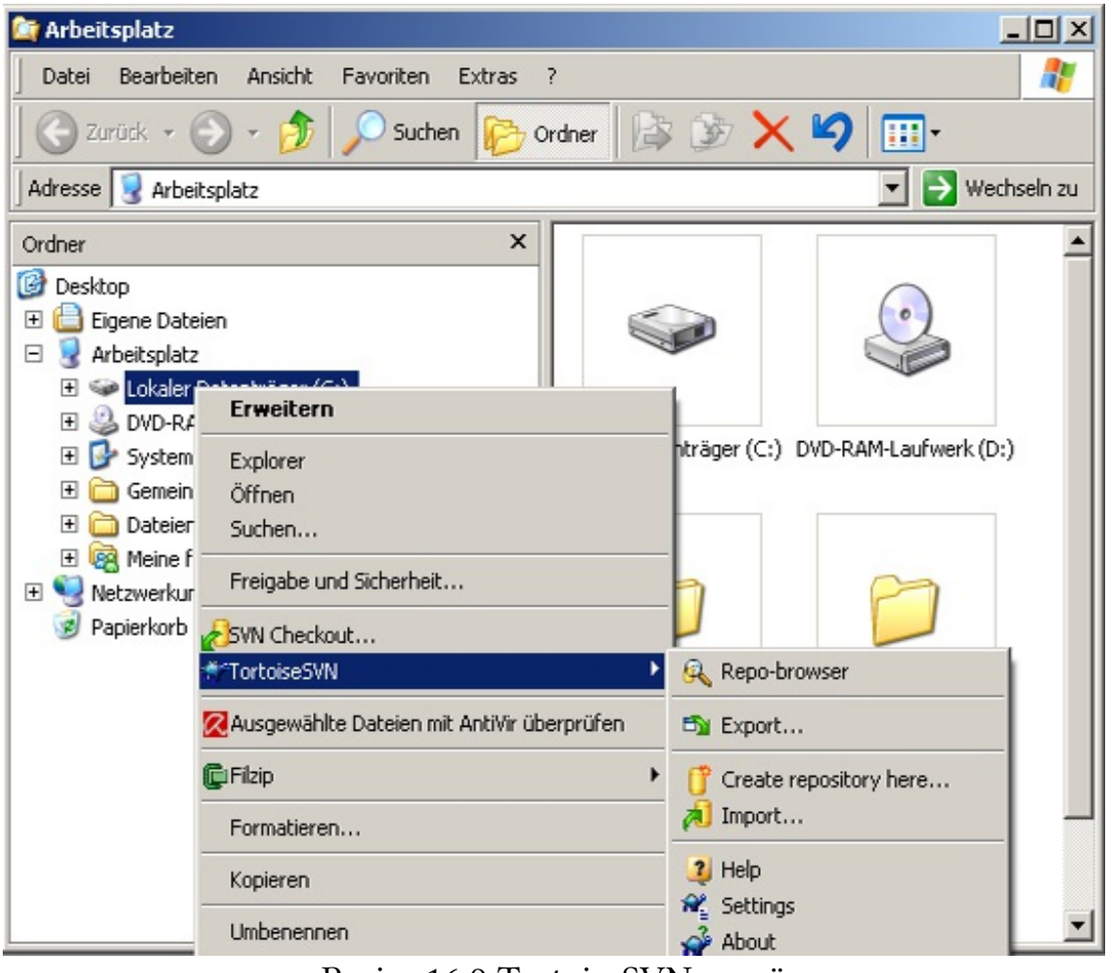
Subversion Client TortoiseSVN

Windows işletim sistemi altında Subversion ile çalışmayı kolaylaştırmak için grafiksel arayüze sahip olan TortoiseSVN programı geliştirilmiştir. Bu programı http://tortoisesvn.tigris.org/ adresinden temin edebilirsiniz. TortoiseSVN bir kullanıcı (client) program olarak düşünülebilir. Subversion sunucu modunda çalıştırıldığı taktirde TortoiseSVN ile sunucuya bağlantı kurularak, dokümanlar üzerinde işlem yapılabilir.



Resim 16.8 TortoiseSVN kurulumu

Kurulma işleminin ardından Windows Explorer içinde herhangi bir dizin üzerinde gidilerek sağ tuşa tıklandığı taktirde TortoiseSVN açılan menüde görünecektir.



Resim 16.9 TortoiseSVN menü

Subversion ile çalışırken hem bin dizininde bulunan svn komutu ile hem de TortoiseSVN gibi grafiksel bir client programı ile çalışabilirsiniz. Bu konudaki seçimi çalışma tarzınız uyumlu olacak şekilde yapabilirsiniz.

Repository (Depo)

Subversion bünyesinde tüm dokümanlar ve bu dokümanların değişik versiyonları repository ismini taşıyan bir depoda tutulur. Kurulum esnasında nasıl bir tip repository kullanılması gerektiği belirlenir. Subversion ın ilk sürümleri Berkeley DB veri tabanını repository olarak kullanırdı. Subversion ın yeni sürümlerinde FSFS ismini taşıyan ve dokümanları dosya sisteminde (filesystem) bir dosya içinde tutabilen alternatif bir repository türü bulunmaktadır. Bu iki repository türü arasında seçim yapılabilir. Hangi tür repository nin kullanılması gerektiği gereksinimler doğrultusunda belirlenmelidir.

Bir repository dokümanların yer aldığı bir dizin ağacı olarak düşünülebilir. Her

ağaç barındırdığı doküman ve dizinlerin belli bir zamanda yapılan değişiklikler sonucu oluşan versiyonlarını ihtiva eder. Bu versiyonlar kullanıcıların yaptığı işlemler sonrasında oluşurlar ve Subversion jargonunda revizyon (revision) olarak isimlendirilir.

Revizyon (Revision)

Revizyonun ne olduğunu bir örnek kullanarak açıklamak istiyorum. HelloWorld.java isminde bir doküman oluşturup, repository ye eklediğimizi düşünelim. Bu andan itibaren repository de 1 nolu revizyon oluşur. Bu revizyon numarası tüm repository genelinde geçerlidir. Aynı repository içinde birden fazla proje yer alabilir. Repository ye değişik projelerden değişik dokümanlar eklense bile genel revizyon numarası bir artırılacaktır.

Subversion her dokümanın yeni versiyonu için o dokümana bir versiyon numarası atamaz. Subversion için her değişiklik yeni bir revizyon oluşturulması anlamına gelir. Örneğin 1. revizyondaki HelloWorld.java isimli dokümanın versiyon numarası yoktur. HelloWorld.java yapılan değişiklerden sonra en son versiyonuyla 1 nolu revizyonda yerini alır.

```
Revizyon 1:
HelloWorld.java
```

Bir nolu revizyon içinde HelloWorld.java dokümanı yer almaktadır. Akabinde Test.java isminde ikinci bir doküman oluşturup, tekrar repository ye ekliyoruz. Bu işlemin ardından revizyon 2 oluşuyor.

```
Revizyon 2:
HelloWorld.java
Test.java
```

2 nolu revizyon bünyesinde HelloWorld.java ve Test.java dokümanlarını barındırmaktadır. HelloWorld.java üzerinde hiçbir değişiklik yapmamış olmamıza rağmen, Subversion otomatik olarak bu dokümanı da göz önünde bulundurarak yeni revizyonu oluşturdu. Bu andan itibaren 2 nolu revizyon söz konusu olduğunda, bu revizyonda yer alan iki dokümanın en son versiyonları karşımıza çıkacaktır.

Siparis.java ismini taşıyan bir dosya daha ekleyerek, 3. revizyonu oluşturuyoruz. Bu arada HelloWorld.java üzerinde bazı değişikler yaptık ve böylece bu

dokümanın yeni bir versiyonu oluşmuş oldu.

```
Revizyon 3:
HelloWorld.java
Test.java
Siparis.java
```

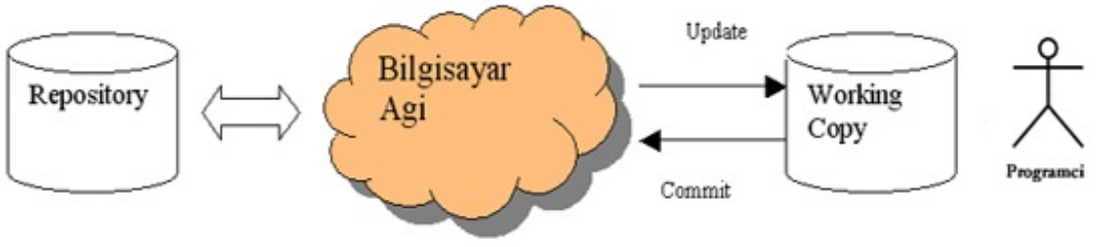
İlk bakışta 3 nolu revizyonda yer alan HelloWorld.java dokümanı üzerinde yapılan değişiklikleri görmemiz mümkün değildir, çünkü bu doküman bir versiyon numarası taşımamaktadır. Sadece en son versiyonun yer aldığı 3 nolu revizyonu görmekteyiz.

Subversion bir dokümanın tarihçesine (history) bakmamızı mümkün kılmaktadır. 3. revizyonda bulunan HelloWorld.java dokümanın tarihçesine bakarak, kim tarafından hangi değişikliklerin yapıldığını görmemiz mümkündür.

Birinci ve üçüncü revizyon kıyaslandığında Test.java ve Siparis.java dokümanlarının yeni eklendiğini ve HelloWorld.java dokümanı üzerinde değişiklikler yapıldığını tespit ederiz. Subversion repository içinde revizyon numaraları üzerinde her dokümanı takip ettiği için yapılan değişiklileri kolaylıkla gösterecektir.

