Unit Testing Konseptleri

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.

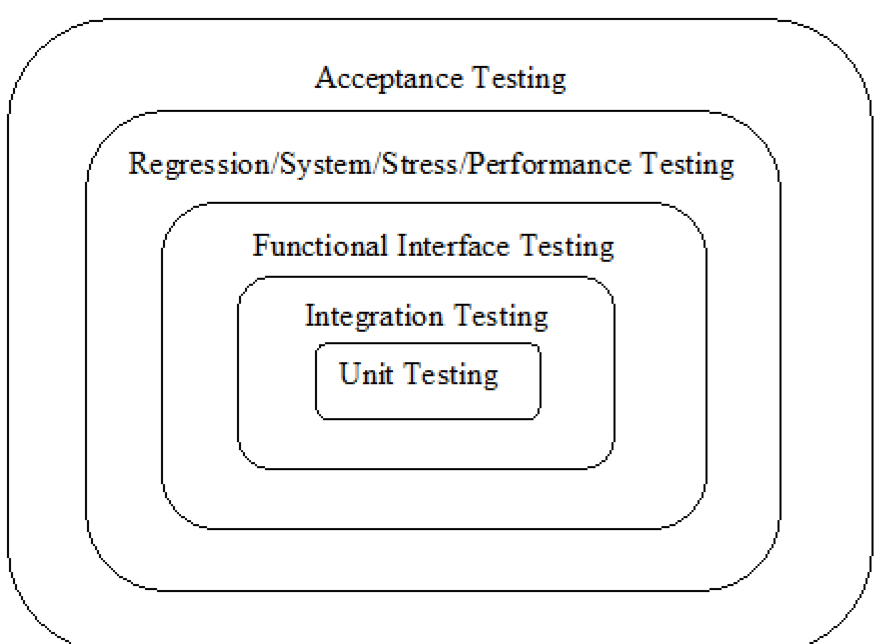
Giriş

Yazılım sürecinde oluşturulan sistemin kalite kontrolü unit testleri ile yapılır. Bu bölümde unit testlerin nasıl hazırlandığını yakından inceleyeceğiz. Bir sonraki bölümünde yer alan test güdümlü yazılımı (Test Driven Development – TDD) uygulayabilmek için unit test konseptlerinin bilinmesi gerekmektedir.

Java tabanlı sistemlerde unit testleri JUnit[1] olarak isimlendirilir. Bu isim aynı ismi taşıyan test frameworkü olan JUnit'den gelmektedir. Java'da unit testleri yazabilmek için JUnit frameworkünden faydalanacağız.

Değişik türde unit testleri oluşturmak mümkündür. Bunlar:

- JUnit testleri (Unit Testing)
- Entegrasyon testleri (Integration testing)
- Arayüz (interface) testleri (Functional Interface Testing)
- Regresyon testleri (Regression Testing)
- Akseptans 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. Unit ismi buradan gelmektedir. Unit testleri ile kendi içinde bütün olan bir kod ünitesi test edilir. Bunlar genellikle sınıfların metotlarıdır. Test sınıfı kullanılarak, test edilen sınıfın işlevlerini doğru

[1] Bakınız: http://www.junit.org

olarak yerine getirip, getirmediği test edilir. Metot bazında hazırlanan testlere **JUnit testleri** diyoruz. Sınıf üzerinde yapılan her değişiklik ardından o sınıf için hazırlanan unit testleri çalıştırılarak, yapılan değişikliğin yan etkileri olup, olmadığı kontrol edilir. Testlerin olumlu netice vermesi durumda, sınıfın görevini hatasız yerine getirdiği ispat edilmiş olur. Bu açıdan bakıldığında unit 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 sadece unit testleri ile kontrol edebiliriz. Onlarca ya da yüzlerce sınıfın olduğu bir programı, her değişikliğin ardından elden kontrol etmek imkansız olduğu için otomatik çalışabilen unit testlerine ihtiyacımız vardır. Daha sonra yakından inceleyeceğimiz JUnit frameworkü 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, JUnit testleri hakkında bir açıklama daha yapma gereği duyuyorum. JUnit testleri, test edilen sınıfları kullandıkları diğer sınıflardan bağımsız olarak test ederler. Örneğin bir sınıf bilgibankasından veri edinmek için bir servis sınıfını kullanıyorsa, JUnit testinde bu servis sınıfı kullanılmaz, çünkü bu bilgibankasının çalışır durumda olmasını gerektirir. Eğer JUnit testi içinde bilgibankası kullanılıyorsa, bu JUnit testi değil, entegrasyon testidir, çünkü test edilen sınıf ile bilgibankası arasındaki ilişki dolaylı olarak test edilmektedir. Daha öncede belirttiğim gibi JUnit testleri metot bazında ve sınıfın dış dünyaya olan bağımlılıklarından bağımsız olarak gerçekleştirilir. Amacımız bir metot içinde bulunan kod satırlarının işlevini test etmektir. Bunun için bilgibankasına bağlantı oluşturulması gerekmez. Test edilen sınıfın bağımlılıklarını ortadan kaldırabilmek için Mock nesneler kullanılır. Bir Mock nesne ile servis sınıfı işlevini yerine getiriyormuşçasına taklit (simulation) edilir. JUnit 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 (bilgibankası, email serveri 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 sistemden 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 alt yapı tanımlanmış bir duruma getirildikten sonra testler uygulanır. Örneğin test öncesi bilgibankası silinerek, test için gerekli veriler tekrar yüklenir. Her test başlangıcında aynı veriler kullanılarak, sistemin nasıl reaksiyon gösterdiği test edilir. Regresyon testlerinin uygulanabilmek için test öncesinde tüm alt yapının başlangıç noktası olarak tanımlanan bir duruma getirilmesi gerekmektedir.

