SE 342 Course Notes - Lecture 9

Testing Theory and Practice

Dr. Uzay Çetin

Maltepe University
Software Engineering Department

April 19, 2017

Outline

- Testing in Action
  - JUnit
  - Building Test Cases
- Graph Coverage
  - Control Flow Graphs
  - Coverage Tools: EclEmma

Examples taken from the book, Introduction to Software Testing, Paul Ammann and Jeff Offutt

Central Notion: Test Coverage

Technically, software testing is based on satisfying a coverage criteria

- Graphs
- Logical Expressions
- Input Space
- Syntax structures

Bezier's insight: "Find a graph and cover it"

- Testing = debugging
- Testing shows software works
- Testing shows software does not work
- Testing can only reduce the risk, but can not quarantee correctness
- Testing is a mental discipline to improve quality

Integration Fault

Mars Lander of Sept. 1999. Two software groups working with different units of measure, (feet vs meter).

Testing in Action

Testing should be a collection of objective, quantitative activities that can be measured and repeated.

- Design test inputs
- Produce test case values
- Run test scripts
- Analyse results
- Report

JUnit Annotations

Table 1. Annotations

JUnit 4	Description
`import org.junit.*`	Import statement for using the following annotations.
`@Test`	Identifies a method as a test method.
`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.
`@Test (expected = Exception.class)`	Fails if the method does not throw the named exception.
`@Test(timeout=100)`	Fails if the method takes longer than 100 milliseconds.

Ref: www.tutorialspoint.com

JUnit Assertions

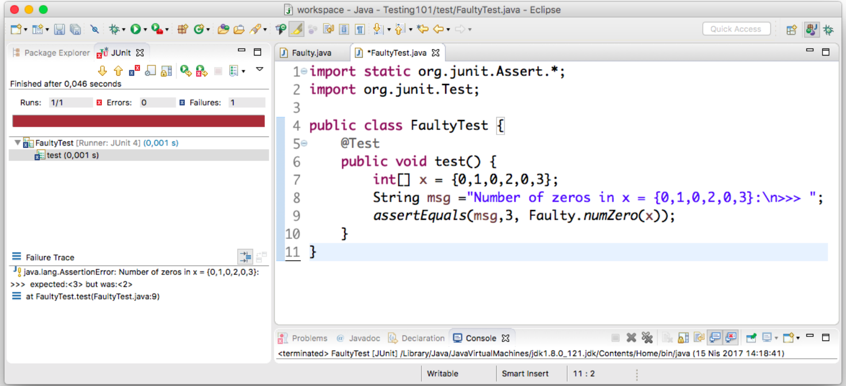
Table 2. Methods to assert test results

Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

Ref: www.tutorialspoint.com

Testing in Action

```
public class Faulty {
   public static int numZero (int[] x) {
      // Effects: if x == null throw NullPointerException
      // else return the number of occurrences of 0 in x
      int count = 0;/* I1 */
      for (int i = 1/* I2 */; i < x.length;/* I3 */ i++ /* I4 */){
         if (x[i] == 0) /* I5 */
         {
         count++; /* I6 */
         }
      }
      return count; /* I7 */
   }
}
```


workspace - Java - Testing101/test/FaultyTest.java - Eclipse
Package Explorer
JUnit
Finished after 0,046 seconds
Runs: 1/1
Errors: 0
Failures: 1
FaultyTest [Runner: JUnit 4] (0,001 s)
test (0,001 s)
Failure Trace
java.lang.AssertionError: Number of zeros in x = {0,1,0,2,0,3}:
>>> expected:<3> but was:<2>
at FaultyTest.test(FaultyTest.java:9)
Faulty.java
*FaultyTest.java
import static org.junit.Assert.*;
import org.junit.Test;
public class FaultyTest {
@Test
public void test() {
int[] x = {0,1,0,2,0,3};
String msg ="Number of zeros in x = {0,1,0,2,0,3}:\n>>> ";
assertEquals(msg,3, Faulty.numZero(x));
}
}
Problems
Javadoc
Declaration
Console
<terminated> FaultyTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (15 Nis 2017 14:18:41)
Writable
Smart Insert
11 : 2

Testing in Action

Faults are design mistakes, root causes of failures.

- Fault: Wrong indexing
  - Java indexing is zero-based.
  - $i = 1$, i should have started from 0.

Error: An internal (hidden) incorrect state due to faults.

- For the first iteration, internal value of i should be zero.