Working Copy (Üzerinde Çalışılan Kopya)

Bir proje bünyesinde oluşan tüm dokümanlar Subversion tarafından oluşturulan repository içinde yer alır. Bir programcının kod üzerinde çalışabilmesi için repository de bulunan dokümanların bir kopyasına sahip olması gerekmektedir. Programcının sahip olduğu bu dokümanlara working copy adı verilir. Programcı yaptığı her değişikliği repository ye gönderebilir (commit) ve en son yenilikleri repository den alabilir (update).



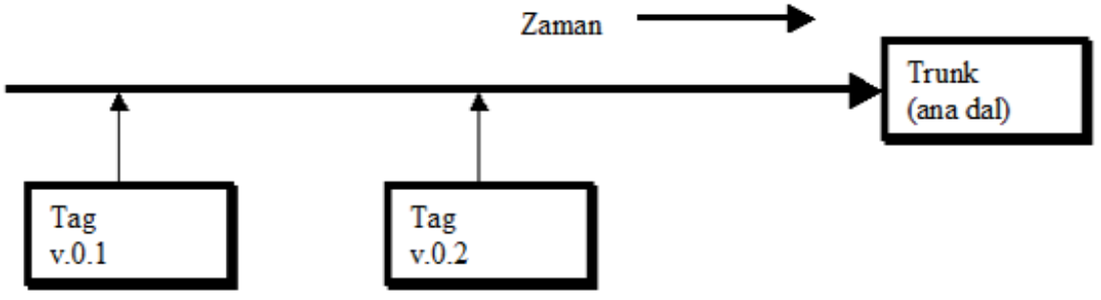
Resim 16.10 Working copy

Working copy oluşturma işlemine Subversion dilinde checkout (çıkış yapmak) ismi verilir. Programcı genelde ufak ve orta boylu projelerde projeye ait tüm dokümanların birer kopyasını kendi bilgisayarına almış olur. Proje eğer büyük bir yapıya sahipse, projenin alt modülleri ya da belirlenmiş kısımları working copy olmak üzere checkout yapılır.

Etiket Kullanımı

Subversion da değişik versiyonların revizyon numarası üzerinden yönetildiğini gördük. Sonuç itibariyle revizyon numarası bir rakam olduğu için akılda kalıcı olmayabilir. Programcılar arasında değişik program versiyonları için version1, release2 gibi isimlerin kullanılması daha yaygındır.

Subversion ile projenin belirli bir aşamasında o zamana kadar yapılmış tüm çalışmaları bir etiket altında toplamak amacıyla etiket (tag) konsepti uygulanabilir. Herhangi bir isim seçilerek projenin belirli bir aşamaya ulaştığını ifade etmek için etiket (tag) oluşturulur. Bu bir fotoğrafın çekilmesi gibi o anki anlık görüntünün alınması anlamına gelir.



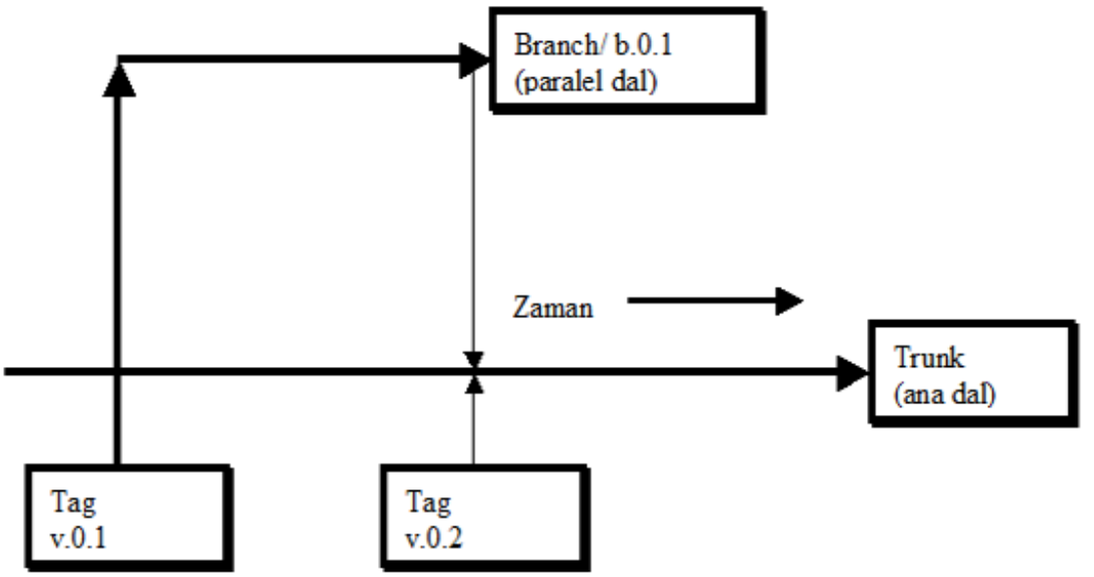
Resim 16.11 Tag konsepti

Etiket ismi kullanılarak projenin bu aşamadaki durumu daha sonra repository den edinilebilir. Tag kullanılarak oluşturulmuş versiyonlar repository den sadece okuma (readonly) amacıyla alınmalıdır. Yapılacak herhangi bir değişiklik yine aynı tag altında kayıtlandığı için etiket ile oluşturulan anlık görüntü bozulmuş olur.

Branch (Dal)

Bir iterasyon sonunda müşteriye yeni bir sürüm oluşturarak teslim ettiğimizi düşünelim. Müşteri bu sürümü kullanırken bizde yeni bir iterasyonda projeye

devam ediyor olalım. Sürüm için yeni bir versiyon etiketi kullandık, örneğin v.0.1. Müşteri programı kullanırken bir hatanın oluştuğunu bize bildirmiş olsun. Bu durumda üzerinde çalıştığımız ana kod üzerinde hatayı gidererek, müşteriye yeni bir sürüm veremeyiz, çünkü üzerinde çalıştığımız ana kod çalışır ve stabil durumda değil ya da yeni implementasyonlar tamamlanmadı. Bu durumda yapılabilecek tek şey: v.0.1 versiyonunu baz alarak, yeni bir dal oluşturmamız ve hatayı o dal üzerinde gidermemiz gerekiyor. Yeni bir dal oluşturduğumuz zaman ana koda paralel olarak başka bir versiyonu baz alan yeni bir yazılım kanadı oluşur.



Resim 16.12 Dal konsepti

Resim 16.12 de dal konsepti gösterilmektedir. Trunk x (zaman) ekseninde ilerleyen ana kod dalıdır. Oluşan hataları gidermek için trunk a paralel olarak yeni bir dal (branch) oluşturmamız gerekir. Hatalar bu dal üzerinde giderildikten ve yeni bir versiyon (v.0.2) oluşturduktan sonra paralel dal üzerinde yaptığımız değişiklikleri trunk a aktarırız (merge). Bu işlemin ardından hem ana dal rahatsız edilmeden hata giderilerek yeni bir sürüm oluşturulur hem de hataların giderilmesi için yapılan değişiklikler merge metoduyla ana dala aktarılır.

Trunk (Ana dal)

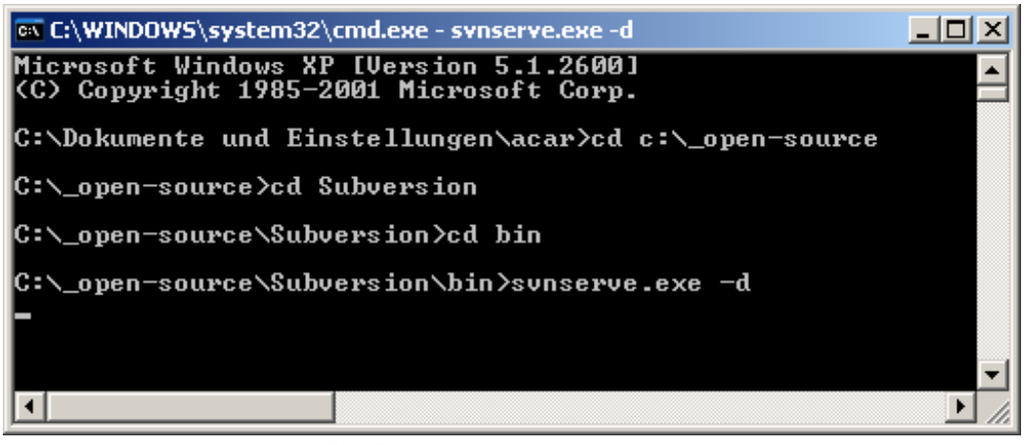
Trunk repository de bulunan ana kod dalıdır. Yapılan tüm değişiklikler trunk içinde yer alır. Tüm etiket ve dallar trunk üzerinden gerçekleştirilir.

Merge (Birleştirme)

Eğer belirli bir sürüm için hata gidermemiz gerekiyorsa, trunka paralel yeni bir dal (branch) oluşturmamız gerekiyor. Burada yapılan değişikliklerin tekrar trunka aktarılmasına merge adı verilir.

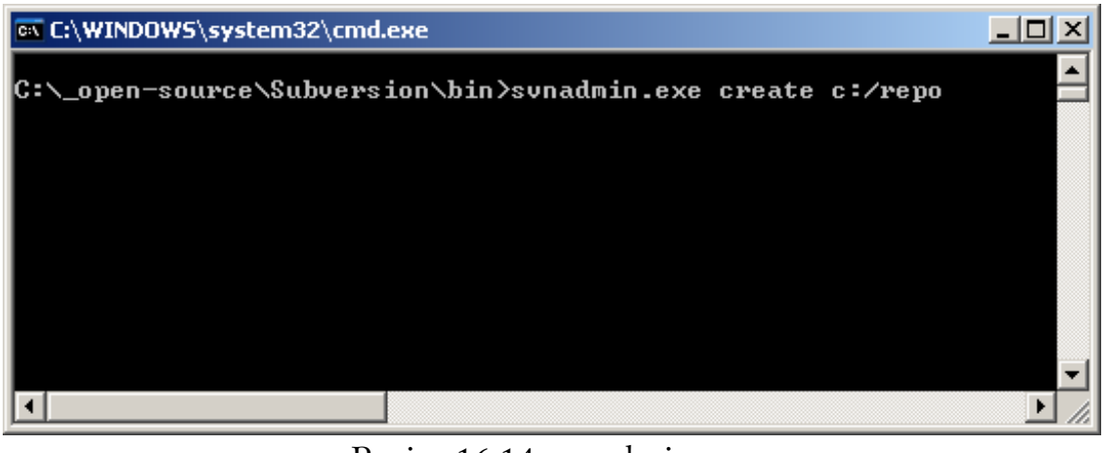
Subversion Sunucu

Programcılar arasında kod paylaşımını sağlayabilmek için Subversion sunucusu kurulması gerekmektedir. svnserve.exe programıyla Subversion ı sunucu modunda çalıştırabiliriz.



