Agile Software Development

Produced by

Dr. Siobhán Drohan (sdrohan@wit.ie)
Eamonn de Leastar (edeleastar@wit.ie)

First JUnit Tests (JUnit 3)

What is JUnit?

JUnit:

- is a Unit Testing Framework for Java.
- enables you to write and run repeatable tests.
- is used to Unit Test a small piece of code.

- When following TDD, developers should write and execute the JUnit tests before writing any code.

JUnit Versions

- **JUnit 3** (http://junit.sourceforge.net/junit3.8.1/)
- **JUnit 4** (http://junit.org/junit4/)
  - Version we will mainly use is 4.12
- **JUnit 5** (http://junit.org/junit5/)
  - First general availability release published on September 10, 2017. Bug fix version released on October 3, 2017.
  - Not supported directly in Eclipse IDE (you can use it via Maven, which we will show you in future weeks)

JUnit Version 3

- *As conventions differ between the versions, it is important to be able to use Version 3 and 4 (at least) and 5 (desirable).*

- In Version 3:

1. Test class must extend TestCase.

2. setUp/tearDown methods are overridden from TestCase (note that this is optional).

3. Test methods must begin with “test” word.

1. Test Class must extend TestCase

unit3.8.1/javadoc/index.html

Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

junit.framework

Class TestCase

```
java.lang.Object
  |
  +--junit.framework.Assert
        |
        +--junit.framework.TestCase
```

All Implemented Interfaces:

Test

Direct Known Subclasses:

ActiveTestTest, ActiveTestTest.SuccessTest, AssertTest, BaseTestRunnerTest, ComparisonFailureTest, DoublePrecisionAssertTest, ExceptionTestCase, ExceptionTestCaseTest, ExtensionTest, Failure, MoneyTest, NoArgTestCaseTest, NoTestCases, NotPublicTestCase, NotVoidTestCase, OneTestCase, RepeatedTestTest, RepeatedTestTest.SuccessTest, SimpleTest, SimpleTestCollectorTest, SorterTest, StackFilterTest, Success, SuiteTest, TestCaseClassLoaderTest, TestCaseTest, TestCaseTest.TornDown, TestImplementorTest, TestListenerTest, TextFeedbackTest, TextRunnerTest, VectorTest, WasRun

public abstract class **TestCase**
extends Assert
implements Test

A test case defines the fixture to run multiple tests. To define a test case
1) implement a subclass of TestCase
2) define instance variables that store the state of the fixture
3) initialize the fixture state by overriding `setUp`
4) clean-up after a test by overriding `tearDown`.

http://junit.sourceforge.net/junit3.8.1/javadoc/index.html

1. Test Class must extend TestCase

unit3.8.1/javadoc/index.html

Overview **Package** **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS | FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

junit.framework

Class TestCase

```
java.lang.Object
  |
  +--junit.framework.Assert
        |
        +--junit.framework.TestCase
```

All Implemented Interfaces:

Test

Direct Known Subclasses:

ActiveTestTest, ActiveTestTest.SuccessTest, AssertTest, BaseTestRunnerTest, ComparisonFailureTest, DoublePrecisionAssertTest, ExceptionTestCase, ExceptionTestCaseTest, ExtensionTest, Failure, MoneyTest, NoArgTestCaseTest, NoTestCases, NotPublicTestCase, NotVoidTestCase, OneTestCase, RepeatedTestTest, RepeatedTestTest.SuccessTest, SimpleTest, SimpleTestCollectorTest, SorterTest, StackFilterTest, Success, SuiteTest, TestCaseClassLoaderTest, TestCaseTest, TestCaseTest.TornDown, TestImplementorTest, TestListenerTest, TextFeedbackTest, TextRunnerTest, VectorTest, WasRun

---

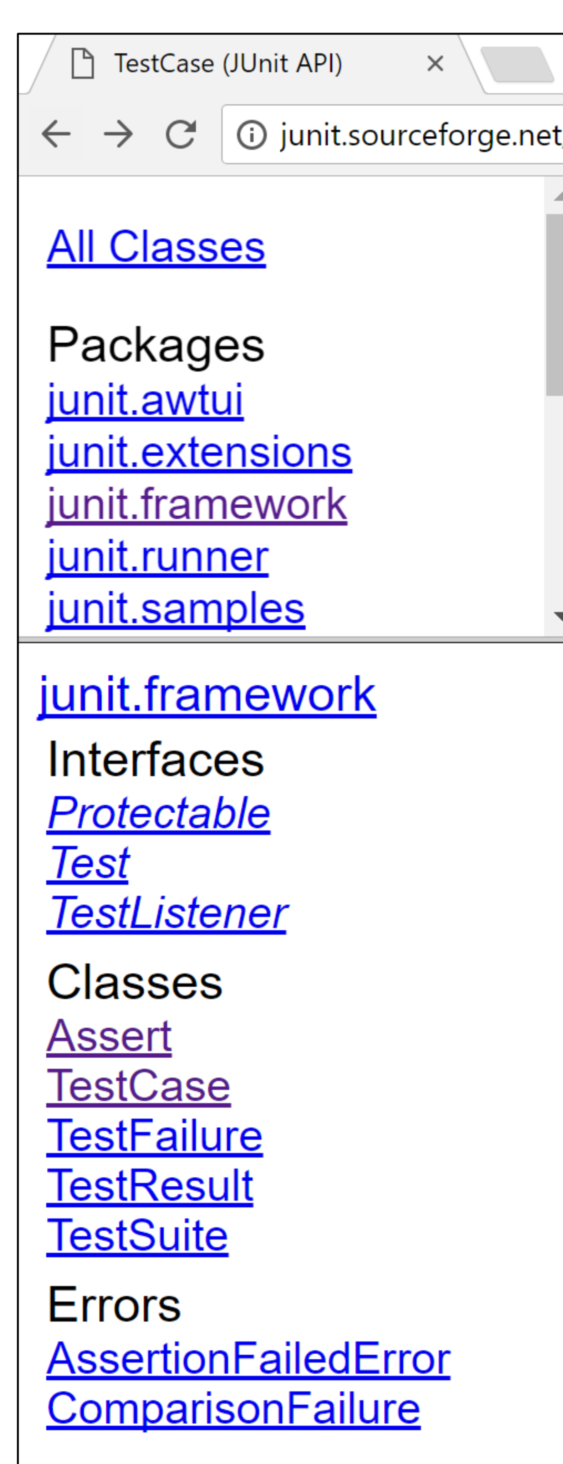
public abstract class **TestCase**
extends Assert
implements Test

