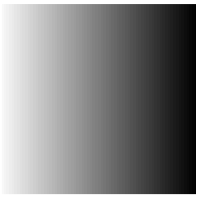


Test Driven Development
Test Güdümlü Yazılım

KurumsalJava.com

Özcan Acar
Bilgisayar Mühendisi
http://www.ozcanacar.com

Bu makale Özcan Acar tarafından yazılmış olan Extreme Programming isimli kitaptan alıntıdır. Extreme Programming ve Çevik Süreçler hakkında genel bilgiyi Özcan Acar tarafından KurumsalJava.com'da yazılmış olan makaleden edinebilirsiniz.

Giriş

Günümüzde kurumsal projelerin büyük bir bölümü geleneksel yazılım metotları ile gerçekleştirilmektedir. Müşteri gereksinimleri en son detayına kadar kağıda döküldükten sonra, programcılar dokumente edilen gereksinimler doğrultusunda yazılımı gerçekleştirmektedirler. Eğer proje bütçesi yeterli ise, yazılım süreci sona erdikten sonra testler hazırlanarak, yazılım sistemi test edilmektedir. çoğu zaman hiçbir unit 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 kardinal hataların ortadan kaldırılması ya imkansız yada 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 sorunları çözmek için test güdümlü yazılım (test driven development – TDD) konsepti geliştirilmiştir.

Kent Beck[1] **Test-Driven Development By Example**[2] isimli kitabında test güdümlü yazılımı şu şekilde tanımlıyor:

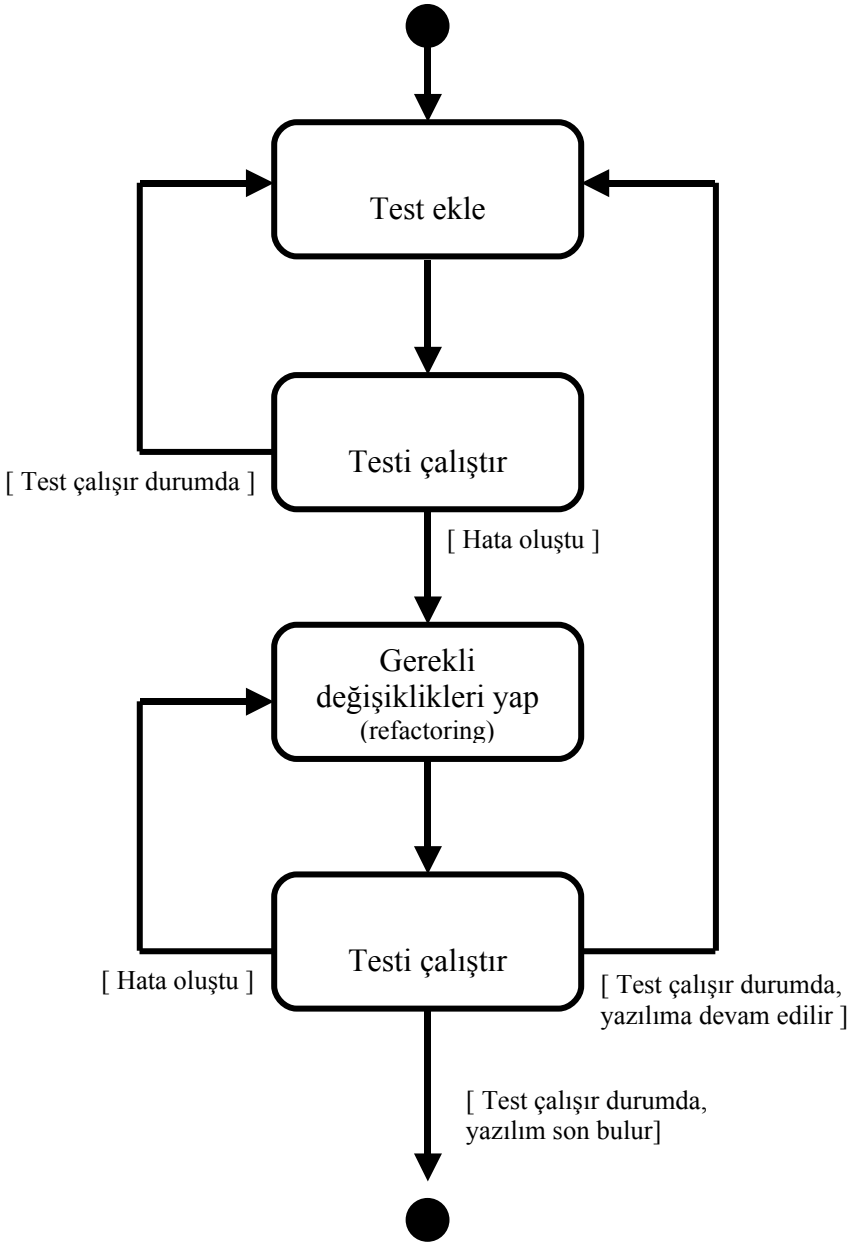
> ***Test-driven development is a set of techniques that any software engineer can follow, which encourage simple design and test suites that inspire confidence.***
>
> ***Test güdümlü yazılım, yazılım mühendislerinin kullanabileceği iyi design ve testleri destekleyen ve dolaylı olarak güven artıran metotlardır.***

Kent Beck aynı kitapta TDD için şu iki kuralı tanımlıyor:

[1] Extreme Programming (XP) Kent Beck tarafından geliştirilmiıtir. Bakınız: http://en.wikipedia.org/wiki/Kent_Beck
[2] ISBN 0-321-14653-0 Addison Wesley

1.**Write a failing automated test before you write any code** (program kodu yazmadan önce çalışmaz durumda olan bir test oluştur)
2.**Remove duplication** (tekrarlanmış kodu yok et)

TDD geleneksel yazılım tarzını tamamen tersine çevirir ve yazılıma JUnit 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.