Resim 16.13 svnserve.exe

Resim 16.13 de görüldüğü gibi svnserve.exe –d (daemon) Subversion sunucusunu aktif hale getirir. Subversion sunucusunun aktif hale gelmesi daha önce bir repository oluşturulmamışsa bir anlam taşımaz. Bu sebepten dolayı ilk önce bir Subversion repository sinin oluşturulması gerekir.



Resim 16.14 svnadmin.exe

Resim 16.13 de svnadmin.exe programı kullanılarak c:\repo dizininde bir repository oluşturulmaktadır.

Resim 16.15 Repository dizin yapısı

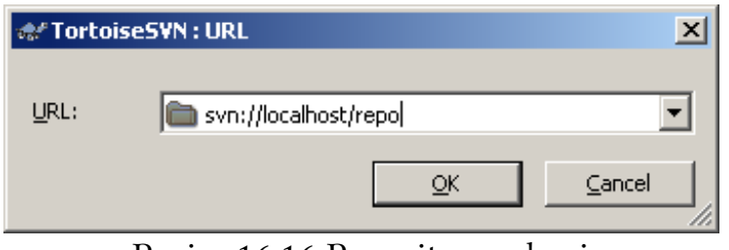
Subversion sunucu ayarları kurulan yeni repository nin conf dizininde bulunan svnserve.conf dosyası üzerinden yapılır.

```
Kod 15.1   svnserve.conf

[general]
password-db = userfile
realm=Shop-Projesi
auth-access = write
```

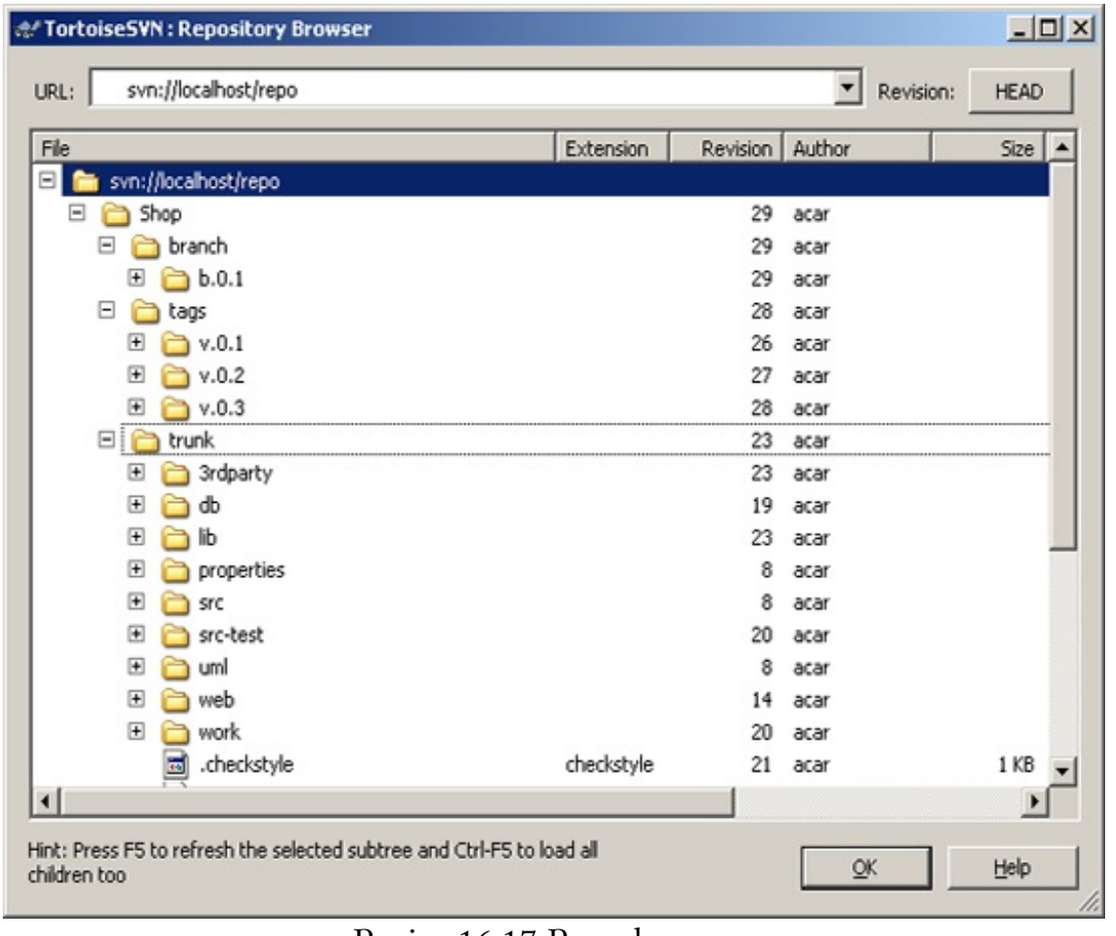
Bu dosyanın en basit hali kod 16.1 de yer almaktadır. password-db anahtarıyla kullanıcıların isim ve şifrelerinin yer aldığı bir dosya tanımlanır. Bu dosyanın içeriği kod 16.2 de yer almaktadır.

```
Kod 15.2   userfile

[users]
acar=acar
```

Bu işlemlerin ardından herhangi bir Subversion client programı ile repository ye bağlanabiliriz.



Resim 16.16 Repository adresi



Resim 16.17 Repo browser

Subversion Proje Dizin Yapısı

Resim 16.17 de genel proje dizin yapısı yer almaktadır. Her proje için aşağıdaki dizin yapısı tavsiye edilmektedir:

- Proje_ismi/trunk
- Proje_ismi/branch
- Proje_ismi/tags

trunk dizini içinde üzerinde aktif çalışılan kodlar yer alır. Programcılar değişikleri trunk üzerinden birbirleriyle paylaşırlar.

Branch dizininde değişik versiyonlar baz alınarak oluşturulmuş ve değişik versiyonlarda oluşan hataları gidermek için kullanılan trunk a paralel kod dalları yer alır. Örneğin v.0.1 de oluşan bir hatayı düzeltmek için b.0.1 isminde bir branch oluşturulur. Hata bu branch içinde giderildikten sonra yapılan değişikliler trunk ile merge edilir ve b.0.1 silinir.

Tags dizininde değişik zamanlarda etiket kullanılarak oluşturulan program

versiyonları yer alır. Bu versiyonlar müşteriye gönderilen sürümlerdir. Müşteri tarafından tespit edilen sürüm hataları tags dizininde yer alan bir versiyon baz alınarak oluşturulan branch içinde giderilir.

17. Bölüm

Proje Takibi

Giriş

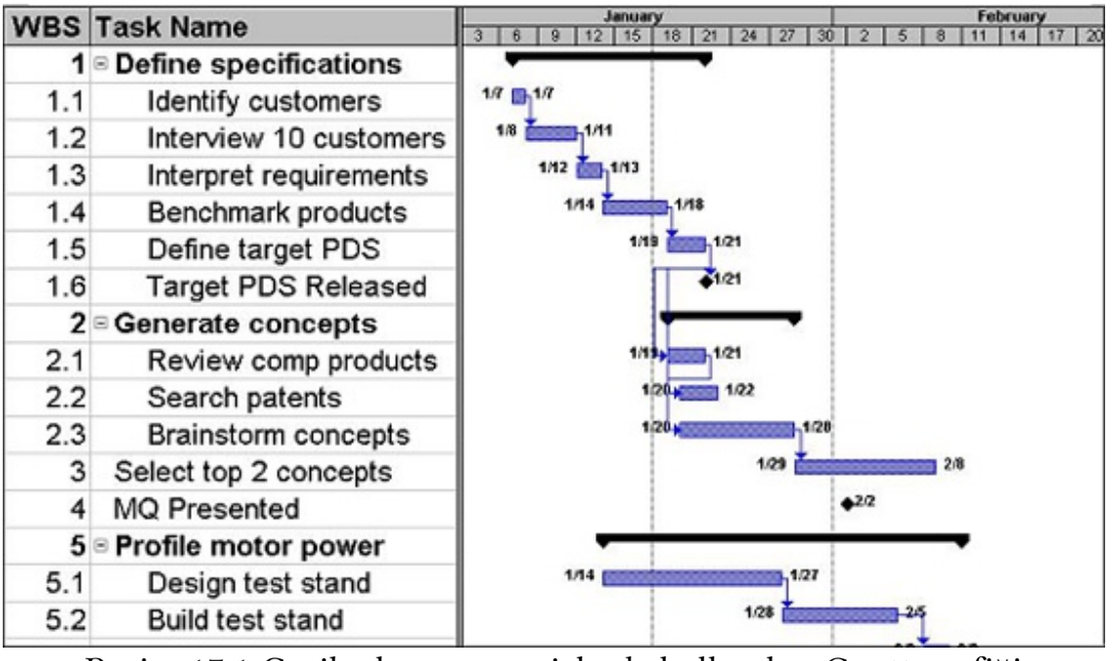
Proje gidişatını izleyebilmek ve gerekli durumlarda rota değişikliği yapabilmek için proje gidişatı hakkında bilgiye sahip olmak gerekmektedir. Sadece bu durumda doğru kararlar aracılığıyla proje kontrol altında tutulabilir.