A test case defines the fixture to run multiple tests. To define a test case
1) implement a subclass of TestCase
2) define instance variables that store the state of the fixture
3) initialize the fixture state by overriding `setUp`
4) clean-up after a test by overriding `tearDown`.

```
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
  //JUnit testing code omitted
}
```

http://junit.sourceforge.net/junit3.8.1/javadoc/index.html

2. setUp/tearDown methods are overridden from TestCase (note that this is optional).



Method Summary

int	countTestCases() Counts the number of test cases executed by run(TestResult result).
protected TestResult	createResult() Creates a default TestResult object
java.lang.String	getName() Gets the name of a TestCase
TestResult	run() A convenience method to run this test, collecting the results with a default TestResult object.
void	run(TestResult result) Runs the test case and collects the results in TestResult.
void	runBare() Runs the bare test sequence.
protected void	runTest() Override to run the test and assert its state.
void	setName(java.lang.String name) Sets the name of a TestCase
protected void	setUp() Sets up the fixture, for example, open a network connection.
protected void	tearDown() Tears down the fixture, for example, close a network connection.
java.lang.String	toString() Returns a string representation of the test case

http://junit.sourceforge.net/junit3.8.1/javadoc/index.html

3. Test methods must begin with "test" word.

```
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
   public TestLargest (String name)
   {
     super(name);
   }

   public void testOrder ()
   {
      int[] arr = new int[3];
      arr[0] = 8;
      arr[1] = 9;
      arr[2] = 7;
      assertEquals(9, Largest.largest(arr));
   }
}
```

Let's look at Assertions now…and then we will look at a JUnit testing a simple program.

```
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
   public TestLargest (String name)
   {
     super(name);
   }

   public void testOrder ()
   {
      int[] arr = new int[3];
      arr[0] = 8;
      arr[1] = 9;
      arr[2] = 7;
      assertEquals(9, Largest.largest(arr));
   }
}
```

Assertions

- To check if code is behaving as you expect
  - use an *assertion i.e.* a simple method call that verifies that something is true.

junit.framework

Class TestCase

```
java.lang.Object
  |
  +--junit.framework.Assert
        |
        +--junit.framework.TestCase
```

All Implemented Interfaces:

Test

Direct Known Subclasses:

ActiveTestTest, ActiveTestTest.SuccessTest, AssertTest, BaseTestRunnerTest, (
ExceptionTestCase, ExceptionTestCaseTest, ExtensionTest, Failure, MoneyTes
NotVoidTestCase, OneTestCase, RepeatedTestTest, RepeatedTestTest.SuccessT
StackFilterTest, Success, SuiteTest, TestCaseClassLoaderTest, TestCaseTest, T
TextFeedbackTest, TextRunnerTest, VectorTest, WasRun

---

public abstract class **TestCase**
extends Assert
implements Test

A test case defines the fixture to run multiple tests. To define a test case
1) implement a subclass of TestCase
2) define instance variables that store the state of the fixture
3) initialize the fixture state by overriding `setUp`
4) clean-up after a test by overriding `tearDown`.

junit.framework

Class Assert

```
java.lang.Object
  |
  +--junit.framework.Assert
```

Direct Known Subclasses:

ClassLoaderTest, LoadedFromJar, TestCase, TestDecorator

---

public class **Assert**
extends java.lang.Object

A set of assert methods. Messages are only displayed when an assert fails.

Some of the many "assertion" methods in the Assert class…

static void	assertEquals(java.lang.String expected, java.lang.String actual) Asserts that two Strings are equal.
static void	assertEquals(java.lang.String message, java.lang.String expected, java.lang.String actual) Asserts that two Strings are equal.
static void	assertFalse(boolean condition) Asserts that a condition is false.
static void	assertFalse(java.lang.String message, boolean condition) Asserts that a condition is false.
static void	assertNotNull(java.lang.Object object) Asserts that an object isn't null.
static void	assertNotNull(java.lang.String message, java.lang.Object object) Asserts that an object isn't null.
static void	assertNotSame(java.lang.Object expected, java.lang.Object actual) Asserts that two objects do not refer to the same object.
static void	assertNotSame(java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects do not refer to the same object.
static void	assertNull(java.lang.Object object) Asserts that an object is null.
static void	assertNull(java.lang.String message, java.lang.Object object) Asserts that an object is null.
static void	assertSame(java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.

Using Asserts

You could use this assert to check all sorts of things, including whether numbers are equal to each other.

```
int a = 2;
//...
assertTrue (a == 2);
```

Using Asserts

You could use this assert to check all sorts of things, including whether numbers are equal to each other.

```
int a = 2;
//...
assertTrue (a == 2);
```

To check that two integers are equal, a method that takes two integer parameters might be more useful.

```
public void assertEquals (int a, int b)
{
  assertTrue(a == b);
}
```

Using Asserts

You could use this assert to check all sorts of things, including whether numbers are equal to each other.

```
int a = 2;
//...
assertTrue (a == 2);
```

To check that two integers are equal, a method that takes two integer parameters might be more useful.

```
public void assertEquals (int a, int b)
{
  assertTrue(a == b);
}
```

We can now write the first test a little more expressively:

```
int a = 2;

assertEquals (2, a);
```

JUnit Example

Testing code to return the largest number in a Primitive Array.

Planning Tests

- Method to test: A static method designed to find the largest number in a list of numbers.

