Test di unità con Junit e codecover

Angelo Gargantini

Testing e Verifica del software 16/17

Unit testing alla XP

JUnit è un framework per l'automazione del testing di unità di programmi Java che si ispira al eXtreme Programming

Ogni test contiene le asserzioni che controllano che il programma non contiene difetti

Nota: XP è molto di più che JUnit, è proprio un modo nuovo di organizzare la fase implementativa (e il modo di lavorare) che coinvolge anche la fase di testing

Ne riparleremo...

Sviluppo guidato dai test

Lo sviluppo guidato dai test "Test-Driven Development Cycle" prevede i seguenti passi:

1. **scrivi i casi di test**
   - dalle specifiche (casi d'uso e storie dell'utente) scrivi un possibile scenario di chiamata di metodi e/o classi
2. **esegui i test (che falliscono)**
3. **scrivi il codice fino a quando tutti i casi di test passano**
4. **ricomincia da 1**

- Se trovi un difetto ricomincia da 1: scrivi i casi di test che falliscono per colpa del difetto e poi modifica il codice

Principi dello sviluppo guidato dai test

Il progetto e il codice evolvono sotto la guida di test e scenari reali

Anche se ti rallentano inizialmente, alla fine portano più vantaggi che svantaggi

- il codice è doppio (applicazione + test)

I test fanno parte della documentazione dell'applicazione

L'unit testing deve essere fatto automaticamente

- il giudizio che un metodo/classe sia corretto deve essere fatto dal codice stesso

Alcuni tools

Ci sono diversi tools per Unit testing nei diversi linguaggi

Per C++:

- CUnit/ CPPUnit

Per Java il più diffuso è JUnit

- http://junit.sourceforge.net/
- scritto da Kent Beck (autore di XP) e Erich Gamma (autore dei design pattern)

Junit

Supporta i programmatori nel:

- definire ed eseguire test e test set
- formalizzare in codice i requisiti sulle unità
- scrivere e debuggare il codice
- integrare il codice e tenerlo sempre funzionante

Integrato con molti IDE (eclipse, netbeans, ...)

L'ultima versione 4 sfrutta le annotation di Java

- useremo questa versione, codice molto più semplice

Mini Esempio con JUnit 4

- Un caso di test consiste in una classe ausiliaria
  - con alcuni metodi annotati `@Test`
  - in cui si controlla la corretta esecuzione del codice da testare mediante istruzioni come `assertEquals`

Esempio per testare l'operazione *:

```
...
public class HelloWorld {
@Test public void testMultiplication() {
      //Testing if 2*2=4:
      assertEquals("Mult", 4, 2*2);
}}
```

Unit test di una classe

Si voglia testare una classe Counter che rappresenta un contatore

le operazioni di Counter sono:

- **il costruttore che crea un contatore e lo setta a 0**
- **il metodo inc che incrementa il contatore e restituisce il nuovo valore incrementato**
- **il metodo dec che decrementa il contatore e restituisce il nuovo valore**

Schema della classe

Dovremmo scrivere prima il caso di test e poi la classe Counter, per chiarezza scriviamo prima lo scheletro di Counter:

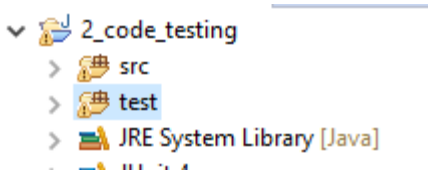
```
public class Counter {
      public Counter(){}
      public int inc() {}
      public int dec() {}
}
```

Test di una classe

Per testare una classe X si crea una classe ausiliaria (in genere con nome XTest)

XTest contiene dei metodi annotati con @Test che rappresentano i casi di test

Nota: E' meglio creare le clasi di test in una source directory separata (ma poi gli stessi package).
Ad esempio
In src metto il codice Java principale
In test metto i casi di test