Projeyi takip edebilmek için enformasyon radyatörü olarak bilinen metot ve araçlardan faydalanılır. Bu bölümde enformasyon radyatörlerinin ne olduklarını ve nasıl kullanıldıklarını yakından inceleyeceğiz.

Enformasyon radyatörlerinde Burndown grafikleri kullanılarak proje gidişatı görselleştirilir. Proje takibi sürüm ve iterasyon bazında yapılabilir. Sürüm ve iterasyon takip planlarında kullanılan Burndown grafikleri, içinde bulunulan iterasyon ve sürüm hakkında durumsal ve zamansal bilgi kazanılmasını kolaylaştırır. İterasyonlarda kullanılan Kanban boardlar programcılara tamamlanan ve başlanmamış görevler hakkında bilgi verir. Kanban boardlar ile ekibin hangi görevler üzerinde çalıştığını anlamak kolaydır.

Proje Takibi

Projedeki ilerlemeyi ya da tam tersi duraklamayı takip edebilmek için proje takip planları oluşturulur. Bu planlarda genelde iki ya da üç boyutlu grafikler (chart) kullanılarak görsellik sağlanır. Çevik olmayan projelerde lisans bedeli olan programlar kullanılarak proje takip planları oluşturulur. Bu planlarda görevler mevcut kaynaklara dağıtılarak ilerleme hesaplanır. Bu tür programların kullanımı proje yöneticilerine proje yönetim kurslarında öğretilir. Bu prensipte ve uygulamada araç kullanımını şart koştuğu ve ön plana çıkardığı için çevikliğe aykırı bir durumdur.



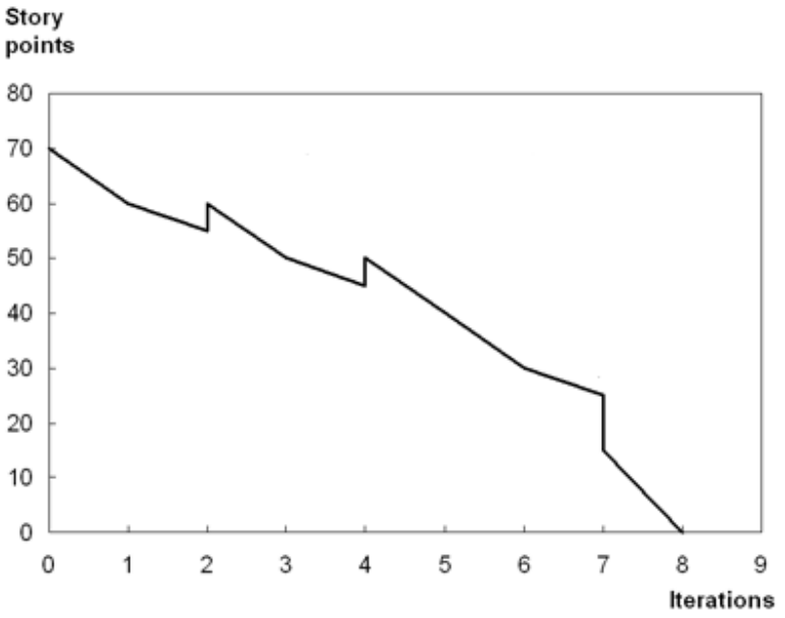
Resim 17.1 Çevik olmayan projelerde kullanılan Gantt grafiği

Çevik projelerde proje planlaması ve kaydedilen ilerlemeyi görselleştirmek için herhangi bir lisans bedeli olan program kullanılma zorunluluğu yoktur. Çok basit araçlar kullanılarak projede kaydedilen ilerleme takip edilebilir. Oluşturulan grafikler çok basit yapıda oldukları için anlaşılmaları da kolaydır.

Çevik projelerde ilerleme Burndown grafikleri kullanılarak görselleştirilir. Sürüm ve iterasyon için değişik türde Burndown grafikleri mevcuttur.

Burndown Grafikleri

Burndown grafiklerinde kullanıcı hikayeleri için tahmin edilen hikaye puanları (story point) ile kullanıcı hikayelerinin implementasyon zamanları ilişkilendirilir. Y koordinatında hikaye puanları, x koordinatında zaman birimleri yer alır. Bir sürüm ya da iterasyon başlangıcında belirli sayıda hikaye puanları vardır. Toplam hikaye puan adedi kullanıcı hikayeleri implemente edildikçe azalır.



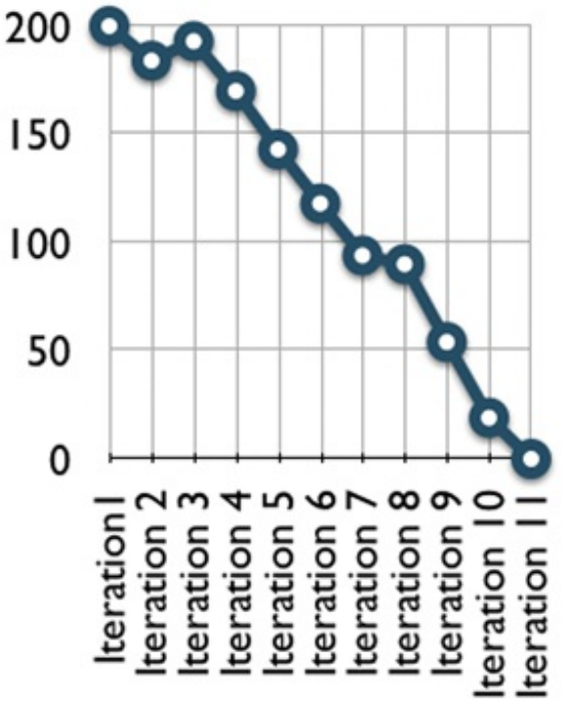
Resim 17.2 Burndown grafiği

X ve y eksenlerinin kesiştiği noktalar birleştirilerek, aşağıya doğru akan bir çizgi oluşturulur. İdeal şartlarda bu çizgi projenin x ekseni üzerinde bulunan bitiş tarihine ulaştığında geriye kalan hikaye puan adedi sıfır olmalıdır. Eğer durum böyle değilse, ekip bazı kullanıcı hikayelerini belirlenen zamanda implemente edememiş demektir.

Birbirine komşu iki x-y kesişme noktasındaki hikaye puanlarının farkı, o zaman birimi için çalışma ekibinin temposudur. Resim 17.2 de bu temponun sıfırıncı iterasyondan birinci iterasyona kadar on hikaye puanı olduğunu görüyoruz. Programcı ekip bir iterasyon süresinde on hikaye puanına eşit gelen kullanıcı hikayesini implemente etmiştir. Zaman birimi bir iterasyon olduğu için aynı zamanda bu değer ekibin çalışma hızıdır (velocity).

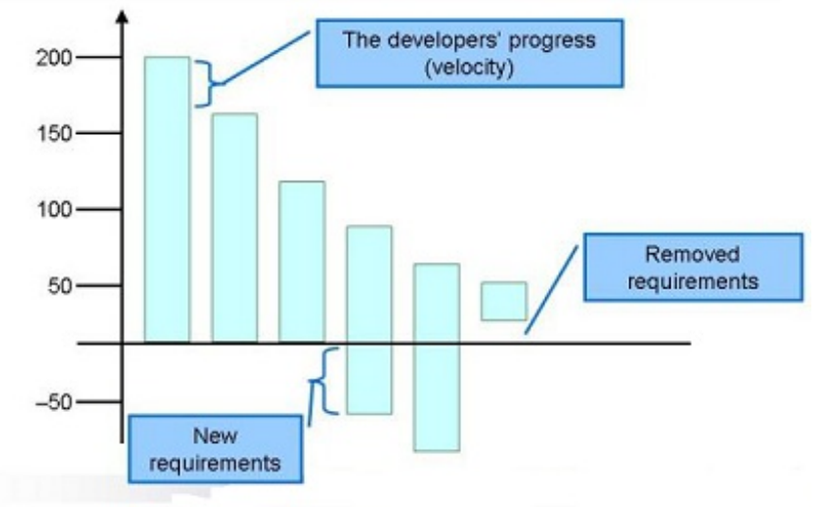
Sürüm Takibi

Sürümleri takip etmek için Burndown grafikleri kullanılabilir. Bunun bir örneğini resim 17.2 ve resim 17.3 de görmekteyiz. Sürüm grafiklerinde x-ekseni üzerinde sürüm içinde yer alan iterasyonlar zaman birimi olarak yer alır. Y ekseni hikaye puanlarını ihtiva eder.



Resim 17.3 Sürüm Burndown grafiği

Resim 17.3 de yer alan sürüm Burndown grafiğinde toplam 200 hikaye puanı 11 iterasyonla ilişkilendirilmiştir. Birinci iterasyon 200 puanla başlamış, bu rakam ikinci iterasyonda 170 civarına düşmüştür. Üçüncü iterasyonda bu rakamın 190 ın üzerine çıktığını görüyoruz. Müşteri ikinci iterasyonun sonunda yeni kullanıcı hikayeleri oluşturmuş ve böylece toplam hikaye puan adedini yükselmiştir. Burndown grafikleri her zaman aşağıya doğru göstermek zorunda değildir. Müşterinin yeni istekleri doğrultusunda grafik zaman zaman yukarı doğru (burnup) gösterebilir.