- The following tests would seem to make sense:

```
public static int largest (int[] list)
{
...
}
```

  - **[7, 8, 9]** → **9**
  - **[8, 9, 7]** → **9**
  - **[9, 7, 8]** → **9**

(**supplied test data** → **expected result**)

More Test Data + First Implementation

- Already have this data:

  [7, 8, 9] → 9
  [8, 9, 7] → 9
  [9, 7, 8] → 9

- What about this set:

  [7, 9, 8, 9] → 9
  [1] → 1
  [-9, -8, -7] → -7

```
public static int largest (int[] list)
{
  int index, max = Integer.MAX_VALUE;

  for (index = 0; index < list.length - 1; index++)
  {
     if (list[index] > max)
     {
        max = list[index];
     }
  }
  return max;
}
```

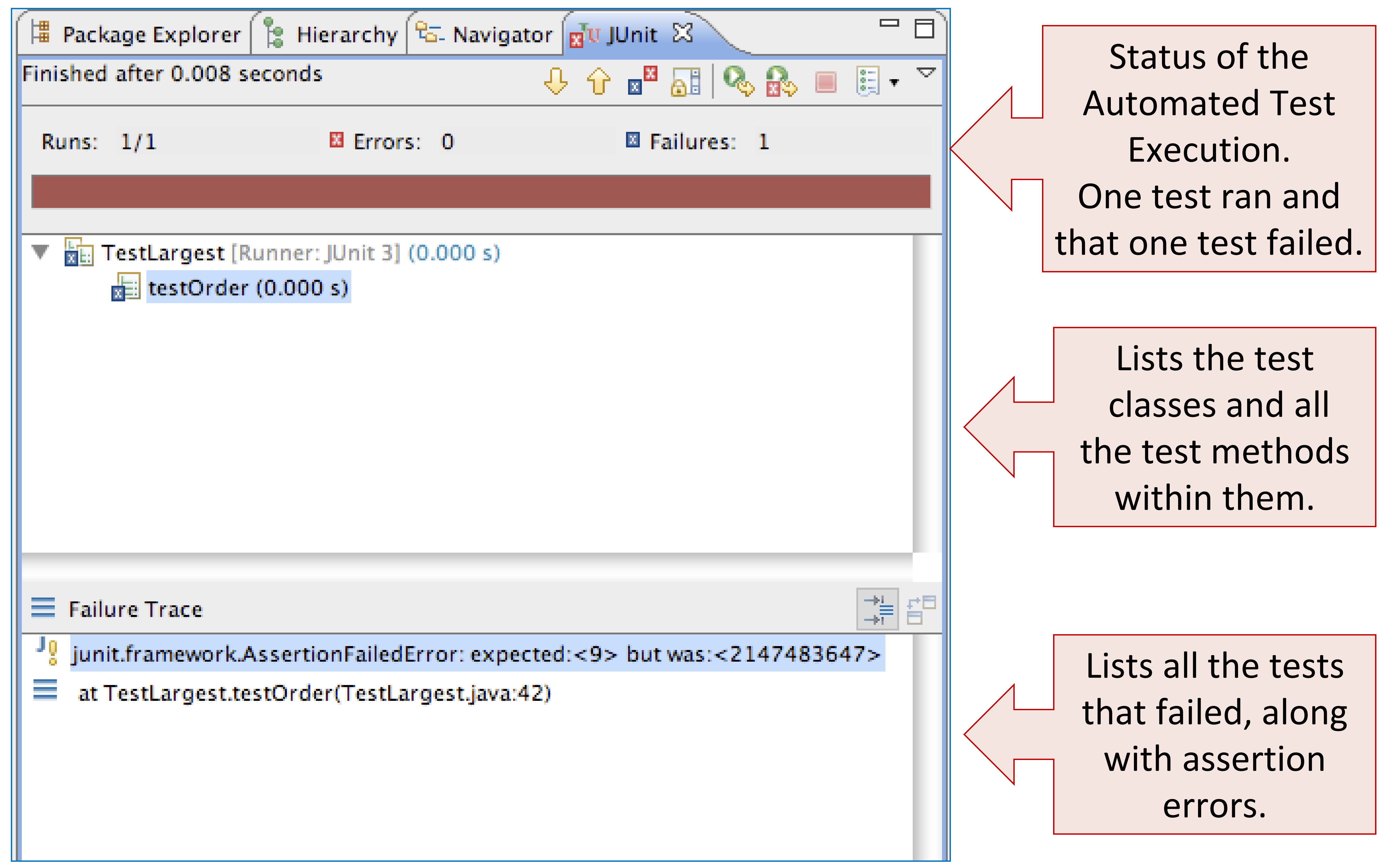
(**supplied test data** → **expected result**)