Failure: An external (open) incorrect behaviour due to faults.

- Expected $< 3 >$ but was $< 2 >$

Testing in Action

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

public class FaultyTest2 {
   @Ignore // This test will be ignored
   @Test
   public void test() {
      int[] x = {0,1,0,2,0,3};
      String msg ="Number of zeros in x = {0,1,0,2,0,3}:\n>>> ";
      assertEquals(msg,3, Faulty.numZero(x));
   }
   // Test will pass if exception is raised
   @Test(expected=NullPointerException.class)
   public void testNull() {
      int[] x = null;
      String msg ="Number of zeros in non-existing x:\n>>> ";
      assertEquals(msg,null, Faulty.numZero(x));
   }
}
```

Testing in Action

Test Case	input	Expected Output	Actual Output	Pass	LC	IC	BC
t1	[0,1,0,2,0,3]	3	2	NO	5,6	I1,I2, I3	NPE-B1
t2	null	NPE	NPE	YES	all	all	B1, !B1,B2, !B2

Here,

- LC: Line coverage
- IC: Instruction coverage
- BC: Branch coverage
- NPE-B1: Null pointer exception
- B1: $i < x.length$
- B2: $(x[i] == 0)$

More tests

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
  for (int i=x.length-1; i > 0; i--)
  {
    if (x[i] == y)
    {
      return i;
    }
  }
  return -1;
}
  // test:  x=[2, 3, 5]; y = 2
  //        Expected = 0
```

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in x

  for (int i = 0; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
  // test:  x=[0, 1, 0]
  //        Expected = 2
```

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in x.
  int count = 0;
  for (int i=0; i < x.length; i++)
  {
    if (x[i] >= 0)
    {
      count++;
    }
  }
  return count;
}
  // test:  x=[-4, 2, 0, 2]
  //        Expected = 2
```