Resim 17.4 Sürüm Burndown bar grafiği

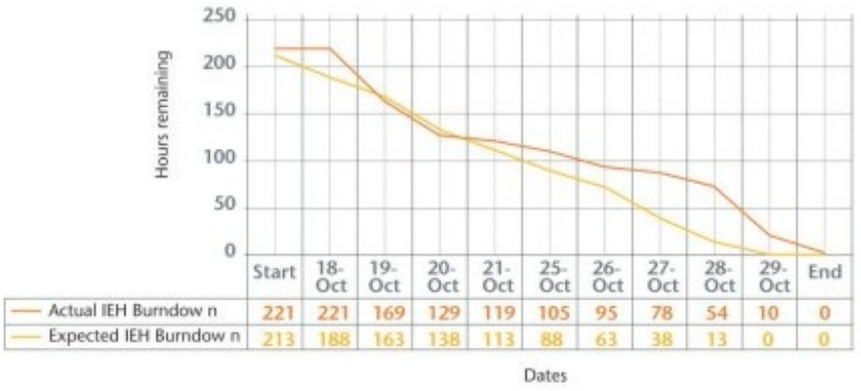
Müşteri tarafından yapılan eklemeleri Burndown grafiklerinde çizginin

yukarıya doğru göstermesi olarak görmüştük. Burndown grafikleri müşteri tarafından yapılan eklemeleri ve çıkartmaları görsel olarak yansıtmada zorlanır. Bu tür değişiklikleri takip etmek için Burndown Bar grafikleri oluşturulur. Bunun bir örneğini resim 17.4 de görmekteyiz.

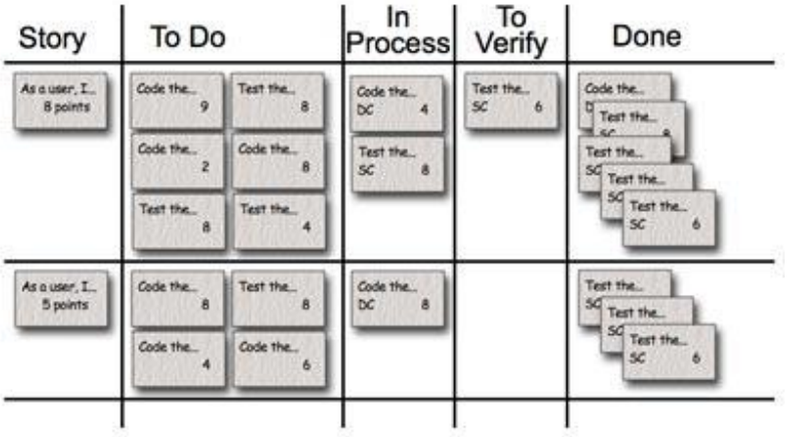
Burndown Bar grafiklerinde her iterasyon dikdörtgen bir bar olarak çizilir. Bu bar x ekseninin altına düşerse, bu müşteri tarafından yeni kullanıcı hikayelerinin iterasyona dahil edildiği anlamına gelir. Eğer bar x eksenine değmeyecek şekilde bu eksene mesafeli kalırsa bu müşteri tarafından bazı kullanıcı hikayelerinin iterasyondan alındığı anlamına gelir. Resim 17.4 de ilk iterasyon 200 hikaye puanı ile start almıştır. İkinci iterasyonda bu değer 170 lere iner. Bu durumda ekibin hızı (velocity) 30 hikaye puanıdır. Dördüncü iterasyonda 50 yeni hikaye puanına denk gelen kullanıcı hikayesi iterasyona eklemiştir. Altıncı iterasyonda 15 civarı hikaye puanı eklenmiştir. Son iterasyonda geri kalan 50 hikaye puanından 15 civarı sistemden çıkartılmıştır.

İterasyon Takibi

Burndown grafikleri iterasyon performansını ölçmek için de kullanılır. İterasyon Burndown grafiklerinde x ekseninde iterasyon günleri yer alır. Örneğin iterasyon süresi bir hafta seçilmişse, x ekseni yedi günlük zaman biriminde oluşur. Bunun bir örneğini resim 17.5 de görmekteyiz. İterasyon Burndown grafiklerinde ölçümler günlük yapılır.



Resim 17.5 Iterasyon Burndown grafiği



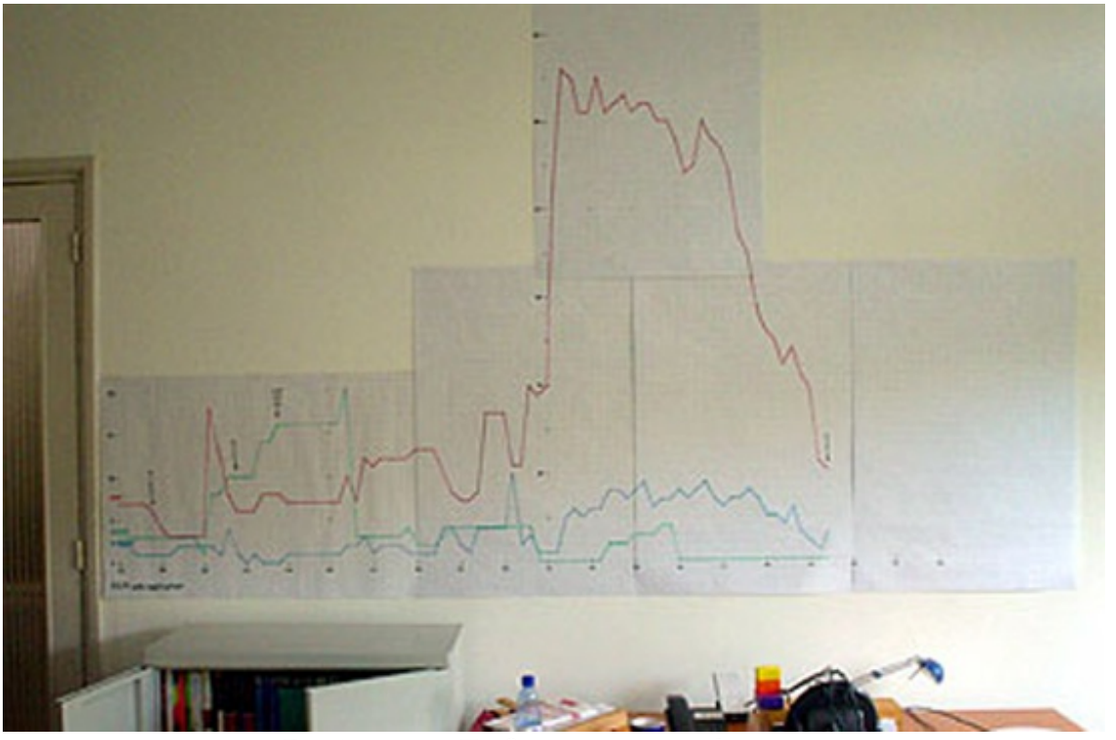
Resim 17.6 İterasyon görev panosu

İterasyonda ilerlemeyi takip edebilmek için kullanılan diğer bir araçta görev panosudur. Her iterasyon öncesinde kullanıcı hikayeleri implemente etmek için gerekli görev kartları (task card) oluşturulur. Bu görev kartları yapılması gerekenler (To Do), yapılanlar (In Progres), kontrol edilmesi gerekenler (To Verify) ve yapılmış olanlar (Done) şeklinde gruplanır. Bu şekilde bir kullanıcı hikayesinin ne oranda implemente edildiği takip edilebilir.

Enformasyon Radyatörleri

Evimizde kışın ısınmak için kalorifer radyatörleri kullanılırız. Radyatörler ısının evin içine doğru yayılmasını sağlarlar. Bu şekilde enformasyonları odanın içine ısı gibi yayan araç ve yöntemlere enformasyon radyatörü adı verilir. İlk kez Alistair Cockburn tarafından Agile Software Development (The Cooperative Game) isimli kitabında bu şekilde isimlendirilen araç ve metotlar yardımıyla çalışma odasında bulunan herkes proje gidişatı hakkında hemen gerekli bilgiyi sahip olabilmektedirler.

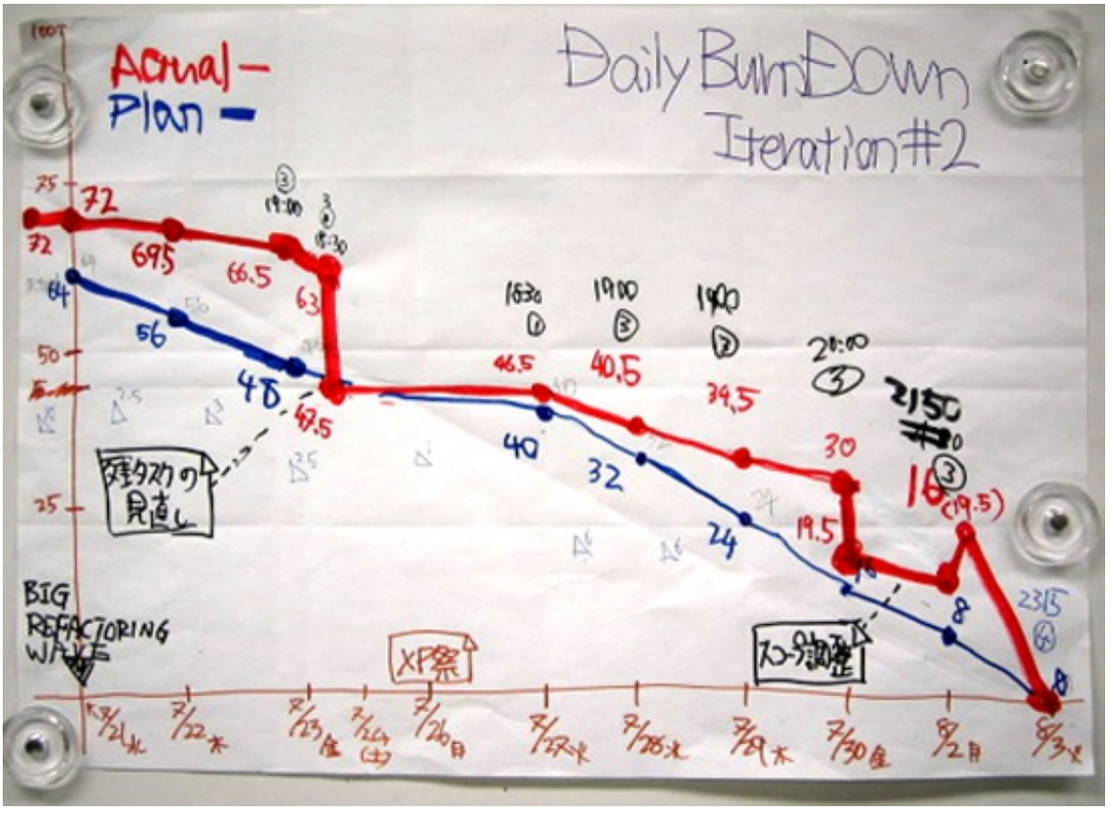
Enformasyon radyatörleri olarak çalışma odalarının duvarlarına asılan büyük panolar kullanılır. Bu panolarda proje hakkında detaylı bilgiler yer alır