Writing the TestCase

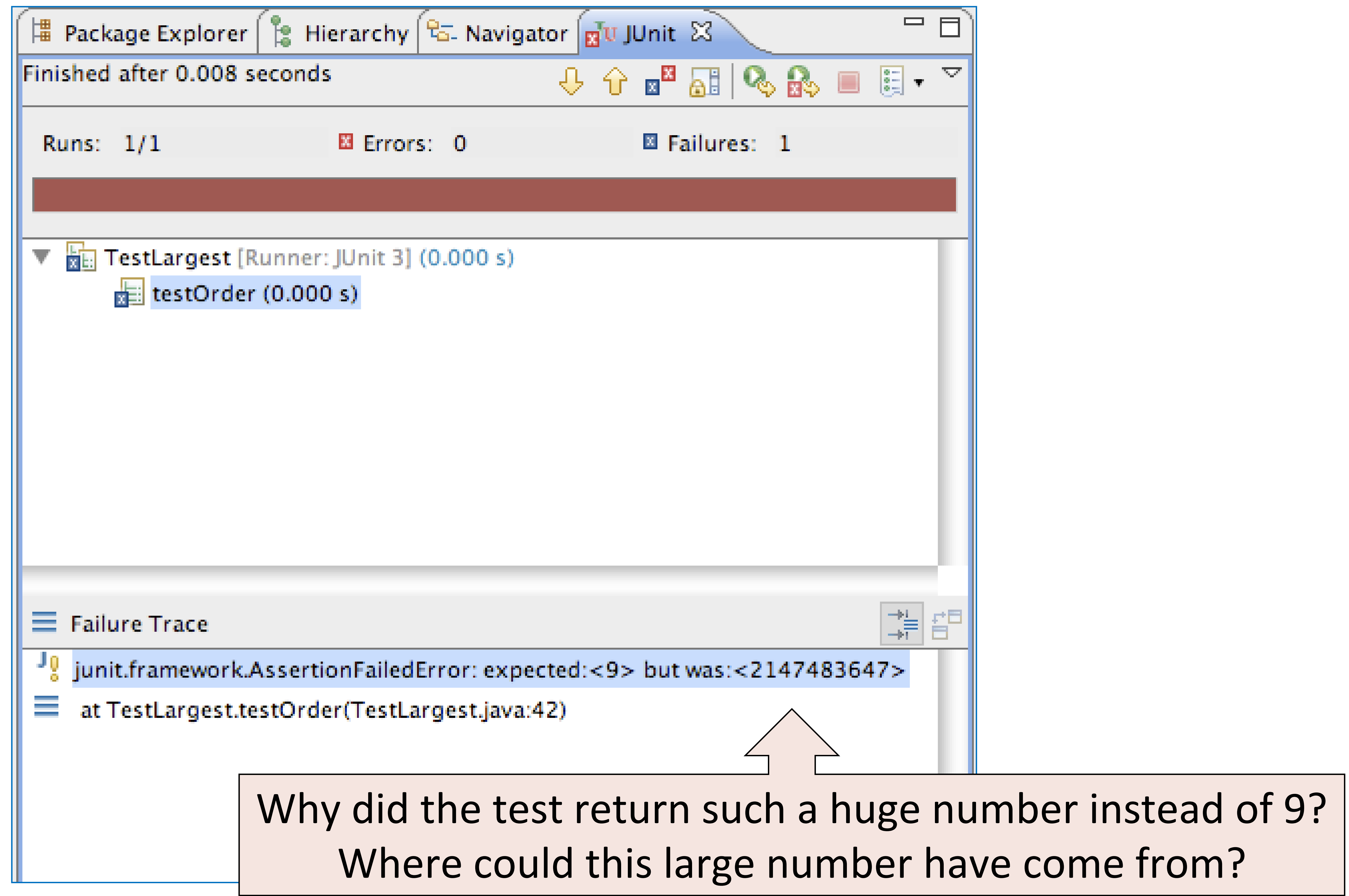
- This is a TestCase called TestLargest.
- It uses the following test data:
  [8, 9, 7] → 9
- It has one Unit Test (testOrder) - to verify the behaviour of the largest method.

```
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
  public TestLargest (String name)
  {
    super(name);
  }

  public void testOrder ()
  {
    int[] arr = new int[3];
    arr[0] = 8;
    arr[1] = 9;
    arr[2] = 7;
    assertEquals(9, Largest.largest(arr));
  }
}
```