Akseptans (onay - kabul) testleri ile sistemin bütünü kullanıcı gözüyle test edilir. Bu tür testlerde sistem kara kutu (blackbox) olarak düşünülür. Bu yüzden akseptans 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. Akseptans testlerinde sistemden beklenen geri dönüm test edilir.

Stres testleri tüm sistemin davranışını eksepsiyonel 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 (hardware, ram, işletim sistemi) stres altındaki davranışı test edilmiş olur.

Performans testleri ile sistemin, tanımlanmış kaynaklar (hardware, ram, 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 olmadığı 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 unit 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 unit testleri oynar. Sınıflar oluşturulmadan önce test sınıfları oluşturulur. Bu belki ilk bakışta çok tuhaf bir yaklaşım gibi görünebilir. Var olmayan bir sınıf için nasıl unit 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 çevirirler. Bu süreçte çoğu zaman sistemi kullanıcı gözlüğüyle değil, programcı gözlüğüyle görürler. Bu da belki ilk etapta gerekli olmayan davranışların sisteme eklenmesine sebep verebilir. Unit testlerinde bu durum farklıdır. Örneğin akseptans testlerinde tüm sistem bir kullanıcının perspektifinden test edilir. Bu testlerde sistemin mutlaka sahip olması gerektiği davranışlar test edilmiş olur. Eğer akseptans 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 yada yüzlerce unit testi oluşturmak çok zor olacağı için, unit 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