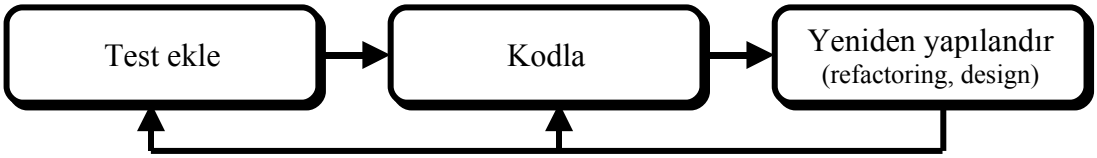
Kent Beck **Test-Driven Development By Example** isimli kitabında TDD için atılması gereken adımların şu şekilde olması gerektiğini yazıyor:

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** (tekrarları 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 design (tasarım) oluşturulur. Akabinde bu design implemente edilir. Son aşama olarak sistem hatalarını bulabilmek için testler yapılır. TDD bu süreci tam tersine çevirir:



"Test ekle – kodla - refactor et" 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 yada 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 design 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 unit testlerinin olması gerektiği sonucunu çıkartıyoruz. Yazılım son bulduğunda sistemde bulunan her sınıf ve metot için JUnit 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:

- JUnit testleri programcıyı, sınıfların nasıl kullanıldıklarını düşünmeye zorlar. Kullanıcı gözüyle sınıflara bakıldığı taktirde, daha basit ve kullanışlı yapılandırılmaları kolaylaşır.
- Programcı JUnit 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.
- JUnit testlerinden yola çıkarak oluşturulan sınıflar daha sağlam bir yapıda olurlar. Programcı JUnit testlerini oluştururken, oluşturduğu sınıfların kullanılış tarzı, olabilecek hatalar ve performansı hakkında düşünür ve buna göre sınıfı yapılandırır.
- JUnit testleri koda olan güveni artırır. Kod üzerinde yapılan değişiklikler yan etkilere sebep verebilir. JUnit testleri olmadan oluşabilecek yan etkilerin tespiti çok zordur. JUnit testleri ile kodun yeniden yapılandırılması (refactoring) kolaylaşır, çünkü oluşabilecek hatalar JUnit testleri ile lokalize edilebilir.
- JUnit testleri sistemin nasıl çalıştığını gösteren dokümentasyon olarak düşünülebilir. Programcılar JUnit testlerini inceleyerek, sistemin nasıl çalıştığını çok kısa bir zaman içinde öğrenebilirler.
- TDD tarzı programlama programcının debugger ile hata arama zamanını kısaltır yada 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 eder. 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, program 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 yada 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.

Test 1: 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 JUnit testi ile işe başlıyoruz.

Kod 9.1 İlk JUnit test sınıfı

```
package dvd;

import junit.framework.TestCase;

public class DvdManagerTest extends TestCase
{
      public void testZeroSizeList()
      {
            assertEquals("listenin büyüklügü 0 degildir",
                        0, manager.size());
      }
}
```

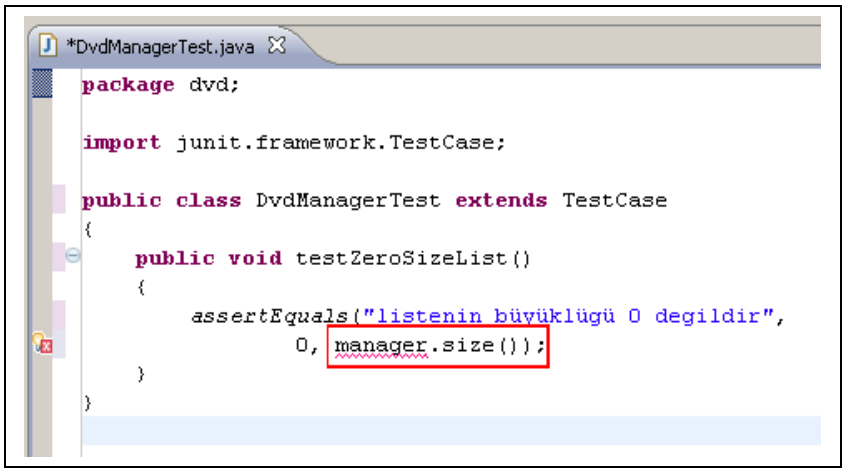
DvdManagerTest isminde bir test sınıfı oluşturuyoruz. İlk test metodunun ismi testZeroSizeList şeklindedir. Test metot isimlerinin seçimi büyük önem taşımaktadır. Metot ismi yapılan işlemi ifade edebilecek şekilde seçilmelidir. Seçtiğimiz metot isminden anlaşıldığı gibi 0 (sıfır) büyüklükte bir listeyi 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[3] 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 sınıfı oluşturur. Programcı çoğu zaman sınıfı oluştururken kullanıcı gözüyle sınıfa bakmadığı için sınıf için oluşturulan metotlar (API) yetersiz yada 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 0 (sıfır) olduğunu belirtiyoruz. Şu ana kadar manager isminde ne bir nesne var, nede 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()

[3] API = Application Programming Interface

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 tamamen 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.