Running the TestCase



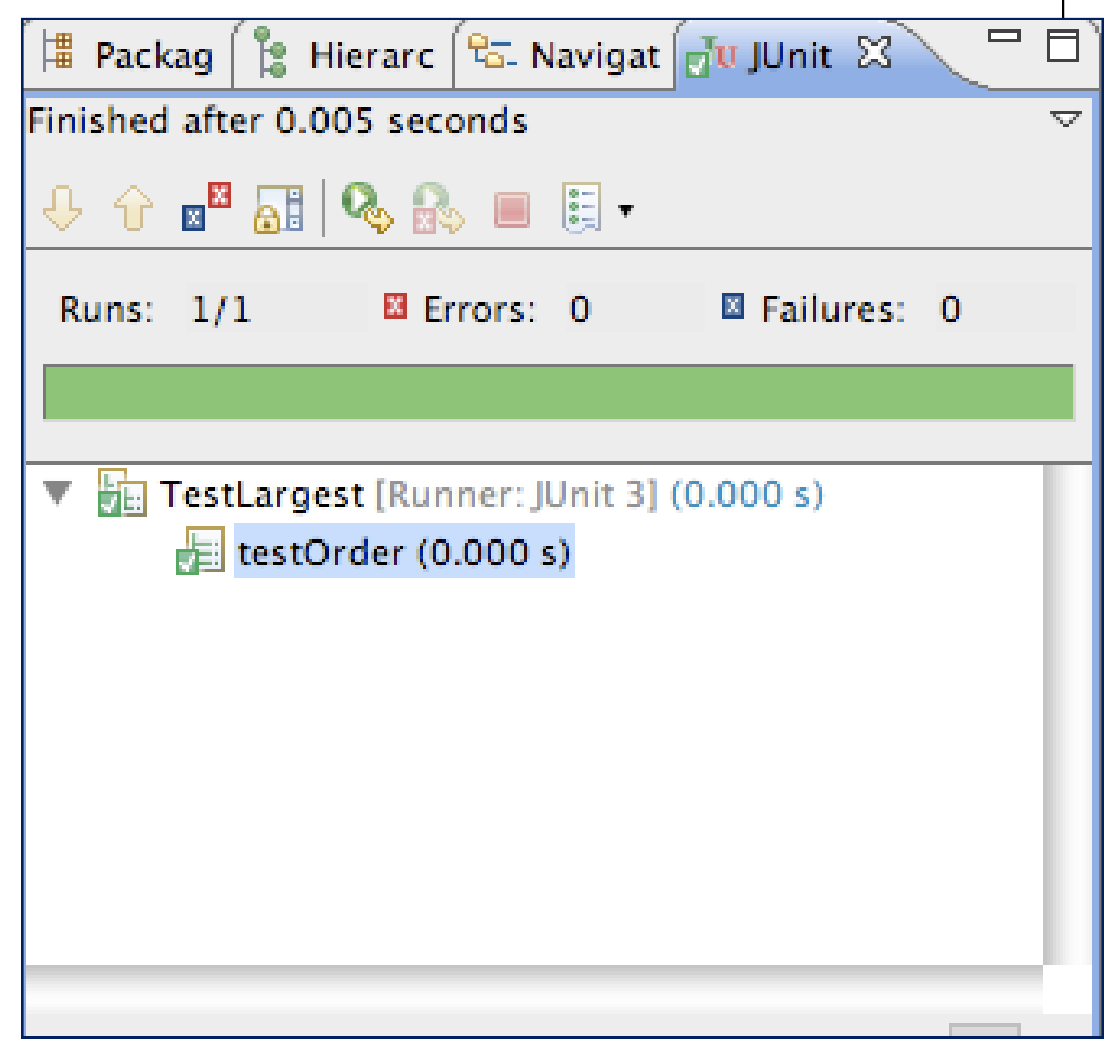
Running the TestCase



Bug

- First line should initialize max to zero, not MAX_VALUE.

```
public static int largest (int[] list)
{
  //int index, max = Integer.MAX_VALUE;
  int index, max = 0;

  for (index = 0; index < list.length - 1; index++)
  {
   if (list[index] > max)
   {
     max = list[index];
   }
  }
  return max;
}
```

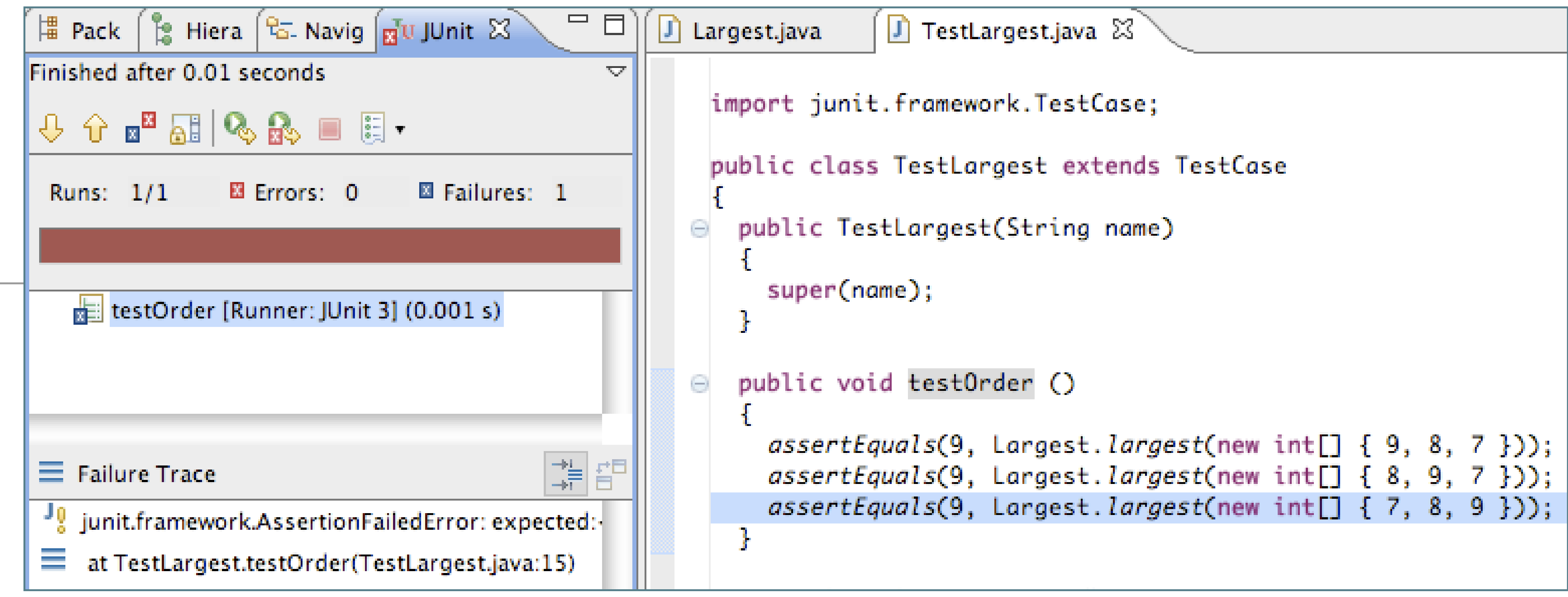


Further Tests

- What happens when the largest number appears in different places in the list - first or last, and somewhere in the middle?
  - Bugs most often show up at the “edges”.
  - In this case, edges occur when the largest number is at the start or end of the array that we pass in.
- Aggregate into a single unit test:

```
public void testOrder ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));
  assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));
  assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));
}
```

Failure



Failure + Fix

Pack | Hiera | Navig | JUnit

Finished after 0.01 seconds

Runs: 1/1 Errors: 0 Failures: 1

testOrder [Runner: JUnit 3] (0.001 s)

Failure Trace

junit.framework.AssertionFailedError: expected:

at TestLargest.testOrder(TestLargest.java:15)

Largest.java | TestLargest.java

```java
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
  public TestLargest(String name)
  {
    super(name);
  }

  public void testOrder ()
  {
    assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));
    assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));
    assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));
  }
```

```java
public static int largest (int[] list)
{
  int index, max = 0;
  //for (index = 0; index < list.length - 1; index++)
  for (index = 0; index < list.length; index++)
  {
   if (list[index] > max)
   {
    max = list[index];
   }
  }
  return max;
}
```

Further Boundary Conditions

```
public void testDups ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 7, 9, 8 }));
}

public void testOne ()
{
  assertEquals(1, Largest.largest(new int[] { 1 }));
}
```