 *
```

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

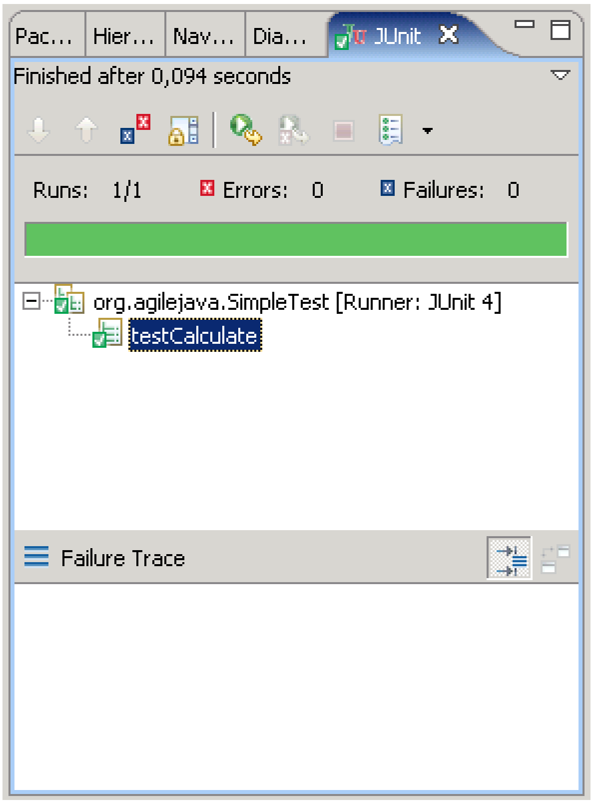
Her JUnit sınıfı **junit.framework.TestCase** sınıfını genişleterek bir JUnit test sınıfı haline gelir. Her JUnit sınıfında setUp() ve tearDown() metotları yeralır. Test metotlarının isimleri **test** ile başlar, örneğin testCalculate(). JUnit frameworkü, JUnit sınıfında tanımlanmış ve test ile başlayan 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.

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 unit test konseptinin en basit halini görmekteyiz. Unit 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 yada belirli bir duruma (state) sahip olur. Bu değer ya da durum JUnit test metodunda assertTrue gibi komutlarla kontrol edilir .Eğer beklediğimiz değer yada durum oluşmuş ise, o zaman test edilen Java sınıfı üzerine düşen görevi doğru olarak yerine getirmiştir. JUnit frameworkünde 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** kontekst menüsü üzerinden çalıştırılabilir.



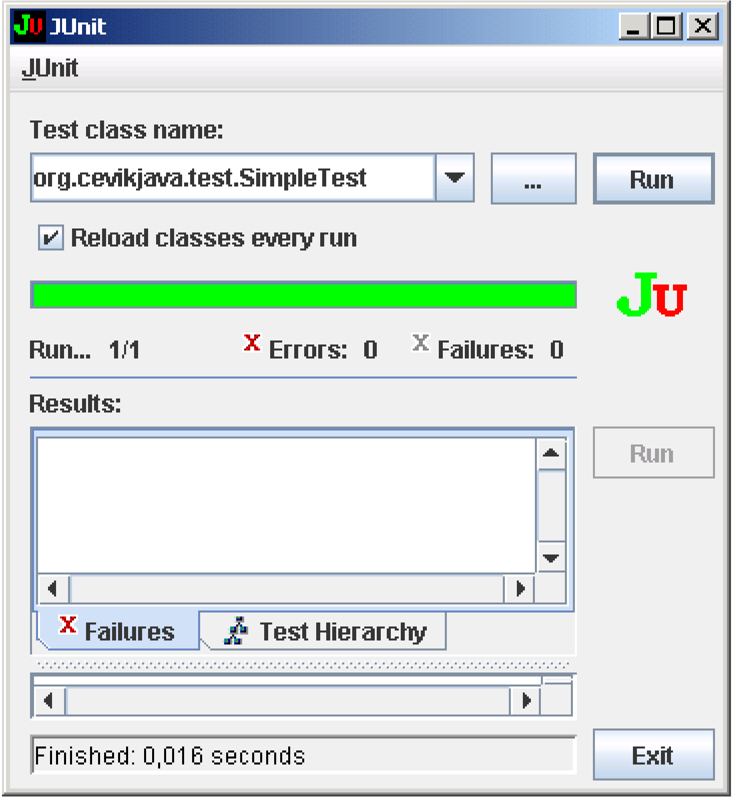
Resim 8.1 JUnit Eclipse Plugin

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 Windows 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

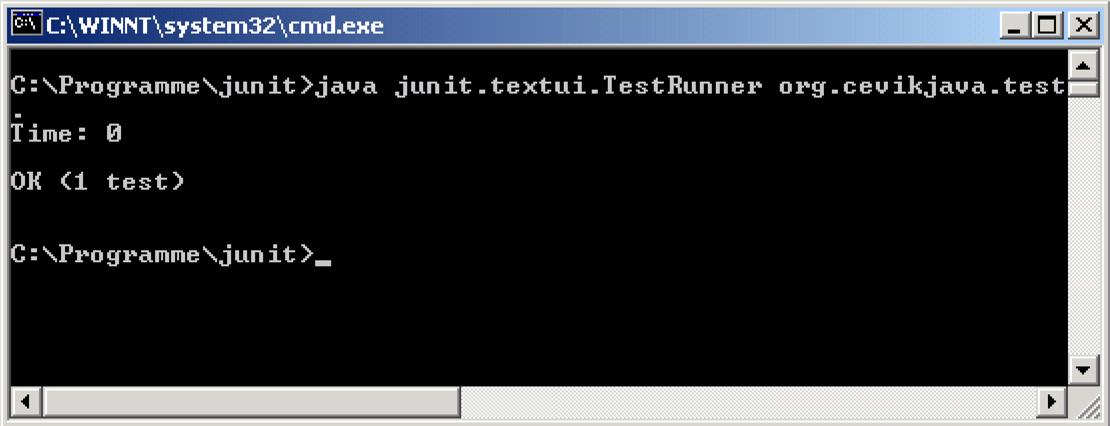


Resim 8.2 JUnit SwingUI

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 sınıfı gruplamak ve aynı anda çalıştırmak için JUnit TestSuite sınıfı kullanılır.

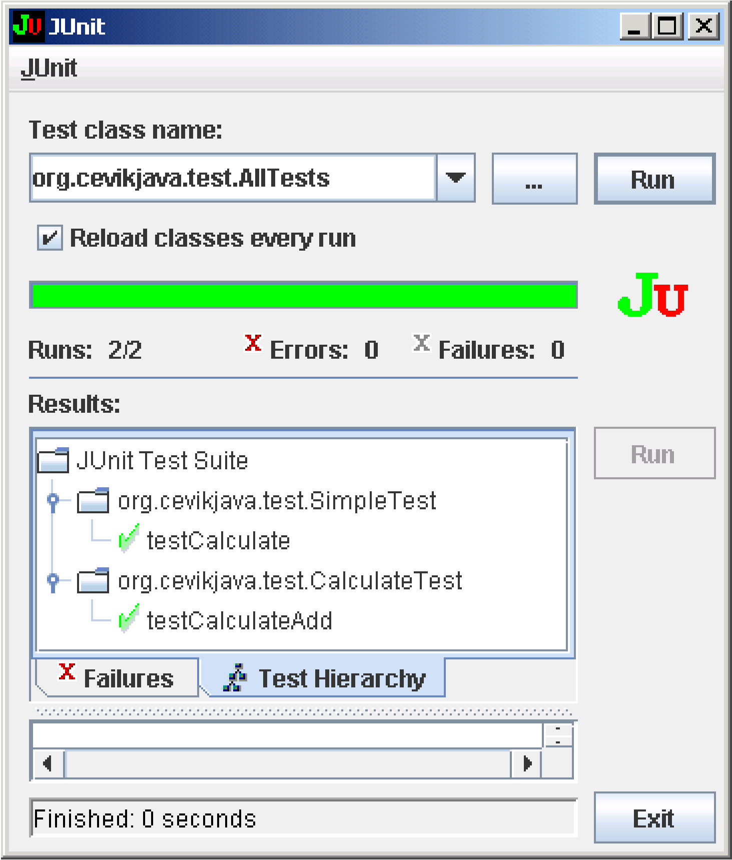
Kod 8.2 AllTests.java