Testare Counter

Per testare l'unità Counter, in particolare i metodi inc e dec, creiamo una nuova classe CounterTest (o qualsiasi nome) così:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
   @Test public void testInc() {}
   @Test public void testDec() {}
}
```

import delle istruzioni di assert

annotazioni per dire che sono metodi che testano altri metodi

Test di un metodo

Nel singolo metodo di test testiamo ogni metodo della classe sotto test

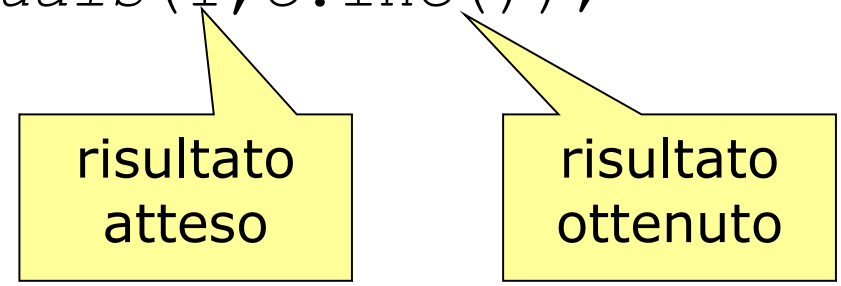
Dobbiamo:

1. creare eventuali oggetti delle classe sottotest
2. chiamare il metodo da testare e ottenere il risultato
3. confrontare il risultato ottenuto con quello atteso
   - **per far questo usiamo dei metodi assert di Junit che permettono di eseguire dei controlli**
   - **se un assert fallisce, JUnit cattura il fallimento e lo comunica al tester**
4. ripetiamo da 1

Test di inc()

In `testInc()` creiamo un'istanza di Counter, lo incrementiamo con il metodo `inc()` e controlliamo (mediante l'istruzione `assertEquals`) che inc() funzioni correttamente:

```
@Test
public void testInc() {
      Counter c = new Counter();
      assertEquals(1,c.inc());
}
```



Metodi assert

Ci sono molti metodi assert:

- **assertEquals**(*expected*, *actual*)
- **assertEquals**(String ***message***, *exp*, *act*)
  - **per controllare l'uguaglianza (con equals) di exp e act**
- **assertTrue**(expression)
  - per controllare che expression sia vera
- **assertNull**(Object ***object***)
- **fail**() e **fail**(String ***message***)
  - per terminare con un fallimento
- **assertSame**
  - per usare == invece che equals per il confronto

Assert methods II

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - This method is heavily overloaded: *arg1* and *arg2* must be both objects *or* both of the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as `public boolean equals(Object o)`-- otherwise it uses `==`

Assert methods II

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects refer to the same object (using ==)

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects do not refer to the same object

Assert methods III

- **assertNull(Object *object*)**
  assertNull(String *message*, Object *object*)
  - Asserts that the object is null

- **assertNotNull(Object *object*)**
  assertNotNull(String *message*, Object *object*)
  - Asserts that the object is null

- **fail(), fail(String *message*)**
  - Causes the test to fail and throw an AssertionFailedError
  - Useful as a result of a complex test, when the other assert methods aren't quite what you want

The (java) assert statement

- **Earlier versions of JUnit had an assert method instead of an assertTrue method**
  - The name had to be changed when Java 1.4 introduced the assert statement

- **L'istruzione assert di Java può essere abilitata dalle preferenze di Junit in eclipse o aggiungendo -ea**

The assert statement

- **There are two forms of the assert statement:**
  - assert ***boolean_condition***;
  - assert ***boolean_condition***: ***error_message***;
  - Both forms throw an AssertionFailedError if the ***boolean_condition*** is false
  - The second form, with an explicit error message, is seldom necessary
- **When to use an assert statement:**
  - Use it to document a condition that you "know" to be true
  - Use assert false; in code that you "know" cannot be reached (such as a default case in a switch statement)
  - Do **not** use assert to check whether parameters have legal values, or other places where throwing an Exception is more appropriate
  - Avoid side effects in assert

Costruttore

```
public class CounterTest {
   Counter counter1;

    public CounterTest() {
       counter1 = new Counter();
    }

   @Test public void testIncrement() {
       assertTrue(counter1.increment() == 1);
       assertTrue(counter1.increment() == 2);
     }

   @Test public void testDecrement() {
       assertTrue(counter1.decrement() == -1);
    }
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

BeforeClass

- **Se un metodo deve essere eseguito una sola volta prima dei test usa:**

  @BeforeClass

  - Il metodo deve essere statico e public

- **Esempio:**

```
@BeforeClass public static void onlyOne(){
...
}
```

- **Esistono anche metodi che vengono eseguiti prima di ogni test**

```
@Before
```