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//       are either odd or positive (or both)
  int count = 0;
  for (int i = 0; i < x.length; i++)
  {
    if (x[i]% 2 == 1 || x[i] > 0)
    {
      count++;
    }
  }
  return count;
}
  // test:  x=[-3, -2, 0, 1, 4]
  //        Expected = 3
```

Test Case

A test case is composed of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test.

- coverage is a property of a set of test cases

Test Requirement, TR

Test Requirement, TR

- A test requirement is a specific element of a software artifact that a test case must satisfy or cover.

Coverage Criterion, C

- A coverage criterion is a rule or collection of rules that impose test requirements on a test set.

Coverage

- Given a set of test requirements TRfor a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr .

Non-software example

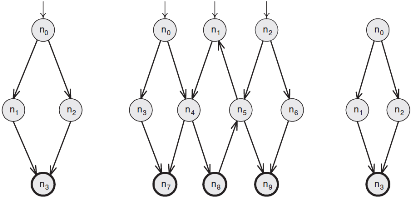
Coverage Level

- Given a set of test requirements TR and a test set T, the coverage level is simply the ratio of the number of test requirements satisfied by T to the size of TR.

Generic View of Graphs

A graph G is

- A set of N nodes
  - A set of $N_0 \subseteq N$ initial nodes,
  - A set of $N_f \subseteq N$ final nodes,
- A set E of edges, where E is a subset of $N \times N$



$N = \{ n_0, n_1, n_2, n_3 \}$
$N_0 = \{ n_0 \}$
$E = \{ (n_0, n_1), (n_0, n_2), (n_1, n_3), (n_2, n_3) \}$
(a) A graph with a single initial node

$N = \{ n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9 \}$
$N_0 = \{ n_0, n_1, n_2 \}$
$|E| = 12$
(b) A graph with mutiple initial nodes

$N = \{ n_0, n_1, n_2, n_3 \}$
$|E| = 4$
(c) A graph with no initial node

Generic View of Graphs

A path

- sequence $[n_1, n_2, ..., n_M]$ of nodes, where $(n_i, n_{i+1}) \in E$, $i \in [1, M]$

A test path

- represents the execution of a test case.
- starts form one node in N_0 and ends at some node in N_f

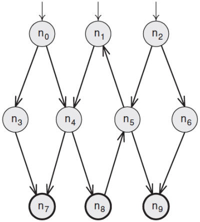


Figure 2.2. Example of paths.

Path Examples	
1	n_0, n_3, n_7
2	n_1, n_4, n_8, n_5, n_1
3	n_2, n_6, n_9

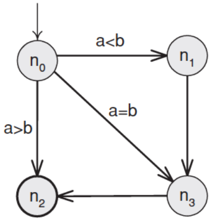
Invalid Path Examples	
1	n_0, n_7
2	n_3, n_4
3	n_2, n_6, n_8

(a) Path examples

Reachability Examples	
1	$reach\ (n_0) = N - \{n_2, n_6\}$
2	$reach\ (n_0, n_1, n_2) = N$
3	$reach\ (n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	$reach\ ([n_6, n_9]) = \{n_9\}$

(b) Reachability examples

A set of test cases and corresponding test paths.



(a) Graph for testing the case with input integers a, b and output (a+b)

Map to

Test case t_1 : (a=0, b=1) → [Test path p_1 : n_0, n_1, n_3, n_2]

Test case t_2 : (a=1, b=1) → [Test path p_2 : n_0, n_3, n_2]

Test case t_3 : (a=2, b=1) → [Test path p_3 : n_0, n_2]

(b) Mapping between test cases and test paths

Graph Coverage Criteria

Graph Coverage:

- Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in path(T) such that p meets tr .

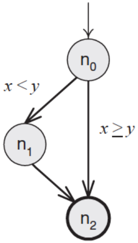
Node Coverage (NC):

- TR contains each reachable node in G.

Edge Coverage (EC)

- TR contains each reachable path of length up to 1, inclusive, in G.

Node Coverage vs Edge Coverage



path $(t_1) = [\, n_0, n_1, n_2 \,]$
path $(t_2) = [\, n_0, n_2 \,]$

$T_1 = \{ t_1 \}$
T_1 satisfies node coverage on the graph

(a) Node Coverage

$T_2 = \{ t_1 , t_2 \}$
T_2 satisfies edge coverage on the graph

(b) Edge Coverage

Node Coverage vs Edge Coverage

Complete path coverage is useless if a graph has a cycle, since this results in an infinite number of paths, and hence an infinite number of test requirements.

Node Coverage vs Edge Coverage

Simple Path

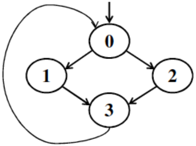
- A path from n_i to n_j is simple if no node appears more than once in the path, with the exception that the first and last nodes may be identical.
  - No internal loops

Prime Path

- A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.

Simple Paths and Prime Paths

- **Simple Path** : ***A path from node ni to nj is simple if no node appears more than once, except possibly the first and last nodes are the same***
  - **No internal loops**
  - **Includes all other subpaths**
  - **A loop is a simple path**
- **Prime Path** : ***A simple path that does not appear as a proper subpath of any other simple path***

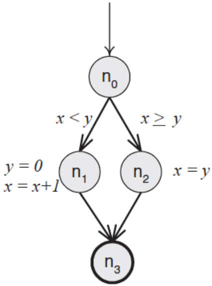


Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

Prime Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

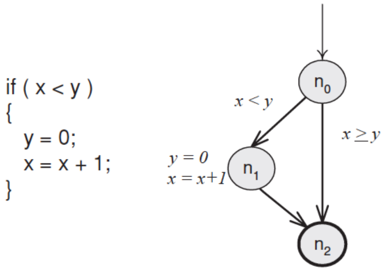
CFG of if-else structure

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



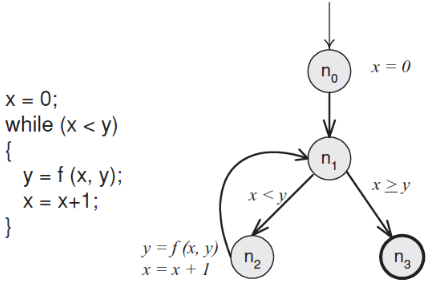
- n_0 : Decision node
- n_3 : Junction node

CFG of if structure



- n_0 : Decision node
- n_3 : Junction node

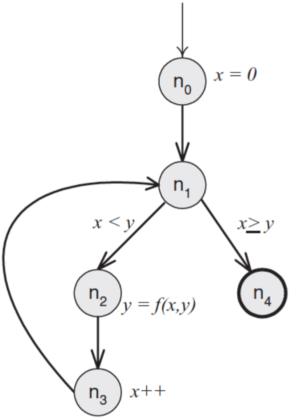
CFG of while loop



- n_1 : Dummy decision node for "while"

CFG of for loop

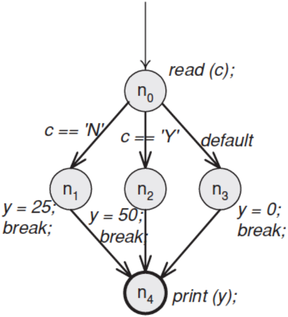
```
for (x = 0; x < y; x++)
{
   y = f(x,y);
}
```



- n_1 : Dummy decision node for "for"

CFG of for switch