```
package org.cevikjava.test;
import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * Birden fazla testi gruplamak
 * için kullanilan Test Suite sınıfı
 *
 * @author Oezcan Acar
 *
 */
public final class AllTests
{
	private AllTests()
	{
	}

	public static Test suite()
	{
		TestSuite suite = new
				TestSuite("JUnit Test Suite");
		suite.addTestSuite(SimpleTest.class);
		suite.addTestSuite(CalculateTest.class);
		return suite;
	}
}
```

Gruplamak istediğimiz testler için AllTests isminde yeni bir sınıf oluşturuyoruz. TestSuite sınıfı kullanabilmek için static ve Test nesnesini geriye veren suite() isminde bir metot oluşturmamız gerekiyor. Bu metot bünyesinde TestSuite sınıfından bir nesne oluşturarak, testleri addTestSuite() metoduyla grupluyoruz.

Oluşturduğumuz yeni TestSuite'i 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 oluşturmakta 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

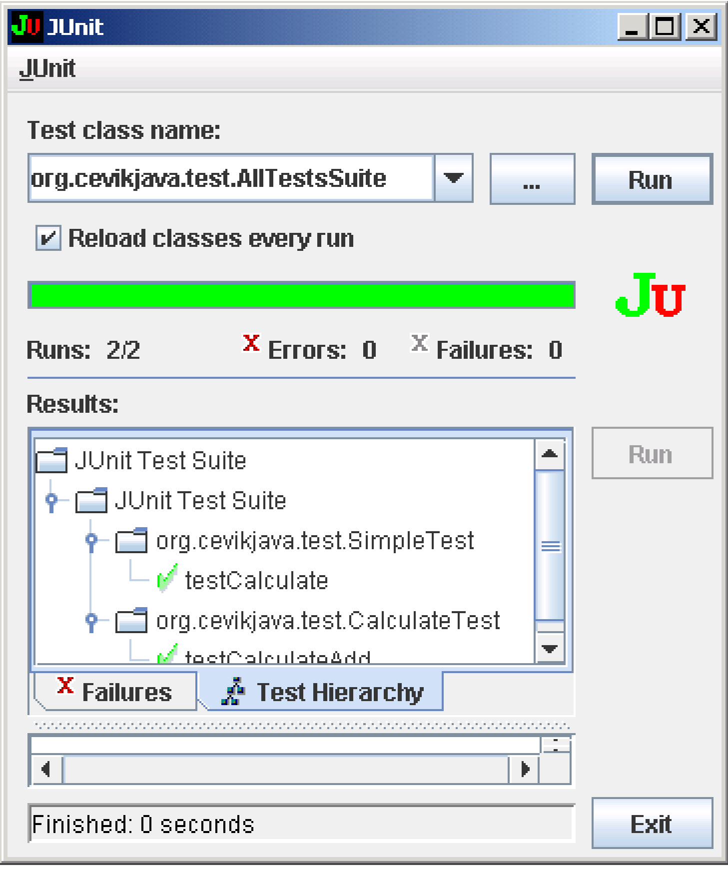
```
package org.cevikjava.test;
import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * JUnit test gruplari (TestSuite)
 * bir araya getirilerek daha büyük
 * gruplar olusturulabilir.
 *
 * @author Oezcan Acar
 *
 */
public final class AllTestsSuite
{
    private AllTestsSuite()
    {
```