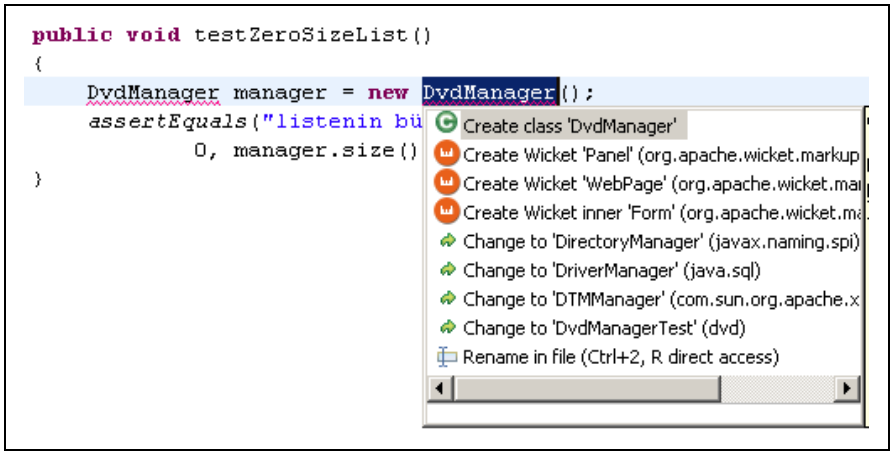
TDD tarzı yazılım yaparken Eclipse gibi yazılım araçları çok faydalı olmaktadır. Eclipse var olmayan nesneleri ve metotları ikaz ederek, bu sınıfların ve metotların otomatik olarak oluşturulmaları için yardımcı olur. Bizde Eclipse'in bu özelliklerinden faydalanarak testimize devam ediyoruz.

Kod 9.2 İlk JUnit test sınıfı

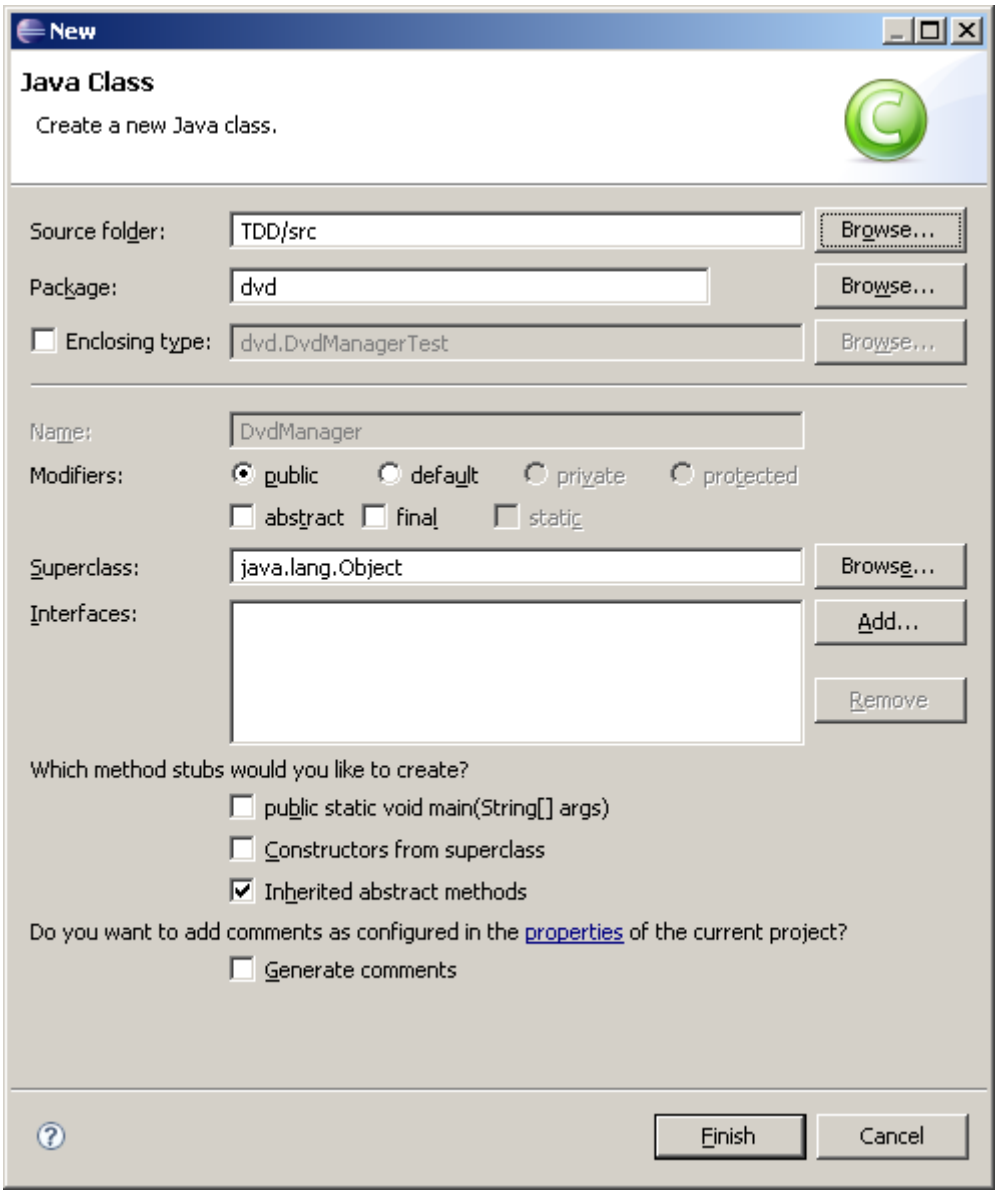
```
package dvd;

import junit.framework.TestCase;

public class DvdManagerTest extends TestCase
{
	public void testZeroSizeList()
	{
		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

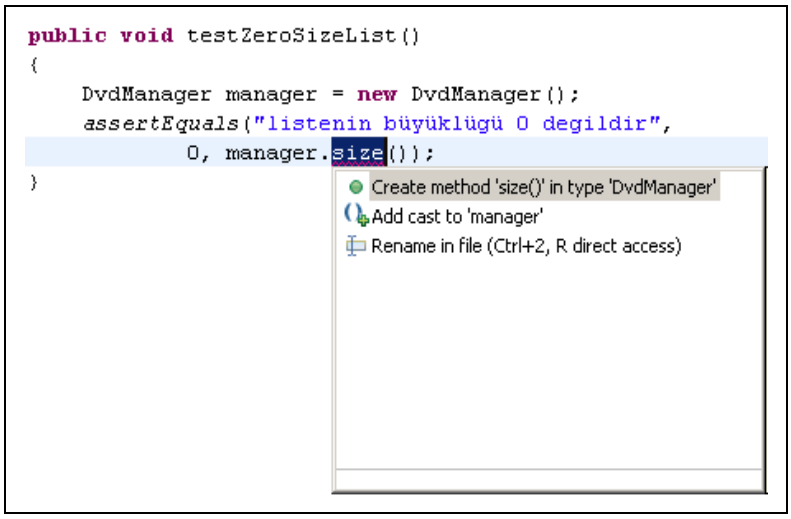


Resim 9.3 Create class DvdManager menüsü üzerinden DvdManager sınıfı oluşturulur

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.



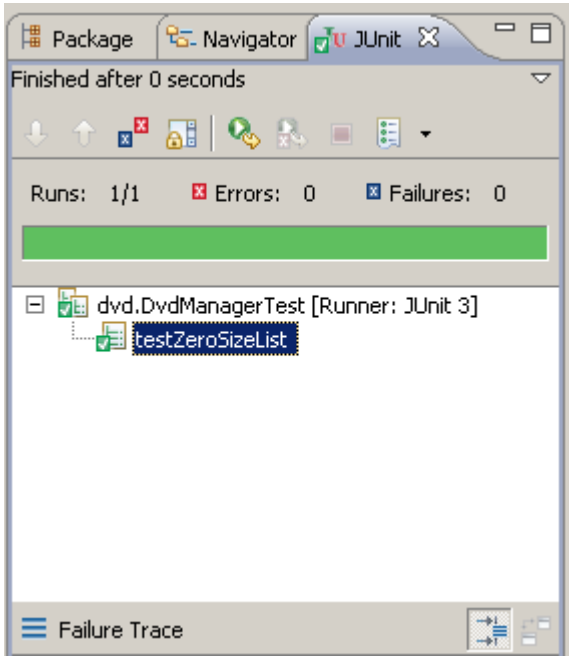
Resim 9.4 size() metodu DvdManager sınıfında henüz bulunmamaktadır

"*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.testZeroSizeList() 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.

Test 2: Listeye bir film eklendiğinde listenin büyüklüğü 1 olmalı.

Yeni bir test metodu oluşturarak, ikinci testi implemente etmeye başlıyoruz.

Kod 9.5 İkinci test metodu