JUnit in Eclipse (1)

E' consigliabile usare un IDE come eclipse che assiste nella scrittura ed esecuzione dei casi di test

Per scrivere una caso di test:

1. scrivi la tua classe al solito (almeno lo scheletro)
2. seleziona la classe per cui vuoi creare i casi di test con il tasto destro -> new -> JUnit Test Case
3. appare un dialogo: selezione JUnit 4 e deseleziona tearDown, setUp ... - non sono necessari per piccoli esercizi

JUnit in Eclipse (2)

1. fai next -> seleziona i metodi per cui vuoi creare i casi di test
2. riempi i metodi di test con il codice che faccia i controlli opportuni

Per eseguire un caso di test in eclipse:

- tasto destro sulla classe di test -> run As -> Junit Test
- appare un pannello con una barra che rimane verde se tutto va bene

eccezioni

- È possibile verificare la presenza di eccezioni, quando un metodo **deve** generare un'eccezione

```
@Test(expected=Exception.class)
public void verificaEccezione(){ …}
```

- Se un metodo non deve generare un'eccezione basterà non dire nulla (se non ci sono eccezioni controllate) o aggiungerle nel throws:

  public void maiEccezione() throws Exp { …}

- Se si verifica l'eccezione, il test fallisce

Controllare Eccezioni

- **Quando un metodo di test deve lanciare una eccezione e poi continuare puoi fare così:**

```
try{
  istruzioneConEccObbligatoria
  fail("non ha lanciato eccezione !!!);
} catch(XXXXException e){
} catch(Exception e){
  fail("ha lanciato una ecc diversa");
}
continua il test
```

Test parametrici con Junit

- Alcune volte si vuole chiamare lo stesso test con alcuni dati
  - Il codice che effettua il test non cambia
  - I dati di ingresso e il controllo sì

- Esempio:
  - Un metodo incrementa inc(x), voglio testarlo con 0, 2 e 5 (e controllare che dia 1,3, 6)
  - Invece che scrivere 3 tests, posso utilizzare i test parametrici
    - Ho due parametri: input e inputIncrementato

Test parametrici (2)

- si dichiarano tante variabili d'istanza quanti sono i parametri usati nel test
  - es. int nLetter;...
- si crea un costruttore del test che ha per parametri la n- upla identica alle variabili d'istanza
  - Es Test(int n){nLetter = n; ...}
- si crea un metodo statico @Parameters che deve restituire una Collection (contiene le n-uple di parametri con i valori)
- si annota la classe con @RunWith(Parameterized.class)

Esempio

- **Voglio testare il metodo**
  - int incrementa(int x){return x+1;}

- **Voglio farlo per un qualche n:**
  - [0,2,5]
  - Output atteso: [1,3,6]
- **Qualcosa come**

```
@Test
 public void testSomma .... input (int) – outputAtteso (int)
 int outputAttuale = incrementa(input);
 assertEquals(outputAtteso,outputAttuale);}
}
```

```
 // runner particolare
@RunWith(Parameterized.class)
 // caso di test
 public class ParameterTest{
   // parametri
   private int input;
   private int outputAtteso;
   // construttore
   public ParameterTest(int p1, int p2){
       input = p1; outputAtteso = p2; }

 @Test
  public void testParametrico(){ ..//qui uso
  int outputAttuale = incrementa(input);
  assertEquals(inputIncrementato,outputAttuale);}
 }
```

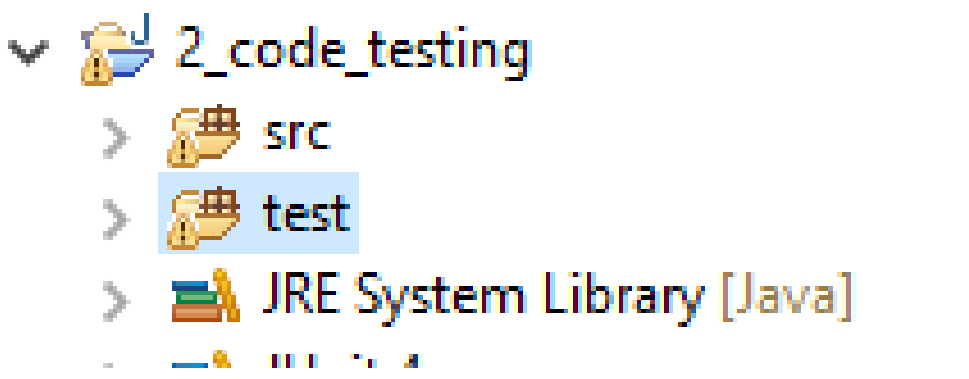
- Per settare i parametri nel costruttore:

```
@Parameters
public static Collection<Object[]> creaParametri() {
return Arrays.asList(new Object[][] {
{ 0, 1 },
{ 2, 3 },
{ 5, 6 }});
}
```