```
	}

	public static Test suite()
	{
		TestSuite suite = new
			TestSuite("JUnit Test Suite");
		suite.addTest(AllTests.suite());
		return suite;
	}
}
```

TestSuite'lerden oluşan bir grup oluşturmak için TestSuite örneğinde olduğu gibi yeni bir sınıf tanımlıyoruz (AllTestsSuite). Bu sınıfın static suite() isminde bir metodu bulunmaktadır. Bu metot bünyesinde bir TestSuite nesnesi oluşturuyoruz. addTest() metodu ile bu test grubuna mevcut bir test grubunu ekliyoruz (AllTests.suite()). addTest() metodu kullanılarak birden fazla test grubu (TestSuite.suite()) bu test grubuna eklenebilir.

Test gruplarından oluşan test grubu aşağıdaki şekilde çalıştırılır:

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



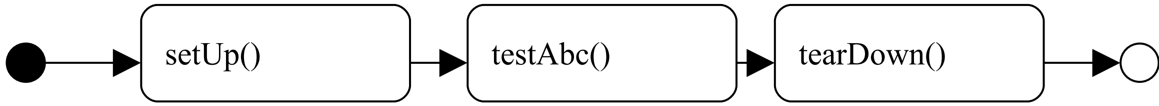
Resim 8.5 JUnit SwingUI

JUnit Anatomisi

TestSuite sınıfı yardımıyla JUnit testlerini gruplayabileceğimizi gördük. Oluşturulan TestSuite sınıfları kullanılarak testler gruplanır ve çalıştırılır. Programcılar günlük işlerinde daha çok TestCase sınıfını kullanırlar. Oluşturulan unit testleri `junit.framework.TestCase` sınıfını genişletir ve bir unit test sınıfı halini alır.

Bazı testler için test öncesi çalışabilecekleri bir ortamın oluşturulması gerekmektedir. Örneğin test bilgibankasına bağlanarak, veri okuyabilir yada veri depolayabilir. Bu durumda bilgibankasına 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 5 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

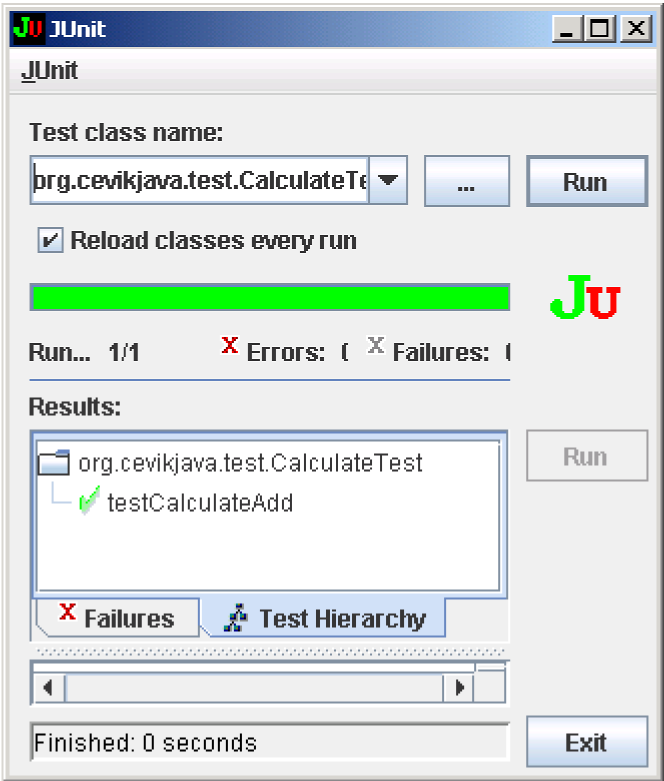
```
package org.cevikjava.test;

/**
 * Toplama islemi yapan Calculate
 * sınıfı.
 *
 * @author Oezcan Acar
 *
 */
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 junit.framework.TestCase;

/**
 * Calculate sınıfini test eder.
 *
 * @author Oezcan Acar
 *
 */
public class CalculateTest extends TestCase
{

	private Calculate calculate;

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

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

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

	/**
	 * teardown() bünyesinde test bittikten
	 * sonra calculate nesnesini ortadan
	 * kaldiriyoruz.
	 */
	public void tearDown() throws Exception
	{
		super.tearDown();
		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 bilgibankasına 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:

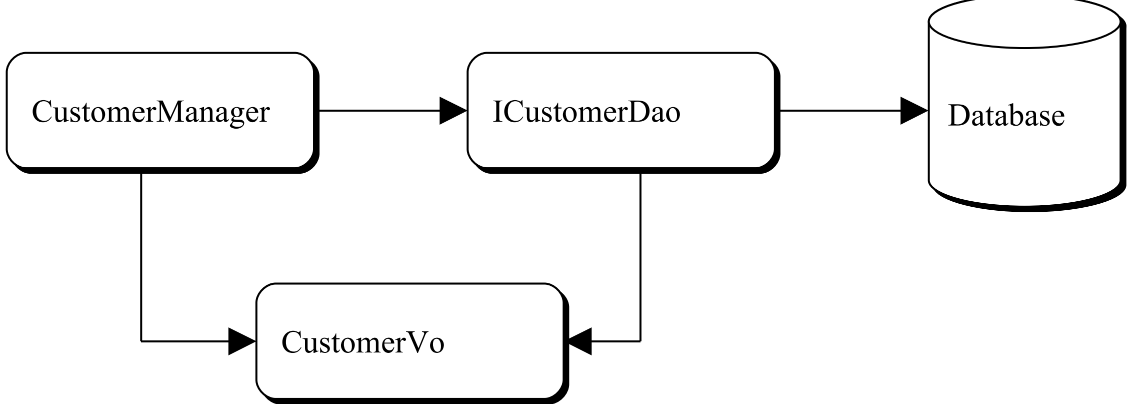
- Kodun daha iyi anlaşılmasını ve takım çalışmasını destekler
- Emma gibi araçlar aracılığıyla kodun hangi bölümlerinin JUnit testleri ile kullanılığı (code coverage) tespit edilebilir. Hiç çalışmayan kod satırlarının lokalizyonu kolaylaşır.
- JUnit testleri programcının oluşturduğu bir nevi dokümentasyondur. 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, yada 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 taklit 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 aktüel 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.samples.customer;

/**
 * Müsteri yönetimini yapan
 * sınıf.
 *
 * @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;
	}
}
```

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.samples.customer;


/**
 * CustomerDao interface sınıfı
 *
 * @author Oezcan Acar
 *
 */
public interface ICustomerDao
{
     CustomerVo getCustomer(long id);
}
```

ICustomerDao bünyesinde getCustomer() isminde bir metod tanımlar. Bu metod aracılığıyla id'si belli bir müşterinin bilgileri bilgibankasından bulunarak CustomerVo nesnesi olarak geri verilir.