```
public void testListSizeAfterAddingOneItem()
{
	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 *testListSizeAfterAddingOneItem* 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

```
public void testListSizeAfterAddingOne()
{
	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

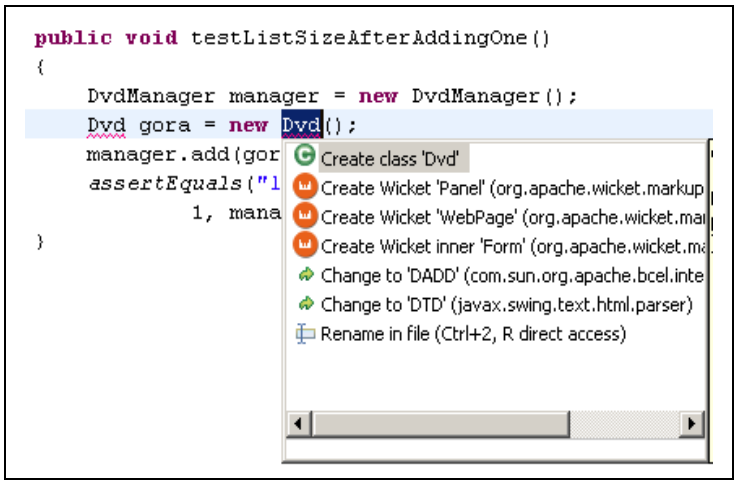
```
public void testListSizeAfterAddingOne()
{
	DvdManager manager = new DvdManager();
	manager.add(gora);
	assertEquals("listenin büyüklügü 1 degildir", 1, manager.size());
}
```

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

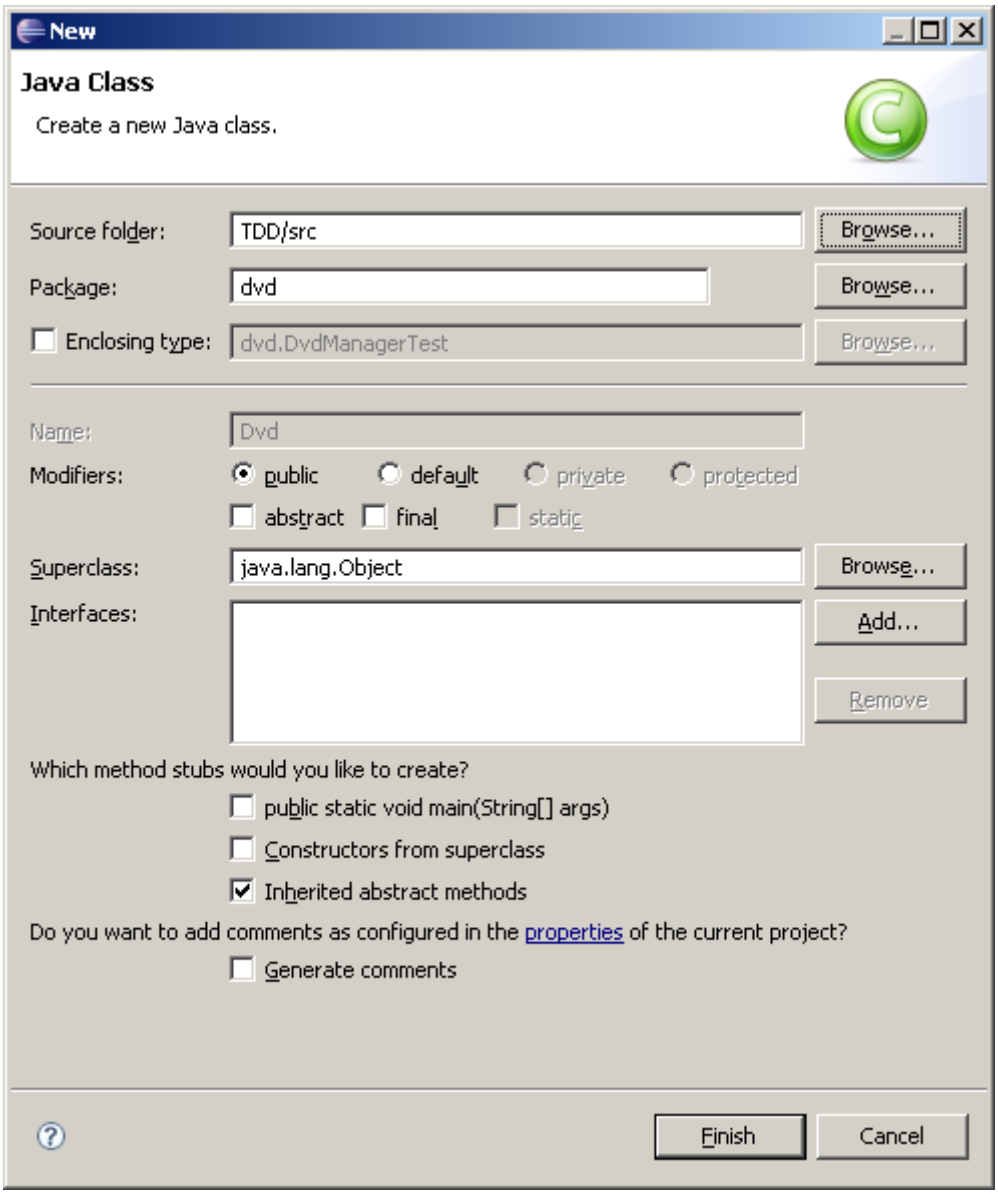
```
public void testListSizeAfterAddingOne()
{
	DvdManager manager = new DvdManager();
```