Ogni elemento della collezione è un array di object che serve per costruire il test (viene convertito automaticamente)

Come separare i test dal codice

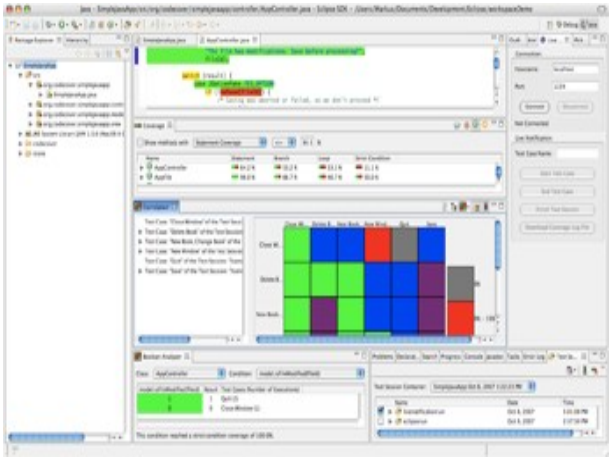
- **Puoi mettere i casi di test in un folder separati**
- **Anche se negli stessi package**
  - Così puoi testare I metodi friendly
- **Soluzione: Crea un source folder "test" e metti I casi di test lì dentro**



In sintesi

- Abbiamo visto che JUnit:
  - fa parte della metodologia eXtreme Programming;
  - adotta lo sviluppo guidato dai test;
  - è completamente automatico;
- Ricordate che in JUnit:
  - per testare una classe utilizzo un’altra classe
  - per testare i metodi uso dei metodi annotati @Test;
  - nei metodi di test eseguo i metodi da testare
  - e controllo la corretta esecuzione con gli assert
- Infine:
  - eclipse supporta JUnit 4

CODE COVERAGE



Esempi

JaCoCo > org.jacoco.report

org.jacoco.report

Element	Instruction Coverage		Missed Classes	Missed Methods	Missed Blocks	Missed Lines
org.jacoco.report.html		63%	10 / 20	54 / 128	84 / 214	117 / 385
org.jacoco.report.csv		20%	8 / 9	35 / 43	52 / 68	100 / 126
org.jacoco.report		75%	3 / 7	6 / 25	12 / 69	26 / 98
org.jacoco.report.xml		90%	2 / 10	9 / 42	13 / 88	17 / 146
org.jacoco.report.html.resources		87%	1 / 3	1 / 7	9 / 40	2 / 35
Total		64%	24 / 49	105 / 245	170 / 479	262 / 790