```
read (c);
switch (c)
{
case 'N':
      y = 25;
      break;
case 'Y':
      y = 50;
      break;
default:
      y = 0;
      break;
}
print (y);
```



- n_1 : Dummy decision node for "switch"

Testing in Action

```java
public class Occur {
   public static int occurrences(char[] v, char c) {
      if (v == null) {
         throw new NullPointerException();
      }
      int n = 0;
      for (int i = 0; i < v.length; i++) {
         if (v[i] == c) {
            n++;
         }
      }
      return n;
   }
}
```

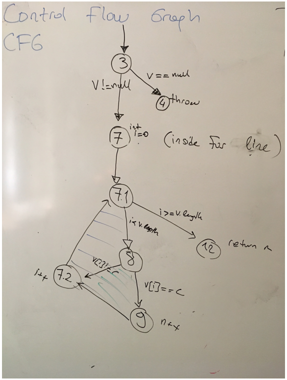
Testing in Action

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

import org.junit.Test;

public class TestOccur {

   @Test(expected=NullPointerException.class)
   public void t1() {
      char[] x = null;
      assertEquals(2,Occur.occurrences(x, 'a'));
   }

   @Test
   public void t2() {
      char[] x = "a".toCharArray();
      assertEquals(1,Occur.occurrences(x, 'a'));
   }

   @Test
   public void t3() {
      char[] x = "xa".toCharArray();
      assertEquals(1,Occur.occurrences(x, 'a'));
   }
}
```

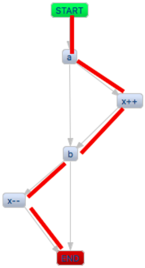
Testing in Action

t1 and t3 satisfies edge coverage!!

- $3 \rightarrow 4$ is not covered in t2 and t3.
- $8 \rightarrow 7.2$ is not covered in t2.

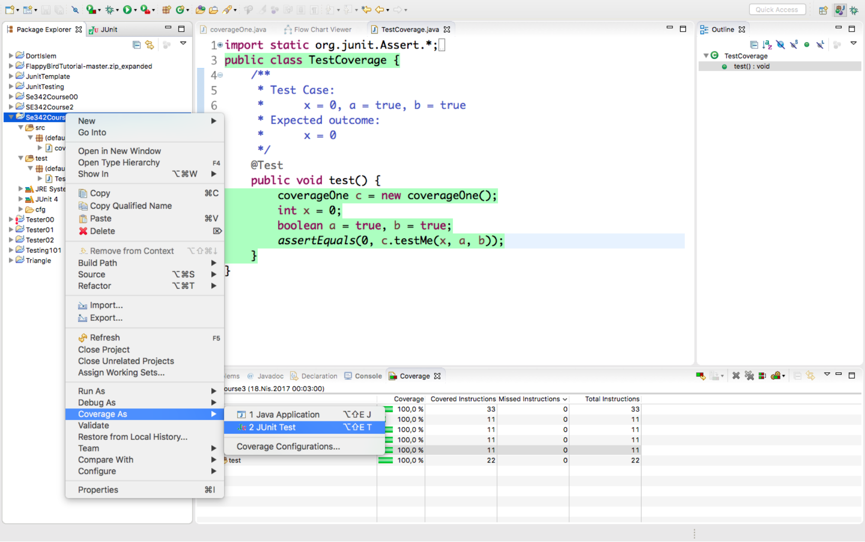


```java
import static org.junit.Assert.*;
import org.junit.Test;
public class TestCoverage {
    /**
     * Test Case:
     *       x = 0, a = true, b = true
     * Expected outcome:
     *       x = 0
     */
    @Test
    public void test() {
        coverageOne c = new coverageOne();
        int x = 0;
        boolean a = true, b = true;
        assertEquals(0, c.testMe(x, a, b));
    }
}
```