```
        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, oda 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

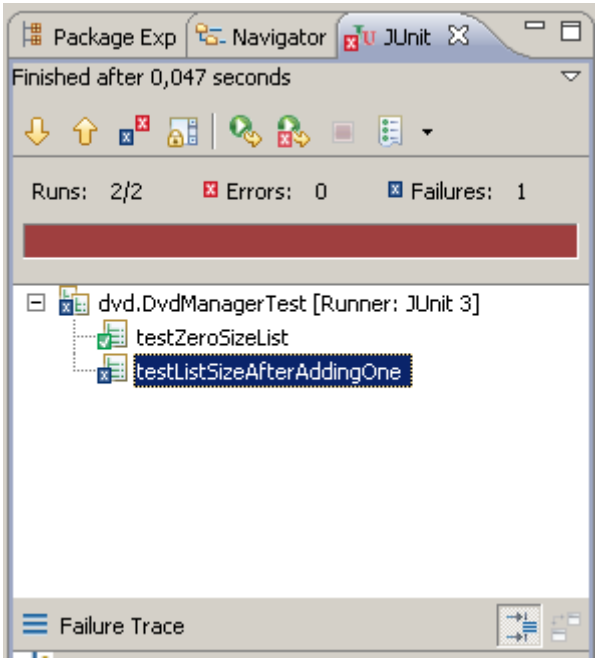
```
package dvd;

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 etkilemesini 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.

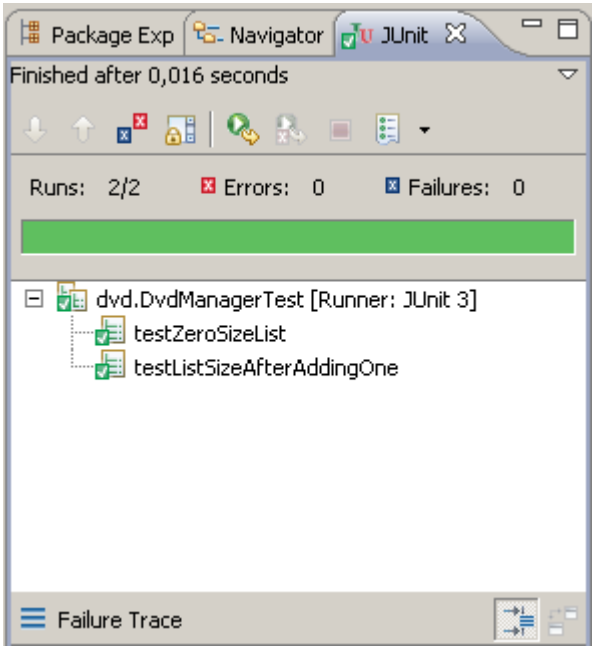
İkinci test olumsuz sonuç vermekte, yani çalışmıyor. Bu testi olumlu sonuç verecek hale getirmek için ne yapmamız gerekiyor? Testten beklentimizi tekrar gözden geçirerek, bu soruya cevap bulabiliriz. Testten beklentimiz şu şekildedir: add() metodunu kullanıp, listeye yeni bir film ekledikten sonra, film 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 yeralmaktadır. 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.

Test 3: 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