Code Coverage Report for JaCoCo 0.1.0.200910271744 26

Created with JaCoCo 0.1.0.200910271744 26

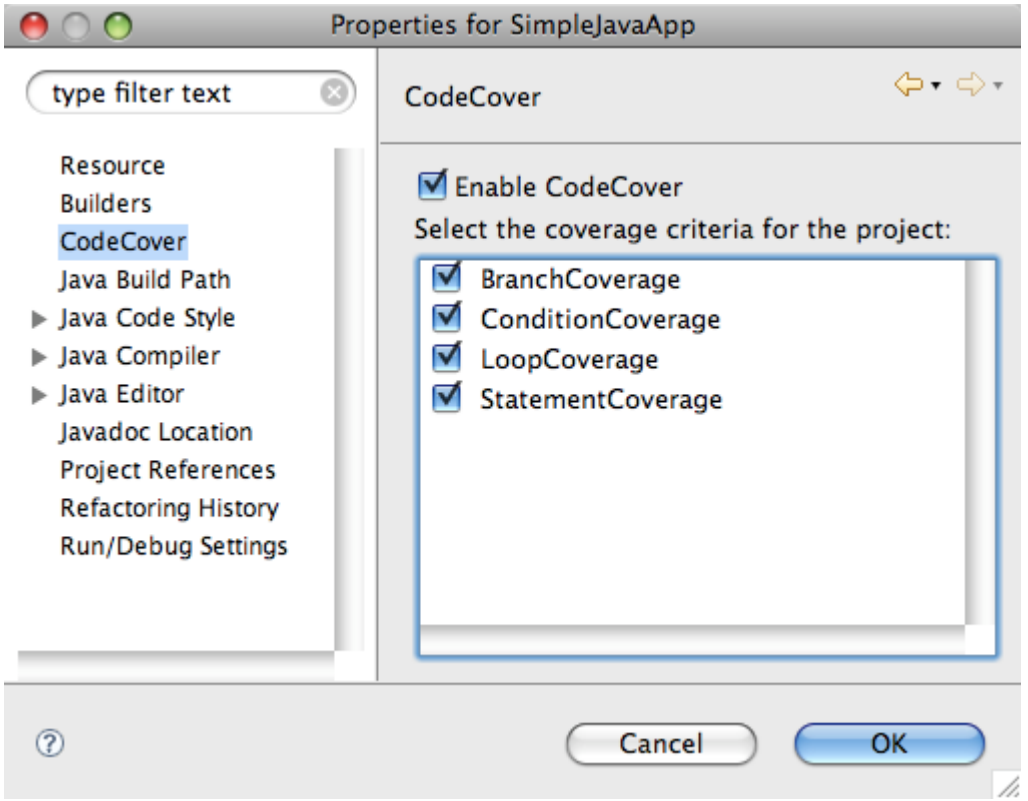
Installation

- **CodeCover Standalone (Batch and Ant support)**
  - For executing codecover outside eclipse
- **CodeCover Eclipse Plugin**

Enable CodeCover

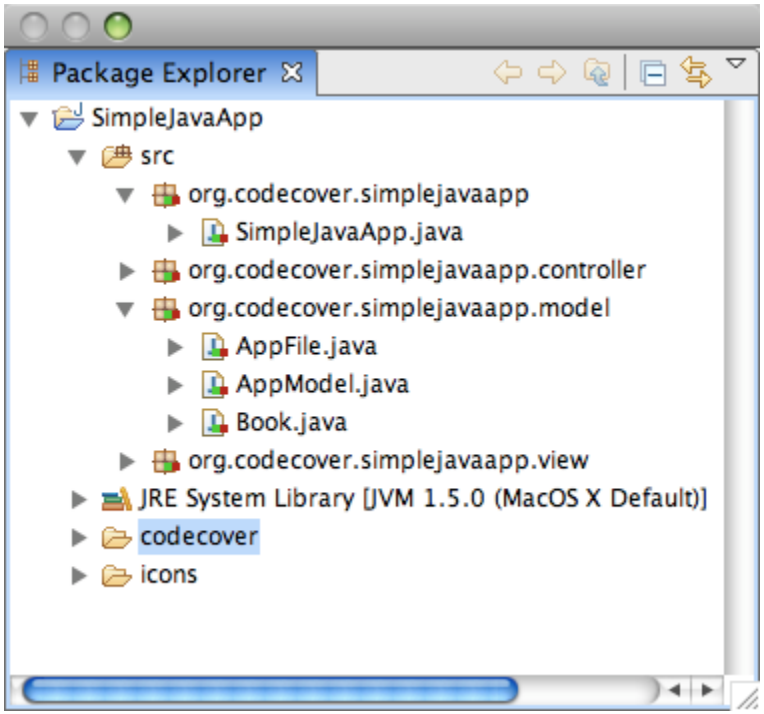
- You need to enable CodeCover for the imported project to measure its coverage.

  - Open the project properties dialog of the project and navigate to the CodeCover category.
  - You also need to select the coverage criteria to be used in the instrumentation. In this case all available criteria were selected.