Resim 17.7 Enformasyon radyatörü

Enformasyon radyatörlerinin kullanımı avantajlıdır, çünkü proje çalışanlarına projenin statüsü, yapılması gerekenler, tamamlanan iş ve proje metrikleri gibi konularda bilgi aktarırlar. Ayrıca enformasyon radyatörleri programcı ekibe tamamlanmamış ve yapılması gereken görevleri hatırlatırlar. Bu proje hakkında genel bir görüntü sahibi olabilmek için önemlidir.

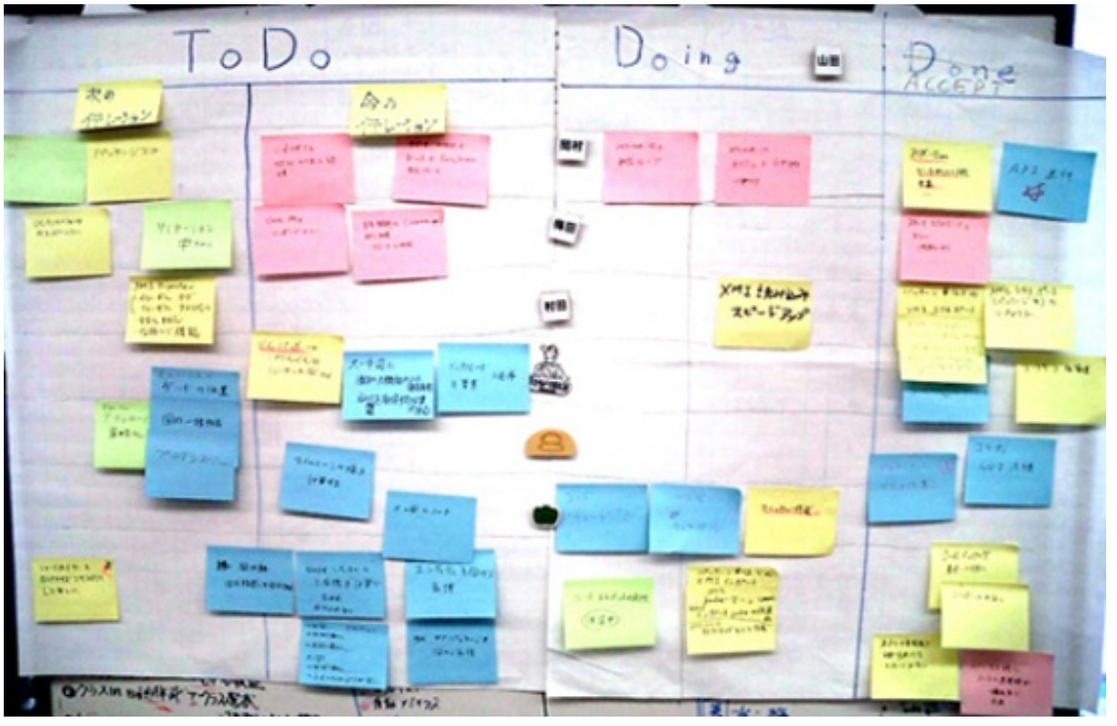
Resim 17.8 Enformasyon radyatörü



Resim 17.9 Enformasyon radyatörü

Kanban Board

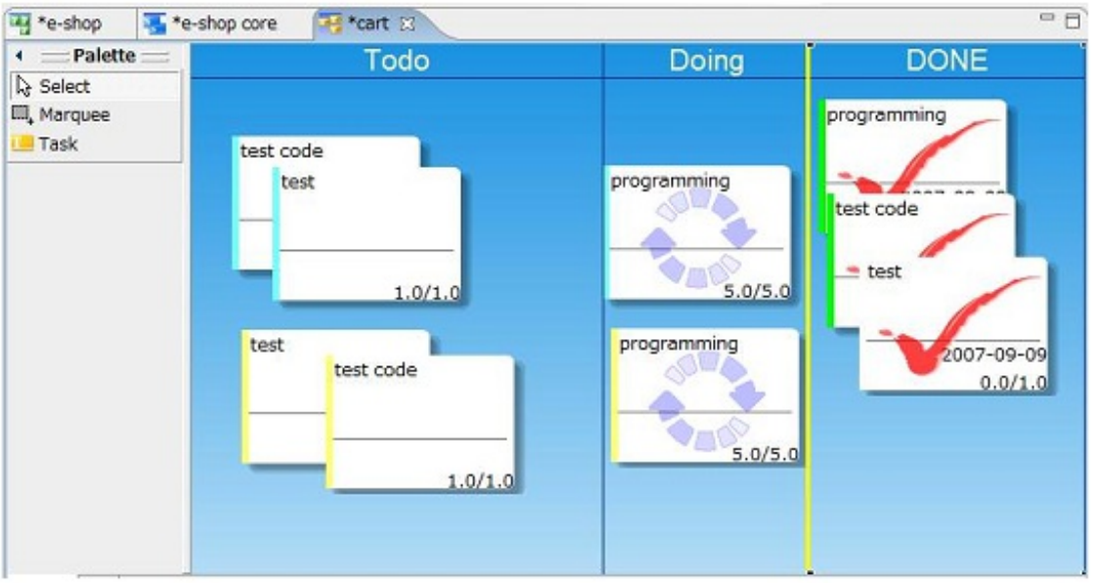
Kanban görev kartı (task card) anlamına gelmektedir. Bir iterasyon içinde implemente edilmesi gereken görevler Kanban boardlarda sahip oldukları statüye göre yer alırlar. Bunun bir örneğini resim 17.10 da görüyoruz. Kanban boardlar ilk olarak JIT (Just In Time) prodüksiyon için TPS (Toyota Production System) tarafından kullanılmıştır ve daha sonra çevik süreçlerde proje takip araçları olarak yaygın kullanılır hale gelmiştir. Resim 17.10 da bu iterasyon bünyesinde implemente edilecek tüm görevler yer almaktadır. Post-It Notes kullanılarak oluşturulan görevler üç değişik alanda gruplamıştır (To Do, Doing, Done).



Resim 17.10 Kanban board

Kanban boardları programcıların nasıl çalıştıklarını anlamaları ve bir sonraki görevin ne olduğunu bilmeleri açısından önemli statü bilgileri ihtiva etmektedir. Kanban boardlar enformasyon radyatörüdür.

Elektronik ortamda Trichord programıyla Kanban boardlar ve sürüm/iterasyon takip planları oluşturulabilir.



Resim 17.11 Elektronik Kanban board

18. Bölüm

XP Hakkında Sorular ve Cevapları

XP Hakkında Sorular

Bu bölümde araştırma yaparken çeşitli kaynaklarda rastladığım ve XP yi öğrenirken kafamda oluşan sorular ve cevapları yer almaktadır. Bu soruların cevabı kitabın değişik bölümlerinde verildi. Ama tekrarlamış olmak adına bu soru kataloğunu sizlere kitabın en son bölümü olarak sunmak istiyorum. Sorular herhangi bir sıraya sahip değildir.

Kullanıcı hikayesi (user story) nedir?

XP projelerinde müşteri gereksinimlerinin yer aldığı kullanıcı hikayeleri oluşturulur. Bir kullanıcı hikayesi sistemin tipik bir özelliğini bir ya da iki cümle ile anlatan araçtır. Örneğin üye girişi olan bir sistemde şöyle bir kullanıcı hikayesi düşünülebilir:

Kullanıcı isim ve şifreni kullanarak sisteme giriş yapar.

Kullanıcı hikayeleri hikaye kartlarına (story card) yazılır. Bu kartlar sürüm ve implementasyon planları yapılırken kullanılır. Her kullanıcı hikayesinin implementasyon zamanı programcılar tarafından tahmin edilir. Müşteri kullanıcı hikayelerine öncelik sırası vererek implementasyon sırasını tayin eder. Programcılar tarafından yapılan tahmin ve müşteri tarafından belirlenen öncelik sırası kullanıcı hikayesinin üzerinde yer aldığı hikaye kartlarına not edilir. Ayrıca müşteri tarafından oluşturulan onay/kabul testleri hikaye kartlarının arka bölümüne not edilir.

Bir kullanıcı hikayesinin büyüklüğü ne kadar olmalı?

Bu sorudaki büyüklük sıfatı ile kullanıcı hikayesinin kaç günde implemente edilebilir olduğu kastedilmektedir. Programcılar tarafından kullanıcı hikayelerin implementasyon süreleri gün bazında tahmin edilir. Bir kullanıcı hikayesinin en fazla dört ya da beş günde implemente edilebilir yapıda olması gerekmektedir. Daha fazla zaman gerektiren kullanıcı hikayelerinin müşteri tarafından bölünerek, küçültülmeleri gerekmektedir.