```
public void testListSizeAfterAddingTwo()
{
	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());
}
```

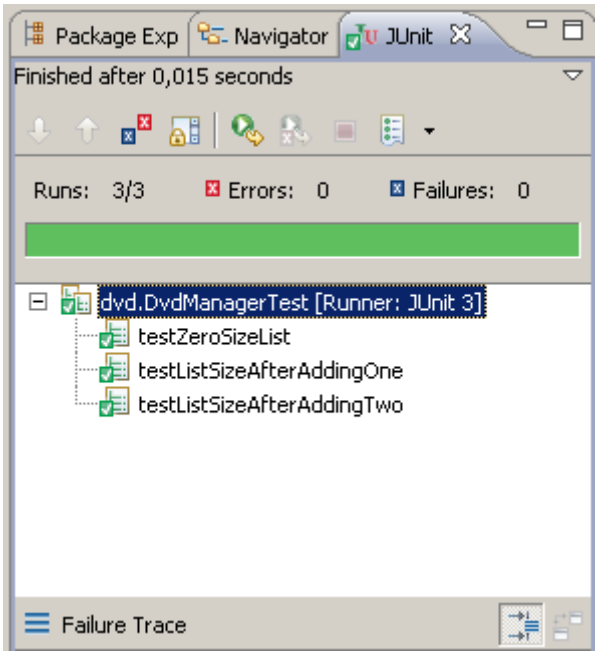
Doğal olarak bu test çalışır durumda değil! add() metodu her kullanıldığında listSize değişkenine 1 değerini eşitler, bu yüzden add() metodu iki sefer arka arkaya kullanılmış olsa bile, listSize değişkeni 1 değerine sahip olacağından, size() metodu 1 değerini geri verecektir. Bizim beklentimiz 2 olduğu 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

Su 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 yeralmaktadır.

Kod 9.14 DvdManagerTest sınıfı

```
package dvd;

import junit.framework.TestCase;

public class DvdManagerTest extends TestCase
{
	public void testZeroSizeList()
	{
		DvdManager manager = new DvdManager();
		assertEquals("listenin büyüklügü 0 degildir",
				0, manager.size());
	}


	public void testListSizeAfterAddingOne()
	{
		DvdManager manager = new DvdManager();
		Dvd gora = new Dvd();
		manager.add(gora);
		assertEquals("listenin büyüklügü 1 degildir",
				1, manager.size());
	}


	public void testListSizeAfterAddingTwo()
	{
		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 yan ısıra Dvd tipinde olan gora değişkeni son iki metotta yer almış. Burada bir kod tekrarlanması (dublication) söz konusu. TDD kurallarına göre tekrarın yok edilmesi gerekiyor. Yeniden yapılandırma (refactoring) metotlarını kullanarak, aynı değişkenin birden fazla yerde kullanılmasını engellememiz gerekiyor. Örneğin bu tekrarlanmış (dublication) 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;

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 tekrarlanmış (dublication) değişkeni (manager) uzaklaştırabiliriz.

Kod 9.16 İlk test metodu

```
public void testZeroSizeList()
{
      assertEquals("listenin büyüklügü 0 degildir", 0, manager.size());
}
```

manager değişkeni artık bir sınıf değişkeni olduğu ve setUp() metodunda oluşturulduğu için, *testZeroSizeList()* 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

kırık oluşabilir ve bunların ortadan kaldırılması zaman alıcı olabilir. Ayrıca aynı zamanda bir kırık yerine, birden fazla kırıkla uğraşmak, odaklandığımız testten uzaklaşmamıza sebep verebilir.

İkinci testide kod 9.17 de olduğu şekilde değiştiriyoruz.

Kod 9.17 İkinci test metodu

```
public void testListSizeAfterAddingOne()
{
	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 yeralmaktadır.

Kod 9.18 Üçüncü test metodu

```
public void testListSizeAfterAddingTwo()
{
	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;

import junit.framework.TestCase;

public class DvdManagerTest extends TestCase
{

	private DvdManager manager = null;
	private Dvd gora = null;
	private Dvd arog = null;

	public void setUp()
	{
		manager = new DvdManager();
		gora = new Dvd();
		arog = new Dvd();
	}

	public void testZeroSizeList()
	{
		assertEquals("listenin büyüklügü 0 degildir",
				0, manager.size());
```

```
	}


	public void testListSizeAfterAddingOne()
	{
		manager.add(gora);
		assertEquals("listenin büyüklügü 1 degildir",
				1, manager.size());
	}


	public void testListSizeAfterAddingTwo()
	{
		manager.add(gora);
		manager.add(arog);
		assertEquals("listenin büyüklügü 2 degildir",
				2, manager.size());
	}
}
```

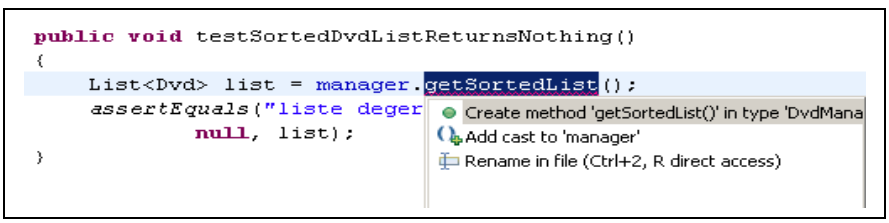
Test 4: BBB ve AAA ismini taşıyan iki film listeye eklendiğinde AAA ismini taşıyan film listede BBB isimli filmden önce yer almalıdır.

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 9.20 Dördüncü test metodu

```
public void testSortedDvdListReturnsNothing()
{
	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.