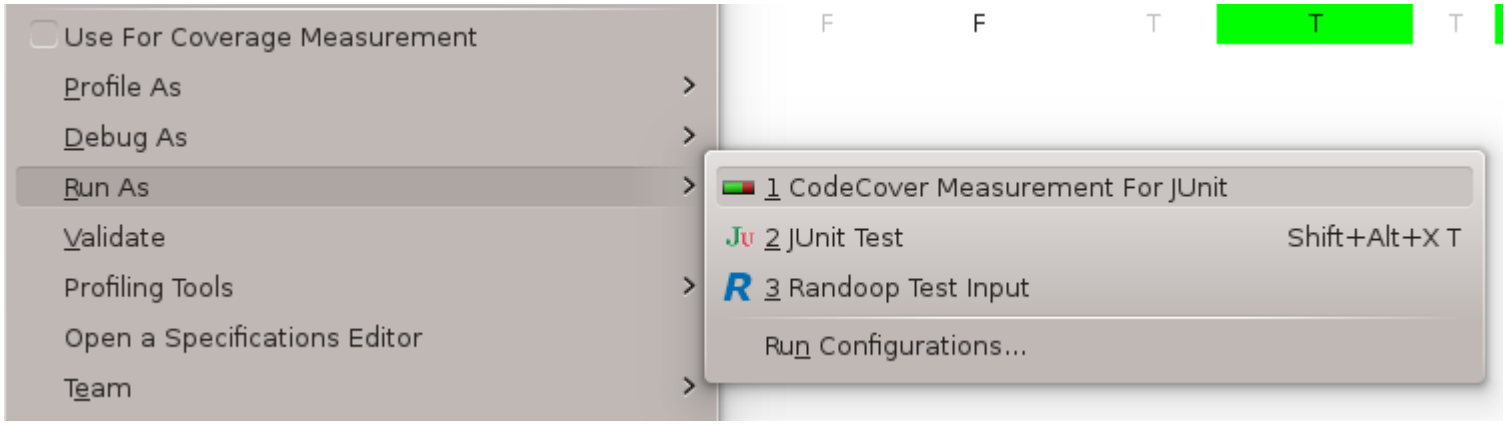
Use for Coverage Measurement

•Select the classes you want to instrument. Open the package explorer view, navigate to the source folder of the project and select those classes you want to be instrumented. Open the context menu of this selection and select the "Use For Coverage Measurement" entry. This will mark the selected items – as shown in the screenshot below – with an icon.



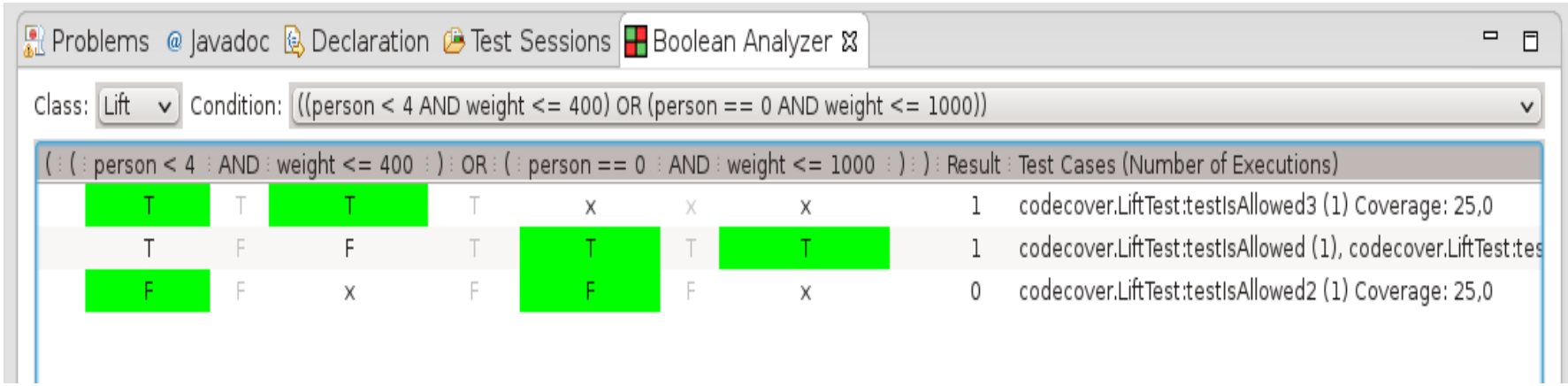
JUnit execution

- To use your existing test suite you need to create a new "CodeCover Measurement For JUnit" run configuration. After terminating, the measurements are automatically stored in a test session called "eclipserun", which holds all the test cases your test suite defined.