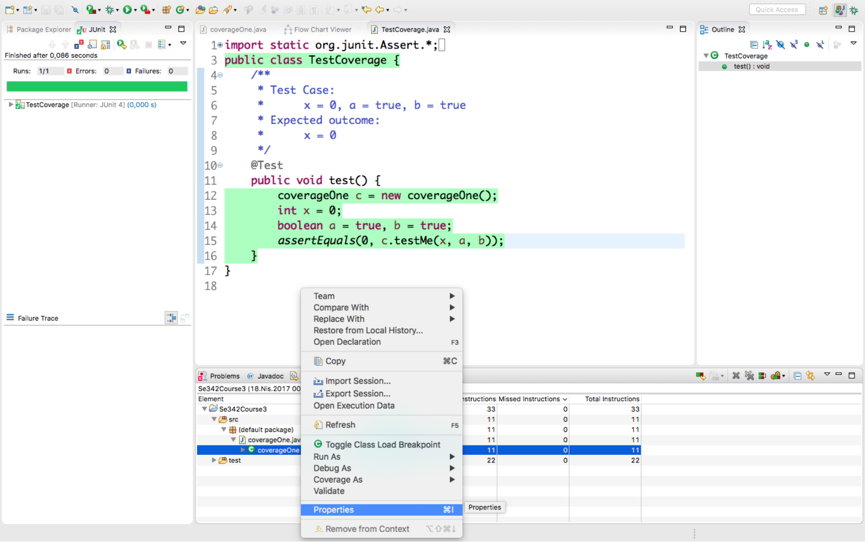


- Statement Coverage: %100
- Branch Coverage: %50
- Path Coverage: %25

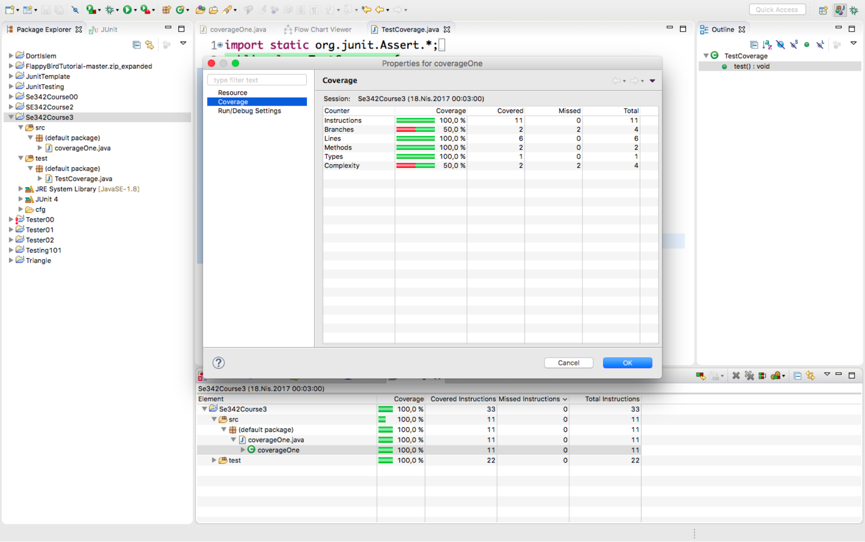
EclEmma

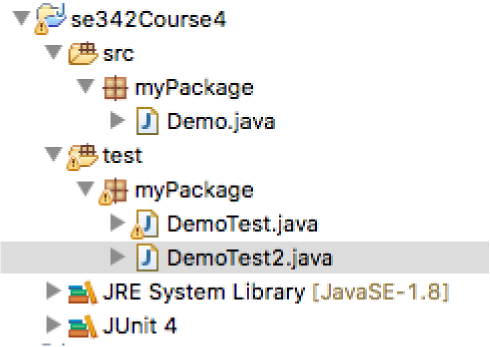

Package Explorer
JUnit
DortIslem
FlappyBirdTutorial-master.zip_expanded
JunitTemplate
JunitTesting
Se342Course00
SE342Course2
Tester00
Tester01
Tester02
Testing101
Triangle
New
Go Into
Open in New Window
Open Type Hierarchy
Show In
Copy
Copy Qualified Name
Paste
Delete
Remove from Context
Build Path
Source
Refactor
Import...
Export...
Refresh
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Coverage As
Validate
Restore from Local History...
Team
Compare With
Configure
Properties
1 Java Application
2 JUnit Test
Coverage Configurations...
coverageOne.java
Flow Chart Viewer
TestCoverage.java
import static org.junit.Assert.*;
public class TestCoverage {
/**
* Test Case:
* x = 0, a = true, b = true
* Expected outcome:
* x = 0
*/
@Test
public void test() {
coverageOne c = new coverageOne();
int x = 0;
boolean a = true, b = true;
assertEquals(0, c.testMe(x, a, b));
}
}
Outline
TestCoverage
test() : void
Javadoc
Declaration
Console
Coverage
Coverage
Covered Instructions
Missed Instructions
Total Instructions
100,0 %
Quick Access