Kullanıcı hikayelerini kim oluşturur?

Kullanıcı hikayelerini müşteri oluşturur, çünkü gereksinimleri en iyi bilen müşteridir.

XP projelerinde müşterin programcılarla beraber çalışması

talep edilir. Müşteri kendi işini bırakıp nasıl proje için çalışabilir? Başka işi yok mu?

Bu genelde müşterinin kendi işini gücünü bırakıp, projede başka işlerle uğraşması gerektiği şeklinde değerlendirilir, ama durum öyle değildir. Müşterinin programcılara yakın bir yerde olması, programcıların oluşan sorulara kısa sürede müşteri yardımıyla cevap bulmalarını kolaylaştırır. Asıl maksatta budur zaten. Müşteri gün boyunca programcıların sorularına cevap verir. Bunun yanı sıra kendi günlük işlerini takip eder. Çoğu zaman günlük işleri yapabilmek için bir bilgisayar yeterli olacaktır. Müşteri kendi işlerini yaparken ara sıra programcılara zaman ayırarak, soruları cevaplar.

Birden fazla müşteri varsa, hangisinin sözü geçerlidir?

Proje ekibinin karşısında sadece bir müşteri olmalıdır. Eğer birden fazla müşteri varsa, bu şahıslar bir araya gelerek, tek bir şahıs gibi programcı ekibi ile iletişim kurmalıdırlar.

Sürüm planını kim oluşturur?

Sürüm planı müşteri ve programcılar tarafından ortaklaşa oluşturulur. Ne ve hangi sıraya göre yapılması gerektiği müşteri tarafından belirlenir. Bu yüzden sürüm planının oluşumunda müşterinin ağırlığı daha fazladır. Planlama için zaman tahminleri programcılar tarafından yapılır. Bu şekilde programcılar proje planlama sürecine aktif olarak katılarak sorumluluk alırlar.

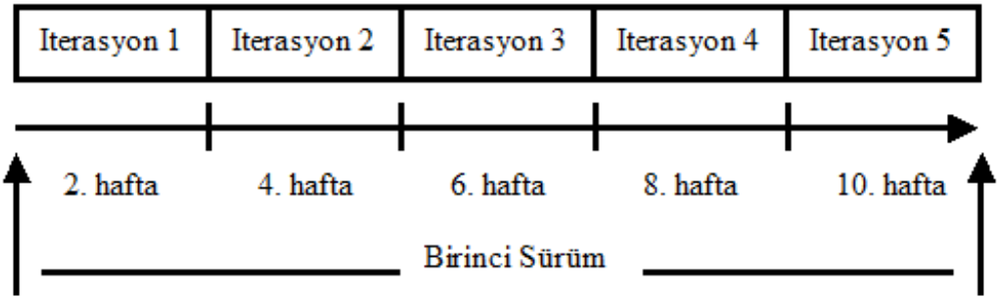
Sürüm planını ne oranda sabittir?

Proje başında oluşturulan sürüm planı projenin çeşitli safhalarında değişikliğe uğrayabilir. Bu doğaldır. Müşteri yazılım sisteminin ilk sürümleriyle gereksinimlerinin ne olduğunu daha iyi anlayabilir ve mevcut kullanıcı hikayeleri üzerinde değişilik yapılmasını talep edebilir ya da yeni kullanıcı hikayeleri oluşturabilir. Bu durumda sürüm planının değiştirilmesi gerekmektedir.

Sürüm ve iterasyon arasındaki fark nedir?

Sürüm yazılım sisteminin belirli bir versiyondaki halidir. Her yeni sürüm yeni özelliklerin implemente edildiği ve müşteri tarafından kullanılan program versiyonudur. XP projelerinde her yeni sürüm bir ile dört ay süren bir çalışma sonunda oluşturulur. Müşteri tarafından seçilen kullanıcı hikayeleri belirli bir zamansal uzunluğa sahip olan iterasyonlarda implemente edilir. İterasyonlar

bir ile dört haftalık zaman birimini kapsarlar.



Resim 18.1 Sürüm ve iterasyon

Resim 18.1 de yer alan sürüm beş iterasyondan oluşmaktadır. Her iterasyon iki hafta sürmektedir. Sürüm onuncu haftanın sonunda oluşturulmaktadır.

Hangi kullanıcı hikayesiyle işe başlanır?

Bu sorunun cevabını sürüm ve iterasyon planı verir. Sürüm ve iterasyon planlarında müşteri tarafından belirlenen kullanıcı hikayeleri yer alır. Müşteri kullanıcı hikayeleri için sahip oldukları değere göre öncelik sırası belirler. Sürüm ve iterasyon planlarında kullanıcı hikayeleri bu öncelik sırasına göre yer alırlar. Programcılar her iterasyonda, o iterasyon için seçilmiş olan kullanıcı hikayelerini öncelik sırası yüksekten düşüğe doğru implemente ederler. Buradaki ana amaç, müşteri için en değerli özelliklerin yer aldığı bir sürümü oluşturarak, müşteri tarafından kullanılabilir hale getirmektir. Bu yüzden her zaman müşteri açısından en değerli olan sistem özellikleri öncelikli olarak implemente edilir.

Kullanıcı hikayesi implementasyonu ne zaman tamamlanmıştır?

Programcılar bir kullanıcı hikayesini test güdümlü implemente ederler. Testler tamamlandıktan sonra onay/kabul testleri yapılmak üzere kullanıcı hikayesinin yer aldığı hikaye kartı (story card) testçiye (tester) devredilir. Testçi müşteri tarafından tanımlanmış olan onay/kabul testlerini implemente eder. Onay/kabul testlerini geçen bir implementasyon bitmiş olarak kabul edilir.

Mevcut projeler üzerinde XP uygulanabilir mi?

Bu büyük ölçüde projedeki birim testleri oluşturma alışkanlığına bağlı. XP test güdümlü implementasyonu şart koşmaktadır. Eğer programcılar tarafından programlara paralel olarak testler geliştiriliyorsa, test güdümlü implementasyona geçmeleri zor olmayacaktır. Bunun yanı sıra projedeki

mevcut iletişim kültürü önemlidir. XP çok yönlü iletişimi gerekli kılmaktadır. Örneğin pair programming gibi tamamen iletişim ve takım işine bağımlı olan bir metot programcılar tarafından ne oranda uygulanabilir, bunun araştırılması gerekmektedir.

Sürekli entegrasyon, test güdümlü yazılım, müşterinin projeye dahil edilmesi, kısa sürelerde yeni sürüm oluşturulması gibi konular XP yi yeni başlamayan projeler için zor adapte edilebilir kılmaktadır. XP nin yeni projelerde adaptasyonu çok daha kolaydır.

Bir iterasyon süresi ne kadar olmalı?

Bu yazılım sisteminin sahip olması gereken özelliklerle doğru orantılıdır. Eğer iki ay içinde ilk sürüm oluşturulması planlanıyorsa, iterasyon süresi bir ile iki hafta olacak şekilde seçilebilir. Bunun yanı sıra müşteri tarafından oluşturulan kullanıcı hikayelerinin implementasyonu için programcılar tarafından verilen tahminlerin dikkate alınması gerekmektedir. Örneğin programcılar tarafından ortalama her kullanıcı hikayesi için bir ile iki gün tahmin edilmişse, bir haftalık iterasyonda en az üç en çok beş kullanıcı hikayesi implemente edilebilir. Eğer kullanıcı hikayeleri ortalama dört veya üzeri günde implemente edilebilir durumda ise, o zaman interasyonun en az iki hafta olarak seçilmesi gerekmektedir. İterasyon süresi sabittir ve uzatılamaz, bu yüzden seçilen kullanıcı hikayelerinin seçilen sürede implemente edilebilir yapıda olmaları gerekmektedir.

Onay/kabul testlerini kim oluşturur?

Onay/kabul testleri kullanıcı hikayesini oluşturan müşteri tarafından tanımlanır. Onay/kabul testlerinin implementasyonunu programcılar ya da testçiler üstlenir.

Kaç adet birim testi hazırlanmalı?

Kullanılan her sınıf için bir birim testi sınıfı oluşturulması gerekmektedir. Bu sınıf bünyesinde birden fazla test metodu yer alır. Sınıfta bulunan her metodun test edilmesi gerekmektedir. Çoğu zaman oluşturulan test sınıfının büyüklüğü test edilen sınıfın 2-3 katı büyüklüğe sahiptir. Bu test adedi hakkında bir fikir sahibi olmanızı kolaylaştırır.

Kod paylaşımı nasıl yapılır?

Kod paylaşımını kolaylaştırmak için bir versiyon kontrol sisteminin

kullanılması şarttır. Subversion ya da Git son zamanlarda kullanılan en popüler açık kaynaklı versiyon kontrol sistemleri haline gelmişlerdir.

Pair programming yaparken tecrübeli bir programcı ile tecrübesiz bir programcının beraber çalışması zaman ve kaynak kaybı değil midir?

Hayır! Pair programming tekniği ile programcıların teknik anlamda aynı seviyeye gelmeleri sağlanır. Tecrübeli programcılar can, tecrübesiz programcılar patlıcan değildir. Tecrübesiz programcılar için seminer düzenlemek yerine, onlara pair programming seanslarında teknoloji ve proje hakkında bilgi transfer etmek daha mantıklıdır.

Pair programming iki programcının bir kişilik iş çıkarması anlamına gelmez mi?

Pair programming maliyetli bir yöntemdir. Lakin iki programcının beraber aynı implementasyon üzerinde çalışmasından sinerjiler doğar. Pair programming iş kalitesini yükseltir. Ayrıca pair programming ile kodun ve tasarımın iki değil dört göz ile kontrol edilmesini sağlanır.

XP projelerinde mimariyi ve tasarım nasıl oluşur?