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? Bir mock nesne kullanarak...

Kod 8.8 CustomerManagerTest.java

```
package org.cevikjava.test.mock;

import org.cevikjava.samples.customer.CustomerManager;
import org.cevikjava.samples.customer.CustomerVo;
import org.cevikjava.samples.customer.ICustomerDao;
import org.jMock.Mock;
import org.jMock.MockObjectTestCase;

/**
 * CustomerManager sınıfı için JUnit
 * testi.
 *
 * @author Oezcan Acar
 *
 */
public class CustomerManagerTest
          extends MockObjectTestCase
{

     /**
```

```
     * Test edilen sınıf.
     */
    private CustomerManager manager;


    /**
     * ICustomerDao yerine kullanilan
     * mock nesne
     */
    private Mock mockDao;

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

    private CustomerVo vo;


    /**
     * setUp() içinde  CustomerManager
     * ve mock nesnesi oluşturulur.
     */
    public void setUp() throws Exception
    {
       super.setUp();
       mockDao = new Mock(ICustomerDao.class);
       dao = (ICustomerDao)mockDao.proxy();
       manager = new CustomerManager(dao);
       vo = new CustomerVo();
       vo.setName("Acar");
       vo.setFirstname("Oezcan");
    }


    /**
     * CustomerManager.getCustomer() metodunu
     * test etmek için kullanilir.
     *
     * CustomerManager sınıfı ICustomerDao
     * interface sınıfina bagimli oldugu için
     * direk test edilmesi zordur. Bu sebepten
     * dolayi bir mock nesne kullanarak,
     * ICustomer sınıfini simule edebiliriz.
     *
     */
    public void testGetCustomer()
    {
         /**
          * Mock nesneyi beklentilerimi
          * dogrultusunda programliyoruz
```

```
           */

          mockDao
          // sadece bir kere getCustomer metodu
          //kullanilacak
          .expects(once())
          // kullanilan metodun ismi getCustomer()
          .method("getCustomer")
          // metod parametre degeri 1
          .with(eq(1L))
          // metodun geri verdigi nesne vo
          .will(returnValue(vo));


          CustomerVo cusVo = manager.getCustomer(1);
          assertNotNull(cusVo);
          assertEquals(cusVo.getName(), vo.getName());
          assertEquals(cusVo.getFirstname(),
                    vo.getFirstname());
     }


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

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 JUnit 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 yerlan 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:

```
mockDao = new Mock(ICustomerDao.class);
```

Mock sınıfı, jMock frameworkünde yer alan bir sınıftır. Bu sınıf mocklanmasını istediğimiz interface sınıfı parametre olarak alır.

ICustomerDao sınıfını implemente etmiş olan mockDao nesnesi kullanılarak, ICustomerDao tipinde bir nesneyi aşağıdaki satırla oluşturuyoruz:

```
dao = (ICustomerDao)mockDao.proxy();
```

dao değişkeni bir mock nesnedir ve CustomerManager tarafından kullanılmak üzere konstruktör 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:

```
mockDao
// sadece bir kere getCustomer metodu
//kullanilacak
.expects(once())
// kullanilan metodun ismi getCustomer()
.method("getCustomer")
// metod parametre degeri 1
.with(eq(1L))
// metodun geri verdigi nesne vo
.will(returnValue(vo));
```

Mock programlama işlemini kısaca su 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. MockDao.expects(once()) ile getCustomer() metodunun sadece bir kere kullanılabileceğini ifade etmiş oluyoruz. Daha sonra göreceğimiz gibi, eğer CustomerManager programladığımız şekilde mock nesneyi kullanmazsa, bu bir hatadır ve test hata verir.

CustomerManager.getCustomer() metodunu aşağıda yer alan satırlarla test ediyoruz:

```
CustomerVo cusVo = manager.getCustomer(1);
assertNotNull(cusVo);
assertEquals(cusVo.getName(), vo.getName());
assertEquals(cusVo.getFirstname(),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:

```
// metodun geri verdigi nesne vo
.will(returnValue(vo));
```

Mock nesnesi getCustomer(1) metodunu çalıştırdıktan sonra vo ismindeki nesneyi geriye verecektir.

setUp() içinde oluşturduğumuz vo isimli nesne şu şekildedir:

```
CustomerVo 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.

Eğer CustomerManager mock nesneyi programlandığı gibi kullanmassa test hata verir. Bunun bir örneği aşağıda yeralmaktadır.