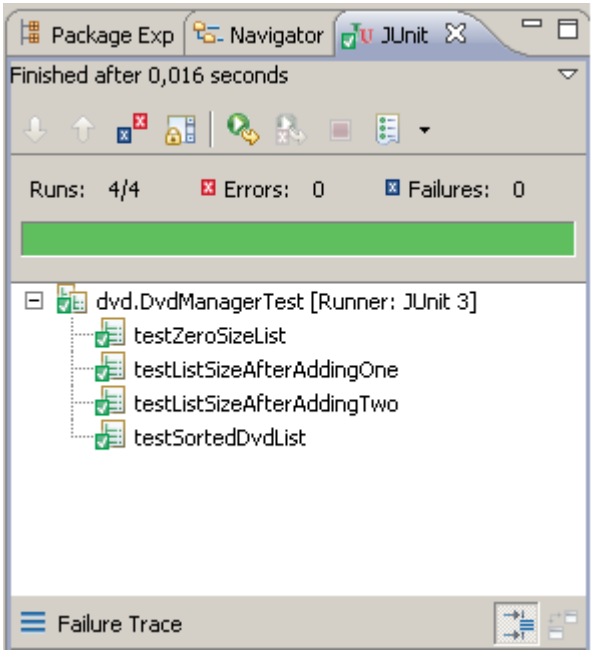
Resim 9.11 Eclipse getSortedList() isimli bir metodun DvdManager sınıfında bulunmadığını tespit etti

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 yada 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

```
public void testSingleDvdReturnedAsSortedList()
{
	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 sinifina setTitle() ve getTitle() metotları ekleniyor

```
package dvd;

public class Dvd
{
	private String title;
```

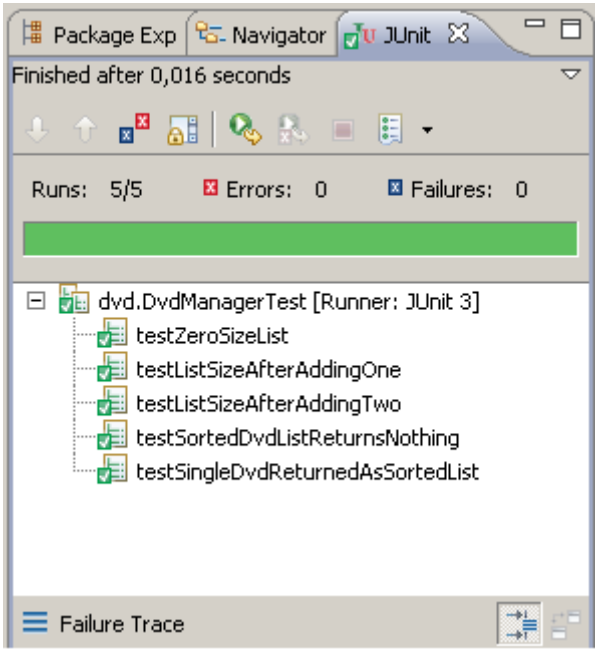
```
	public String getTitle()
	{
		return title;
	}

	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 tekrarlanma (dublication) 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 edinmek mümkündür. Aynı zamanda DvdManager.size() metodunu da listenin büyüklüğünü geri verecek şekilde değiştirdiğimiz taktirde, hem kod tekrarlamaları elimine etmiş olacağız, hemde 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

```
public void testMultipleDvdReturnedAsSortedList()
{
      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:<Arog> but was:<Gora>

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 yeralmaktadır. 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;
}
```

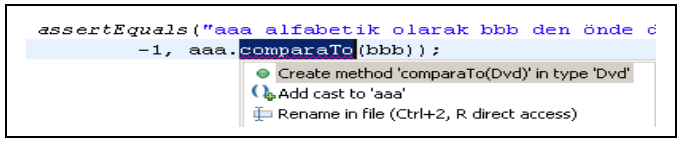
Kullandığımız liste Dvd tipinde nesnelere göre ayarlanmıştır (List<Dvd>). Bu durumda Collectios.sort() metodu bizim listemizi bu halı ile alfabetik hale getiremez, çünkü liste içinde yer alan Dvd nesneleri kıyaslayabileceği bir mekanizmaya sahip değil. Dvd nesnelerini birbirleri ile kıyaslamak için Comparable 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;

import junit.framework.TestCase;