EclEmma


Package Explorer
JUnit
Finished after 0,086 seconds
Runs: 1/1
Errors: 0
Failures: 0
TestCoverage [Runner: JUnit 4] (0,000 s)
Failure Trace
coverageOne.java
Flow Chart Viewer
TestCoverage.java
import static org.junit.Assert.*;
public class TestCoverage {
/**
* Test Case:
* x = 0, a = true, b = true
* Expected outcome:
* x = 0
*/
@Test
public void test() {
coverageOne c = new coverageOne();
int x = 0;
boolean a = true, b = true;
assertEquals(0, c.testMe(x, a, b));
}
}
Outline
TestCoverage
test() : void
Quick Access
Team
Compare With
Replace With
Restore from Local History...
Open Declaration F3
Copy ⌘C
Import Session...
Export Session...
Open Execution Data
Refresh F5
Toggle Class Load Breakpoint
Run As
Debug As
Coverage As
Validate
Properties ⌘I
Remove from Context
Properties
Problems
Javadoc
Se342Course3 (18.Nis.2017 00
Element
Missed Instructions
Total Instructions
Se342Course3 33 0 33
src 11 0 11
(default package) 11 0 11
coverageOne.jav 11 0 11
coverageOne 11 0 11
test 22 0 22

EclEmma


Package Explorer
JUnit
DortIslem
FlappyBirdTutorial-master.zip_expanded
JunitTemplate
JunitTesting
Se342Course00
SE342Course2
Se342Course3
src
(default package)
coverageOne.java
test
(default package)
TestCoverage.java
JRE System Library [JavaSE-1.8]
JUnit 4
cfg
Tester00
Tester01
Tester02
Testing101
Triangle
coverageOne.java
Flow Chart Viewer
TestCoverage.java
1⊕import static org.junit.Assert.*;
Properties for coverageOne
type filter text
Resource
Coverage
Run/Debug Settings
Coverage
Session: Se342Course3 (18.Nis.2017 00:03:00)
Counter Coverage Covered Missed Total
Instructions 100,0 % 11 0 11
Branches 50,0 % 2 2 4
Lines 100,0 % 6 0 6
Methods 100,0 % 2 0 2
Types 100,0 % 1 0 1
Complexity 50,0 % 2 2 4
Cancel
OK
Quick Access
Outline
TestCoverage
test() : void
Se342Course3 (18.Nis.2017 00:03:00)
Element Coverage Covered Instructions Missed Instructions Total Instructions
Se342Course3 100,0 % 33 0 33
src 100,0 % 11 0 11
(default package) 100,0 % 11 0 11
coverageOne.java 100,0 % 11 0 11
coverageOne 100,0 % 11 0 11
test 100,0 % 22 0 22


se342Course4
src
myPackage
Demo.java
test
myPackage
DemoTest.java
DemoTest2.java
JRE System Library [JavaSE-1.8]
JUnit 4

Testing in Action

```
package myPackage;

public class Demo {
   public static int add(int a, int b){
      return a - b;
   }

   public static int multiply(int a, int b){
      return a * b;
   }
}
```

Testing in Action

```
package myPackage;

import static org.junit.Assert.*;

import org.junit.Ignore;
import org.junit.Test;

public class DemoTest {
   /**
    * Test case: testAdd
    *       input : <a = 5, b = 3>
    *       Expected Outcome : 8
    *       Actual Outcome : 2
    *       Pass : No
    */
   @Test
   public void testAdd() {
      int a = 5;
      int b = 3;
      assertEquals(8,Demo.add(a, b));
   }
}
```

Testing in Action

```java
package myPackage;

import static org.junit.Assert.*;

import org.junit.Test;

public class DemoTest2 {
   /**
    * Test case: testMultiply
    *        input : <a = 5, b = 3>
    *        Expected Outcome : 15
    *        Actual Outcome : ?
    *        Pass : ?
    */
   @Test
   public void testMultiply() {
      int a = 5;
      int b = 3;
      assertEquals(15,Demo.multiply(a, b));
   }
}
```