```
org.jMock.core.DynamicMockError: mockICustomerDao: unexpected invocation
Invoked: mockICustomerDao.getCustomer(<1L>)
Allowed:
expected once: getCustomer( eq(<2L>) ), returns
<org.cevikjava.samples.customer.CustomerVo@8fce95>

        at
org.jMock.core.AbstractDynamicMock.mockInvocation(AbstractDynamicMock.java:96)
        at org.jMock.core.CoreMock.invoke(CoreMock.java:39)
        at $Proxy0.getCustomer(Unknown Source)
        at
org.cevikjava.samples.customer.CustomerManager.getCustomer(CustomerManager.java:
22)
        at
org.cevikjava.test.mock.CustomerManagerTest.testGetCustomer(CustomerManagerTest.jav
a:86)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:
39)
```

```
	at 
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
25)
	at java.lang.reflect.Method.invoke(Method.java:585)
	at junit.framework.TestCase.runTest(TestCase.java:154)
	at org.jMock.core.VerifyingTestCase.runBare(VerifyingTestCase.java:39)
	at junit.framework.TestResult$1.protect(TestResult.java:106)
	at junit.framework.TestResult.runProtected(TestResult.java:124)
	at junit.framework.TestResult.run(TestResult.java:109)
	at junit.framework.TestCase.run(TestCase.java:118)
	at 
org.eclipse.jdt.internal.junit.runner.junit3.JUnit3TestReference.run(JUnit3TestReference.jav
a:130)
	at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
	at 
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:
460)
	at 
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:
673)
	at 
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:386)
	at 
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:196)
```

Yukarda yerlan hata getCustomer() metodunun metot parametresi olarak 1 yerine 2 rakamı kullanıldığında oluşmaktadır. Mock nesnesini programlarken metot parametresi olarak 1 rakamının kullanılacağını teyit etmiştik:

```
mockDao
// sadece bir kere getCustomer metodu
//kullanilacak
.expects(once())
// kullanilan metodun ismi getCustomer()
.method("getCustomer")
// metod parametre degeri 1
.with(eq(1L))
// metodun geri verdigi nesne vo
.will(returnValue(vo));
```

Bu durumda test esnasında manager.getCustomer(1); şeklinde kullanım gerekmektedir. Aksi taktirde jMock testi geçersiz olarak durduracak ve yukarda yeralan hata mesajını verecektir.

Mock nesneleri oluşturmak için jMock gibi bir framework 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.samples.customer;


/**
 * Dummy ICustomerDao interface
 * implementasyonu.
 *
 * @author Oezcan Acar
 *
 */
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 bilgibankasına 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.test.mock;

import junit.framework.TestCase;
import org.cevikjava.samples.customer.CustomerDummyDaoImpl;
import org.cevikjava.samples.customer.CustomerManager;
import org.cevikjava.samples.customer.CustomerVo;
import org.cevikjava.samples.customer.ICustomerDao;

/**
 * CustomerManager sınıfı için JUnit
 * testi.
 *
 * @author Oezcan Acar
 *
 */
```

```java
public class CustomerManagerTest2
          extends TestCase
{

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



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

     private CustomerVo vo;


     /**
      * setUp() içinde  CustomerManager
      * ve mock nesnesi oluşturulur.
      */
     public void setUp() throws Exception
     {
        super.setUp();
        dao = new CustomerDummyDaoImpl();
        manager = new CustomerManager(dao);
        vo = new CustomerVo();
         vo.setName("Acar");
        vo.setFirstname("Oezcan");

     }


     /**
      * CustomerManager.getCustomer() metodunu
      * test etmek için kullanilir.
      *
      * CustomerManager sınıfı ICustomerDao
      * interface sınıfina bagimli oldugu için
      * direk test edilmesi zordur. Bu sebepten
      * dolayi bir mock nesne kullanarak,
      * ICustomer sınıfini simule edebiliriz.
      *
      */
     public void testGetCustomer()
     {
          CustomerVo cusVo = manager.getCustomer(1);
          assertNotNull(cusVo);
          assertEquals(cusVo.getName(), vo.getName());
```