public class DvdTest extends TestCase
{
	public void testCompareAfterBefore()
	{
		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 deki 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

```
public void testCompareSame()
{
	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 testCompareBeforeAfter()
{
	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 filmini alfabetik olarak aaa filminden 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 yeralmaktadır. 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;
import junit.framework.TestCase;

public class DvdManagerTest extends TestCase
{

	private DvdManager manager = null;
	private Dvd gora = null;
	private Dvd arog = null;

	public void setUp()
	{
		manager = new DvdManager();
		gora = new Dvd();
		arog = new Dvd();
	}

	public void testZeroSizeList()
	{
		assertEquals("listenin büyüklügü 0 degildir",
```

```
                        0, manager.size());
        }


        public void testListSizeAfterAddingOne()
        {
                manager.add(gora);
                assertEquals("listenin büyüklügü 1 degildir",
                        1, manager.size());
        }


        public void testListSizeAfterAddingTwo()
        {
                manager.add(gora);
                manager.add(arog);
                assertEquals("listenin büyüklügü 2 degildir",
                        2, manager.size());
        }


        public void testSortedDvdListReturnsNothing()
        {
                List<Dvd> list = manager.getSortedList();
                assertEquals("liste bos degil",
                        0, list.size());
        }


        public void testSingleDvdReturnedAsSortedList()
        {
                manager.add(gora);
                gora.setTitle("Gora");
                List<Dvd> list = manager.getSortedList();
                assertEquals("listedeki ilk film Gora degildir",
                        "Gora", list.get(0).getTitle());
        }


        public void testMultipleDvdReturnedAsSortedList()
        {
                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 junit.framework.TestCase;

public class DvdTest extends TestCase
{
	public void testCompareAfterBefore()
	{
		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));
	}

	public void testCompareSame()
	{
		Dvd aaa = new Dvd();
		aaa.setTitle("aaa");

		assertEquals("aaa alfabetik olarak aaa ile ayni siradadir",
				0, aaa.compareTo(aaa));
	}

	public void testCompareBeforeAfter()
	{
		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)

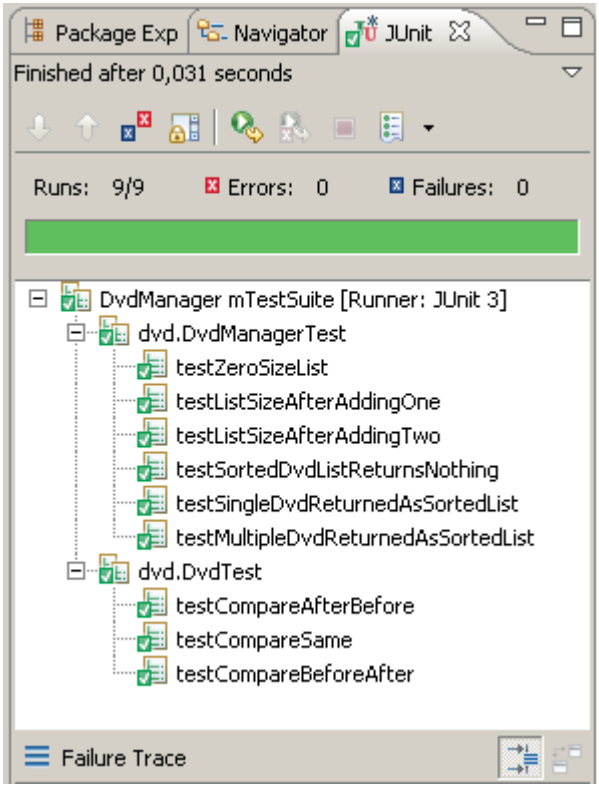
Test kapsama alanı konusunu 8. bölümde ele almıştık. Hazırladığımız testlerin kodun yüzde kaçını işletir hale getirdiğini ölçmek için Eclipse Plugin olan EclEmma programından yararlanabiliriz.

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 yeralmaktadır.

Kod 7.36 DvdManager TestSuite

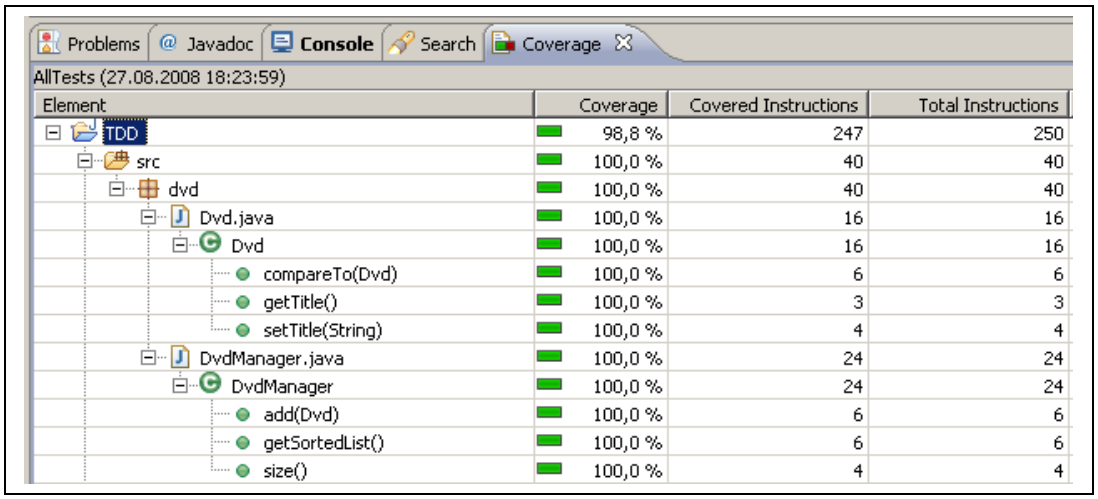
```
package dvd;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests
{
	public static Test suite()
	{
		TestSuite suite = new TestSuite("DvdManager TestSuite");

		suite.addTestSuite(DvdManagerTest.class);
		suite.addTestSuite(DvdTest.class);
		return suite;
	}
}
```

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 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.