View measured data in Eclipse

- **Sessions/Coverage View/Correlation View**
- **Boolean Analyzer**



Automatic Test generation

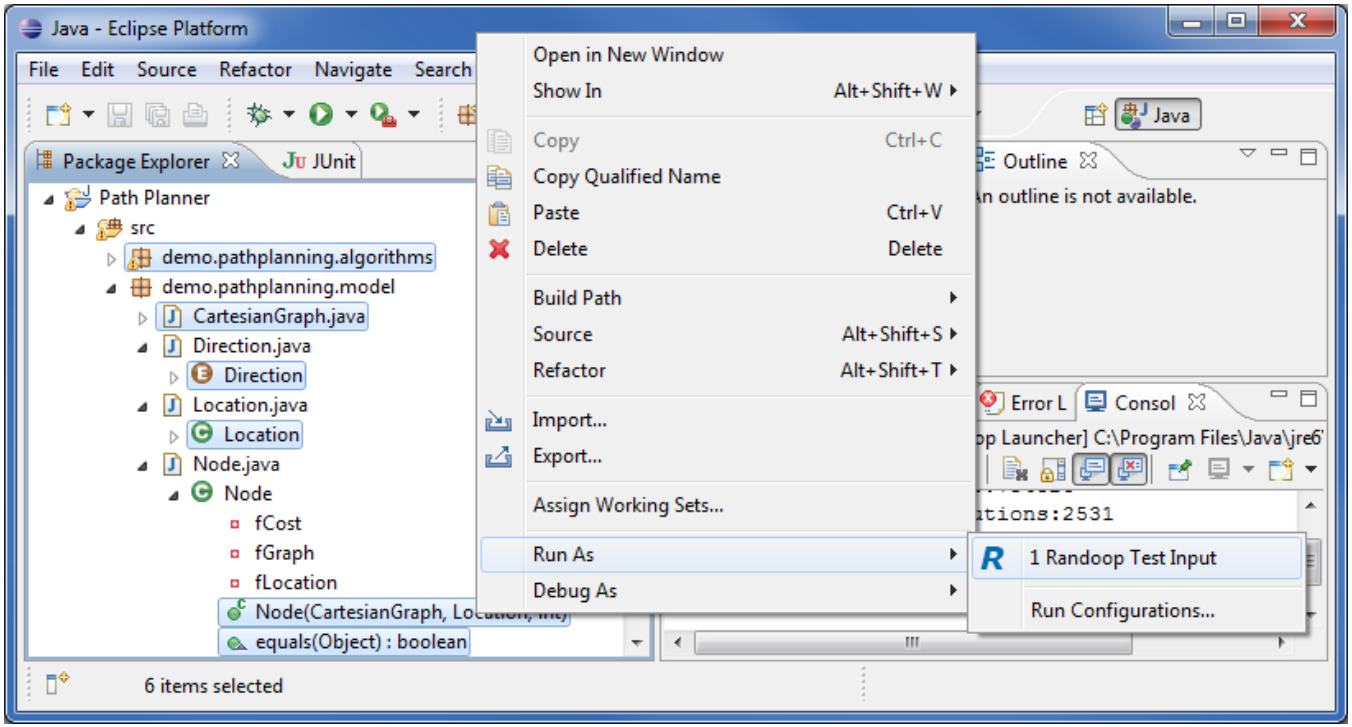
Generazione automatica

Si possono utilizzare dei tool per la generazione automatica di casi di test

- Lo scopo è quello di coprire i metodi testati in modo da raggiungere una copertura sufficiente.
- Scoprire possibili difetti
- Produrre automaticamente anche l'oracolo
  - che sia in grado di dire se il risultato è corretto
  - Mediante introduzione di assert
- Utile: per regression testing
  - So che un metodo funziona, genero i casi di test e poi lo modifico (miglioro) e ricontrollo
- Probelmi: creazione degli oggetti ...

Tecniche random

- **Creo degli oggetti e degli input random**
  - http://code.google.com/p/randoop/
- **Esempio:**
- **Randoop- demo**
-



Search based Soft. Engineering

•To find defects in software, one needs test cases that execute the software systematically, and oracles that assess the correctness of the observed behavior when running these test cases. EvoSuite is a tool that automatically generates test cases with assertions for classes written in Java code. To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behaviour.

Testing e contiunous integration

- **Lo vediamo in classe**