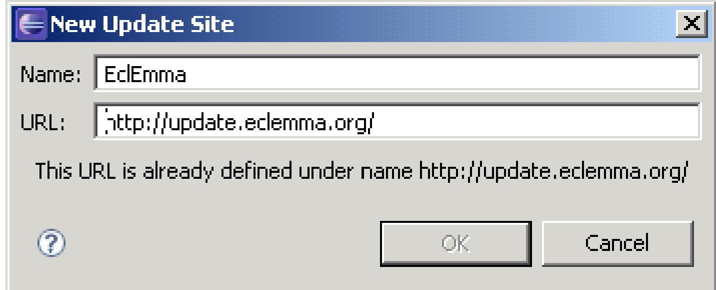
```
            assertEquals(cusVo.getFirstname(),
                        vo.getFirstname());
      }


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

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 konstruktör 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 kompleks yapıdaki nesneleri taklit etmek için gerçek mock (jMock gibi) nesneleri kullanılmalıdır. Mock nesnelerini programlamak ve davranış biçimlerini tayin etmek, kompleks nesnelerde daha kolay olacaktır.

Test Kapsama Alanı (Test Coverage)

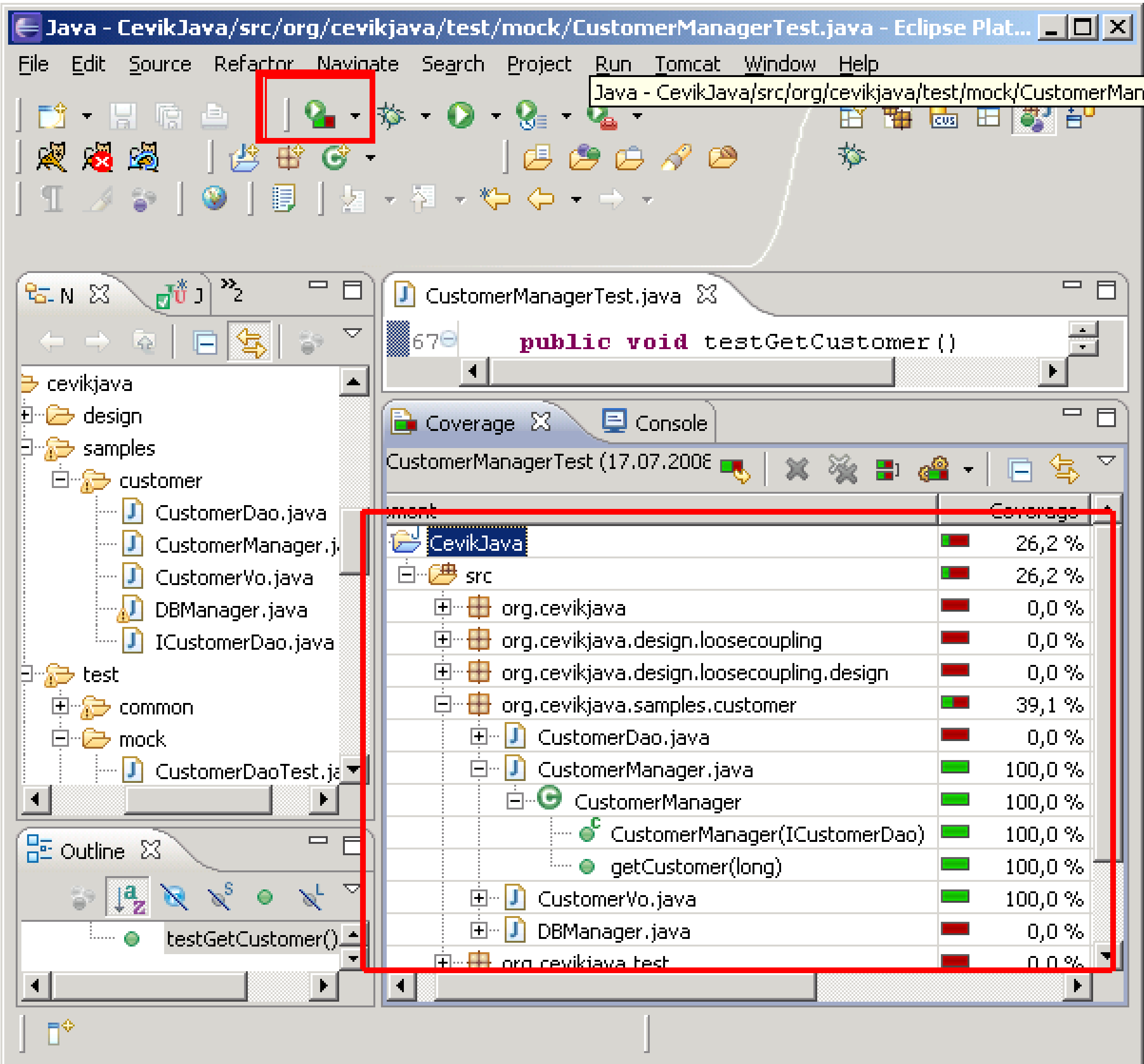
JUnit testleri ve gerekli araçlar yardımıyla kodun hangi satırlarının işlem gördüğünü, hangi satırların hiç kullanılmadığını ve ölü olduğunu tespit edebiliriz. Bu işleme testin kapsama alanı tespiti denir. Test kapsama alanı ne kadar geniş olursa, programın o kadar büyük bölümü test edilmiş demektir. 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[2] programıyla tespit etmek mümkündür. EclEmma Eclipse plugindir ve Resim 8.7 de görüldüğü gibi install edilebilir.



Resim 8.7 EclEmma installasyonu Eclipse altında Help>Software Updates > Find and Install bölümünden yapılabilir.

[2] Bakınız: http://www.eclemma.org



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, çalıştırabiliriz. 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 %26.2 dir. **org.cevikjava.samples.customer** paketi için test kapsama alanının %39.1 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() 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 8.9 da 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 8.9 Test kapsama alanı EclEmma tarafından renkli olarak gösterilir.