- Now exercising multiple tests

Pack Hiera Navig JUnit

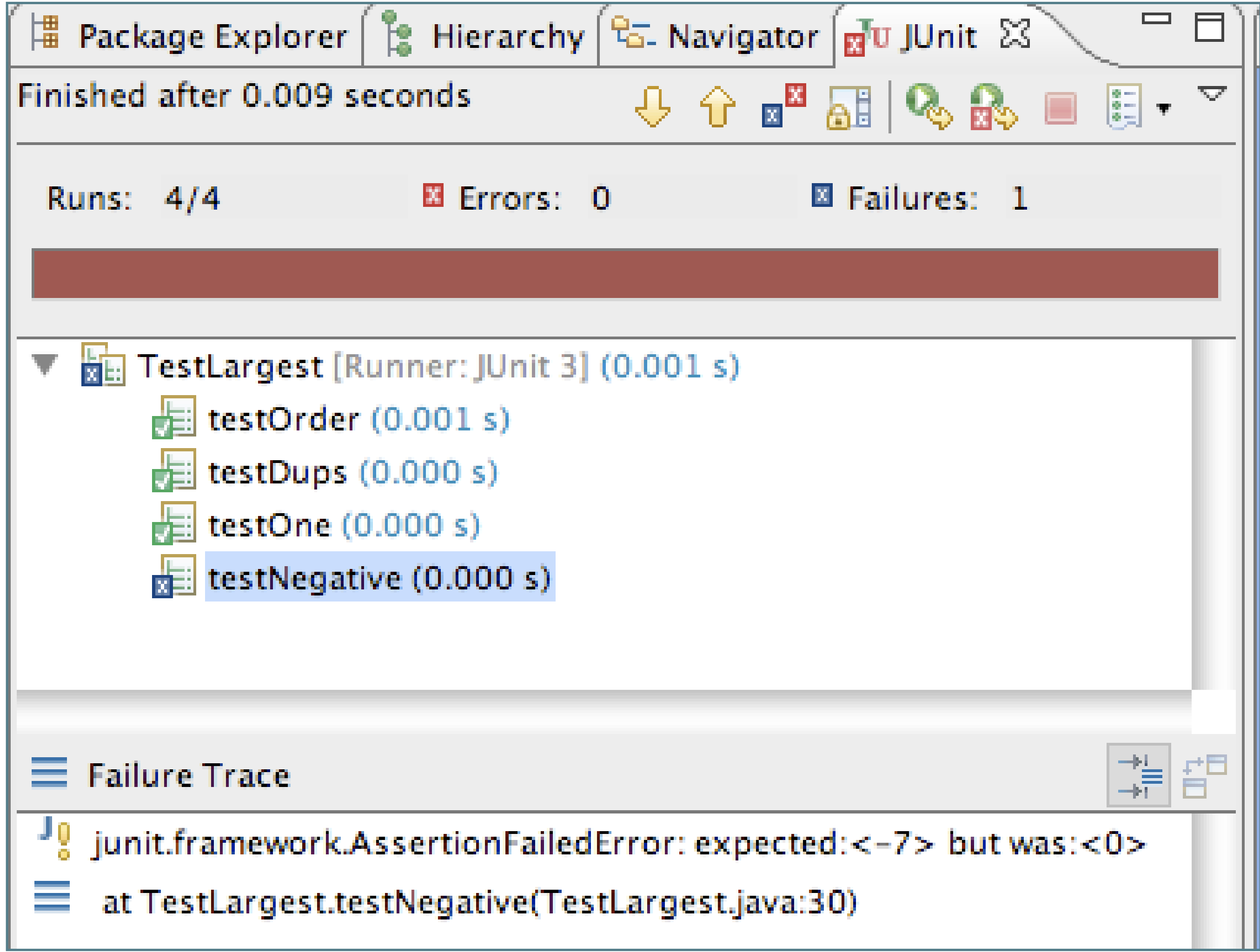
Finished after 0.007 seconds

Runs: 3/3 Errors: 0 Failures: 0

TestLargest [Runner: JUnit 3] (0.000 s)
testOrder (0.000 s)
testDups (0.000 s)
testOne (0.000 s)

Failure on testNegative

```
public void testNegative ()
{
    int[] negList = new int[] { -9, -8, -7 };
    assertEquals(-7, Largest.largest(negList));
}
```



fix testNegative

- Choosing 0 to initialize max was a bad idea;
- Should have been MIN VALUE, so as to be less than all negative numbers as well.

```
public static int largest (int[] list)
{
    //int index, max = 0;
    int index, max = Integer.MIN_VALUE;

    for (index = 0; index < list.length; index++)
    {
        if (list[index] > max)
        {
            max = list[index];
        }
    }
    return max;
}
```

Expected Errors?

- If the array is empty, this is considered an error, and an exception should be thrown.

```
public static int largest (int[] list)
{
  int index, max = Integer.MIN_VALUE;

  if (list.length == 0)
  {
    throw new RuntimeException("Empty list");
  }
  for (index = 0; index < list.length; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