Mimari (altyapı) proje öncesinde yapılan keşif safhasında (Exploration Phase) oluşur. Programcılar müşteri tarafından oluşturulan kullanıcı hikayelerini okudukça, neye ihtiyaç duyduklarını anlarlar ve ona göre altyapıyı geliştirirler.

Proje öncesi detaylı tasarım oluşturulmaz. Tasarım test güdümlü implementasyon esnasında programcılar tarafından oluşturulur. Eğer programcılar implementasyon esnasında sorunlarla karşılaşırlarsa, refactoring yöntemleri kullanarak tasarım üzerinde değişiklik yaparlar. Unit testleri refactoring işleminin yapılmasını kolaylaştırır. Yapılması gereken değişiklikler sonraya bırakılmaz, çünkü bu ilerde maliyetin yükselmesine sebep olabilir.

XP projelerinde mimariyi ve tasarımı kim oluşturur?

Programcılar.

Veri tabanı olan bir sistemde test güdümlü yazılım nasıl uygulanır?

Bunun çeşitli yöntemleri vardır. Öncelikle veri tabanı işlemlerinin DAO (Dao Access Object) tasarım şablonu kullanılarak bir interface sınıf arkasında

saklanması en mantıklı çözümdür. Mock sınıflar kullanılarak DAO katmanı simüle edilebilir. Bu sistemin diğer bölümlerinin test güdümlü implementasyonunu kolaylaştırır. DAO kullanıldığı taktirde gerçek veri tabanına olan bağımlılık azaltılır. DAO interface sınıfını değişik türde implemente ederek veri tabanı yerine başka bir yapıda kullanılabilir.

Hibernate ve IBatis gibi bir çatıların kullanılması durumunda test güdümlü yazılımı mümkün kılabilmek için bu çatıların sunduğu interface sınıflar (SessionFactory, Session vs.) mock nesneler ile simüle edilebilir.

Son Söz

8 aylık bir çalışmanın ardından elinizde tuttuğunuz bu kitap oluştu. Çalışma hayatıma paralel olarak hazırladığım bu kitapta edindiğim bilgi ve tecrübeleri sizinle paylaşmak istedim. Umarım yer yer çok karmaşık olabilen bu konuyu sade ve anlaşılır bir dille size aktarabilmişimdir.

Çevik Yazılım ve Extreme Programming özellikle Kuzey Amerika ve Avrupa'da yazılım yöntemlerinin tamamen değiştirilmesine sebep olan çağımızın en önemli yazılım akımlarıdır. Artık yazılım müşteri ve onun isteklerine odaklı bir şekilde yapılmaktadır. Teknik bilgilerine sevdalı, müşterin ne istediğini tam anlayamamış ya da ilerleyen zamanlarda müşteriden gelecek yeni değişiklik taleplerine kapalı bir ekibin ortaya koyacağı yazılım sistemi, müşterinin gereksinimlerini tatmin etmekten ışık yılı uzakta olacaktır. Eğer amaç müşteriyi memnun etmekse, o zaman yazılımcı olarak ona kulak vermemiz ve her zaman değişiklikle yaşamayı öğrenmemiz gerekiyor. Extreme Programming bizi bu süreçte sahip olduğu değer, prensip ve tekniklerle destekleyecek yapıdadır. Extreme Programming damgasını taşıyan ürünler diğerlerine nazaran daha dayanıklı, değişikliğe ve yeniliğe açık yapıdadır. Bu yüzden ülkemizde de Extreme Programming in yaygınlaşması adına bu yeni metotların projelerde uygulanması çok önemlidir.

Zaman bulup, bu kitabı okuduğunuz için teşekkür ederim. Umarım sizde edindiğiniz bilgileri başkalarıyla paylaşma fırsatı bulursunuz, çünkü bu kelimelerle ifade edilmesi zor bir hazdır.

Son bir rica: Üzerinde yaşadığımız dünyamız bizim çok kahrımızı çekmektedir. Lütfen ona saygı duyalım ve daha fazla incitmeyelim. Doğayı kirletmeyelim ve bize sunulan kaynakları sadece gerektiği kadar kullanalım. Bizden sonra gelecek

nesillere temiz bir dünya bırakmamız gerekiyor, bu bizim onlara olan borcumuz!

Kendinize iyi bakın. Aydınlık yarınlar umuduyla.....

Saygılarımla

Özcan Acar, Mayıs 2014

Kaynaklar

Tavsiye Edilen ve Kitapta Kaynak Olarak Kullanılan Literatür Listesi

- Robert C. Martin. Agile Software Development. Prentice Hall 2002
- Robert C. Martin. UML for Java Programmers. Prentice Hall 2003
- Anil Hemrajani. Agile Java Development. Sams Publishing 2006
- Scott W. Ambler. Agile Modeling. Wiley 2002
- David Astels. Test-driven development. Prentice Hall 2003
- Ken Schwaber. The Enterprise And Scrum. Microsoft Press 2007
- Mike Cohn. Agile Estimating and Planning. Printece Hall 2006
- James Shore. The Art of Agile Development. O'Reilly 2007
- The ThoughtWorks Anthology. Pragmatic Programmers 2008
- Lasse Koskela. Test Driven. Manning 2008-09-13
- Andrew Hunt. Pragmatic Unit Testing. Pragmatic Programmers 2003
- Mike Cohn. User Stories Applied. Addison Wesley 2004
- Esther Derby. Agile Retrospectives. Pragmatic Programmers 2006
- Alistair Cockburn. Agile Software Development. Addison Wesley 2007
- Jutta Eckstein. Agile Software Development in the Large. Dorset 2004
- Kent Beck. Extreme Programming Explained. Addison Wesley 2005
- Kent Beck. Planning Extreme Programming. Addison Wesley 2001
- Lisa Crispin. Testing Extreme Programming. Addison Wesley 2003
- William C. Wake. Extreme Programming Explored. Addison Wesley 2002
- Ken Auer. Extreme Programming Applied. Addison Wesley 2002
- Ron Jeffries. Extreme Programming Installed. Addison Wesley 2001
- Kent Beck. Test Driven Development. Addison Wesley 2003
- William C. Wake. Refactoring Workbook. Addison Wesley 2004
- Özcan Acar. Java Tasarim Sablonlari ve Yazilim Mimarileri. Pusula 2008
- Peter Schuh. Integration Agile Development in the Real World. Charles River 2005
- Jim Highsmith. Agile Project Management. Addison Wesley 2004
- Christian Bauer. Java Persistence with Hibernate. Manning 2007
- Martin Fowler. Refactoring. Addison Wesley 2005
- Sanjiv Augustine. Managing Agile Projects.Prentice Hall 2005
- Jennifer Stapleton. Business Focused Development. Addison Wesley 2003
- Venkat Subramaniam. Practices of an Agile Developer. Pragmatic Programmers 2007
- Ken Pugh. Prefactoring. O'Reilly 2005
- Scott W. Ambler. The Object Primer.Cambridge 2004
- Barbara Liskov. Program Development in Java. Addison Wesley 2001
- Scott W. Ambler. Agile Database Techniques. Wiley 2003

- Kirk Knoernschild. Java Design. Addison Wesley 2002
- James Newkirk. Extreme Programming in Practice. Addison Wesley 2001
- Dogu Wallace. Extreme Programming for Web Projects. Addison Wesley 2002
- Vincent Massol. JUnit in Action. Manning 2004
- J. B. Rainsberger. JUnit Recipes. Manning 2005
- Chris Richardson. POJO's in Action.Manning 2006
- Jeffrey Machols. Subversion in Action. Manning 2005
- Eric Freeman. Head First Design Patterns. O'Reilly 2004
- Erich Gamma. Design Patterns. Addison Wesley 1995
- Steven John Metsker. Design Patterns in Java. Addison Wesley 2006
- Mary Poppendieck. Lean Software Development. Addison Wesley 2003
- Eric Burke. Java Extreme Programming Cockbook. O'Reilly 2003
- Robert. C. Martin. Clean Code. Prentice 2008

Tavsiye Edilen Online Kaynaklar

- http://www.extremeprogramming.org
- http://www.xprogramming.com
- http://objectmentor.com
- http://www.industriallogic.com
- http://www.c2.com
- http://www.refactoring.com
- http://www.martinfowler.com
- http://www.agilemanifesto.org
- http://www.cetus-links.org
- http://hillside.net/patterns
- http://alistair.cockburn.us
- http://www.omg.org/technology/uml
- http://www.industrialxp.org
- http://www.xpdeveloper.net
- http://www.xpexchange.net/

Tavsiye Edilen Online Grup

- http://groups.yahoo.com/group/extremeprogramming

BTSoru.com

BTSoru.com yazılımcıları bir araya getirmek için kurduğum bir soru-cevap platformu. Bu kitap hakkındaki sorularınızı BTSoru.com üzerinden bana yöneltebilirsiniz.

KurumsalJava.com

KurumsalJava.com adresinde blog yazılarım yer almaktadır. Java ve Spring konularındaki yazılarımı buradan takip edebilirsiniz.

KurumsalJava.com'da yer alan yazılarımı bir kitap haline getirdim. PDF formatındaki bu kitabı buradan edinebilirsiniz.

EOF (End Of Fun)

Yolculuğumuzun sonuna geldik. Sizin için burası son durak değil. Extreme Programming ve çevik süreçler hakkında internette birçok faydalı kaynak bulabilirsiniz.

Kitapla birlikte gelen Ant bazlı projeleri pratik yapmak için kullanabilirsiniz. Bu projeleri temel alıp, kendi projelerinizi geliştirebilirsiniz. Aklınıza takılan soruları BTSoru.com üzerinden benimle ve diğer bilişimci arkadaşlarla paylaşabilirsiniz.

Sağlıkcakla kalın. Her şey gönlünüzce olsun.

Özcan Acar