```
public void testEmpty ()
{
  try
  {
    Largest.largest(new int[] {});
    fail("Should have thrown an exception");
  }
  catch (RuntimeException e)
  {
    assertTrue(true);
  }
}
```

Some more TDD theory…

TDD – Common Pitfalls (individual programmer)

- Forgetting to run tests frequently
- Writing too many tests at once
- Writing tests that are too large or coarse-grained
- Writing overly trivial tests, for instance omitting assertions
- Writing tests for trivial code, for instance accessors

https://www.agilealliance.org/glossary/tdd/

TDD – Common Pitfalls (teams)

- Partial adoption - only a few developers on the team use TDD.
- Poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time.
- Abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover.

https://www.agilealliance.org/glossary/tdd/

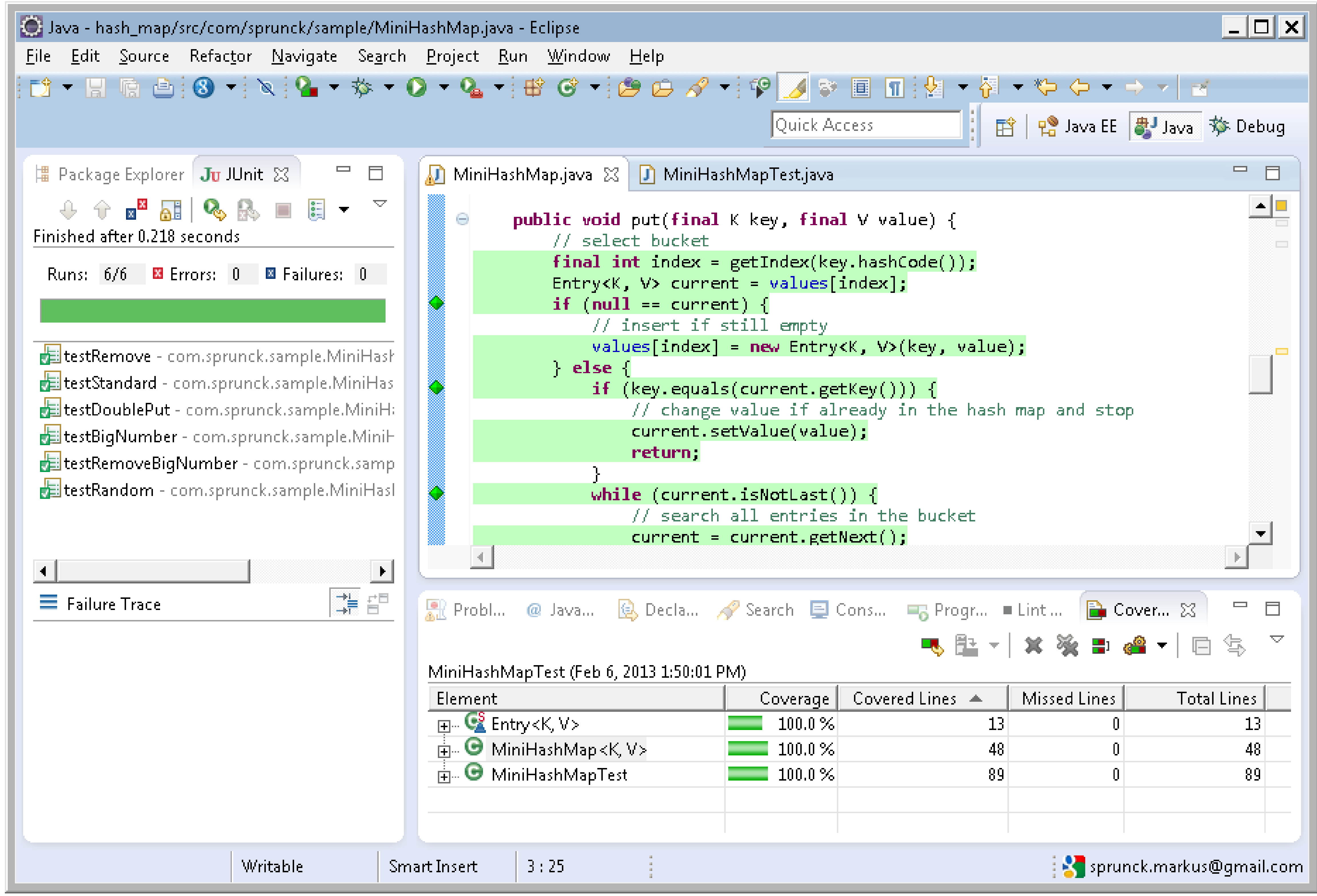
TDD – Signs of Use

- "code coverage" is a common approach to evidencing the use of TDD; while high coverage does not guarantee appropriate use of TDD, coverage below 80% is likely to indicate deficiencies in a team's mastery of TDD.

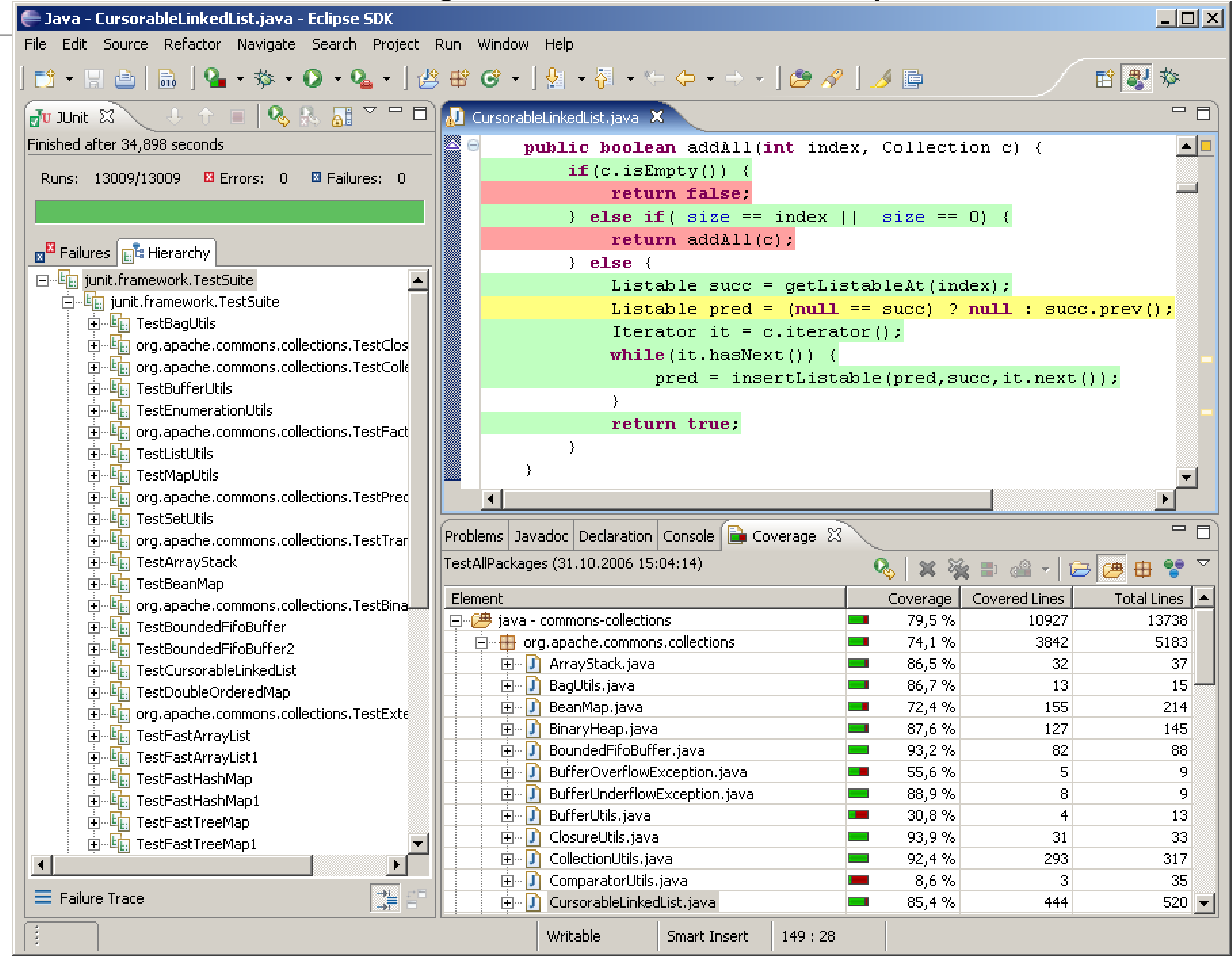
- version control logs should show that test code is checked in each time product code is checked in, in roughly comparable amounts.

https://www.agilealliance.org/glossary/tdd/

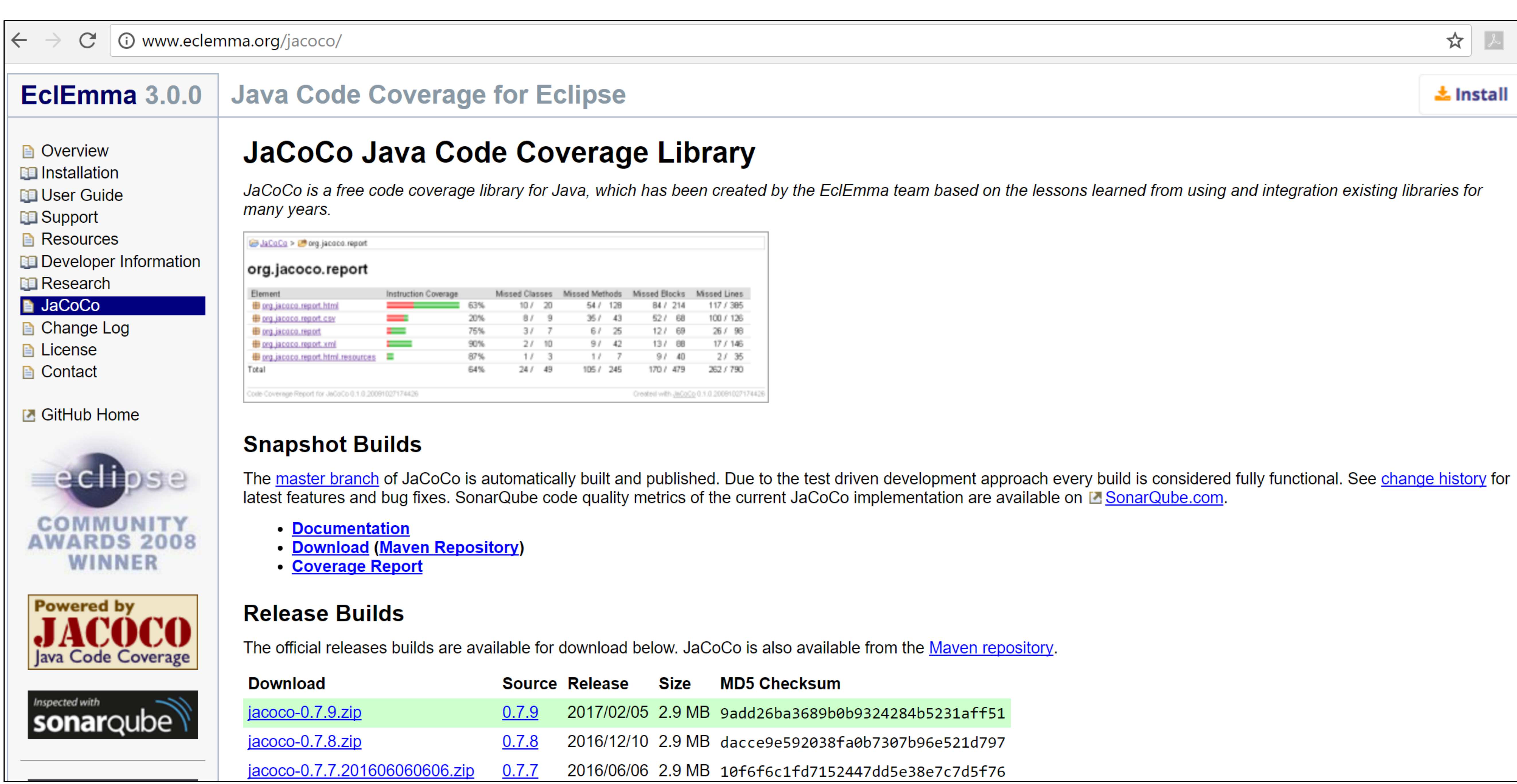
TDD – Code Coverage – 100% Example

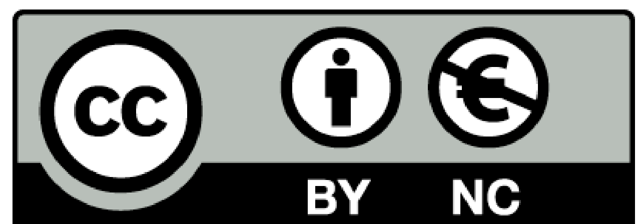


TDD – Code Coverage – 85.4% Example



TDD – Code Coverage Tool





Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see http://creativecommons.org/licenses/by-nc/3.0/