JUnit

A tool for test-driven development

History

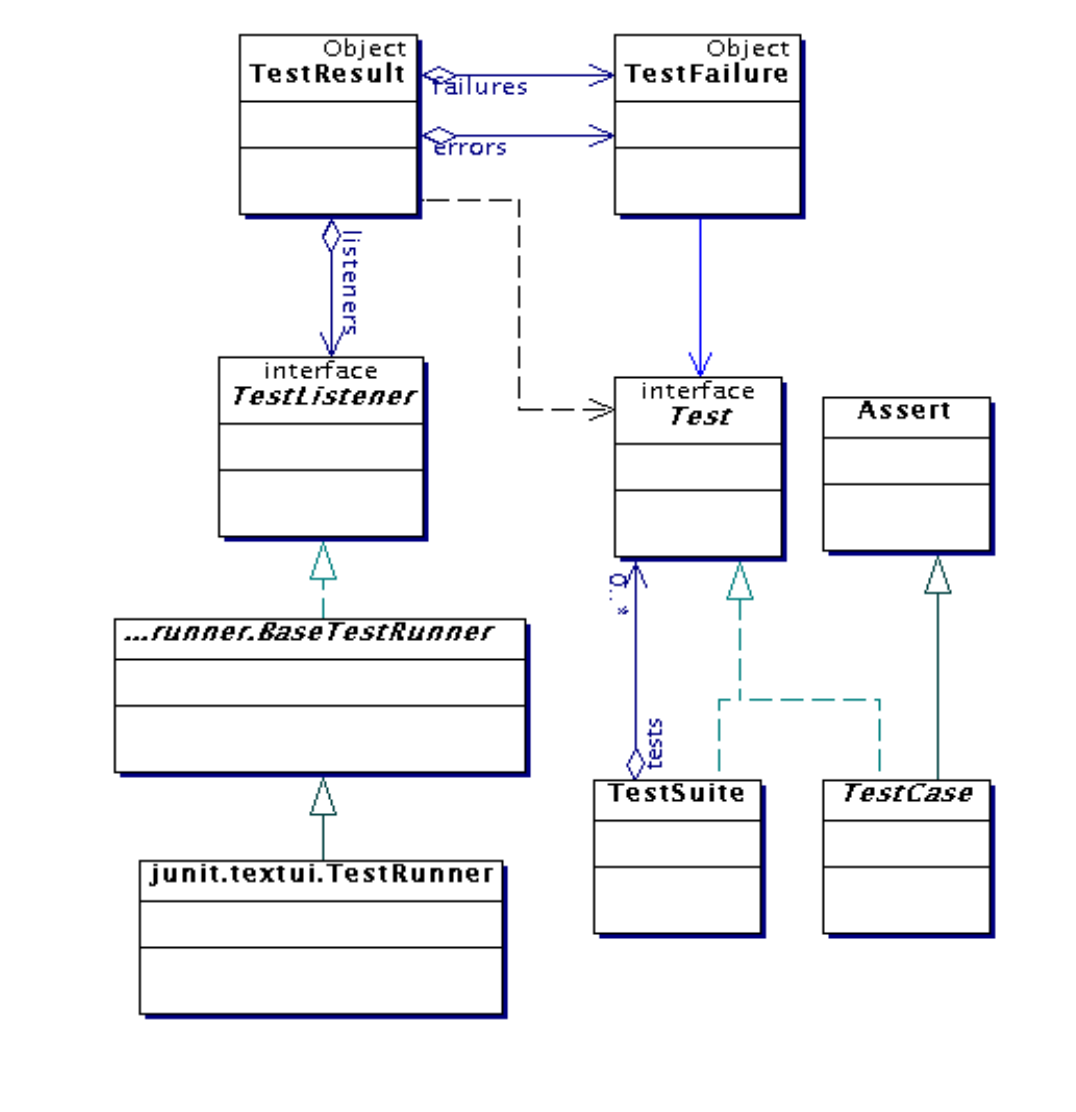
- Kent Beck developed the first xUnit automated test tool for Smalltalk in mid-90’s
- Beck and Gamma (of design patterns Gang of Four) developed JUnit on a flight from Zurich to Washington, D.C.
- Martin Fowler: “Never in the field of software development was so much owed by so many to so few lines of code.”
- Junit has become the standard tool for Test-Driven Development in Java (see Junit.org)
- Junit test generators now part of many Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava)
- Xunit tools have since been developed for many other languages (Perl, C++, Python, Visual Basic, C#, …)

Why create a test suite?

- Obviously you have to test your code—right?
  - You can do *ad hoc* testing (running whatever tests occur to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of a test suite
  - It’s a lot of extra programming
    - True, but use of a good test framework can help quite a bit
  - You don’t have time to do all that extra work
    - *False!* Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
  - Reduces total number of bugs in delivered code
  - Makes code much more maintainable and refactorable

Architectural overview

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- **TestRunner** runs tests and reports **TestResults**
- You test your class by extending abstract class ***TestCase***
- To write test cases, you need to know and understand the **Assert** class



Writing a TestCase

- To start using JUnit, create a subclass of *TestCase*, to which you add test methods
- Here’s a skeletal test class:

```
import junit.framework.TestCase;
public class TestBowl extends TestCase {
} //Test my class Bowl
```

- Name of class is important – should be of the form ***Test***MyClass or MyClass***Test***
- This naming convention lets TestRunner automatically find your test classes

Writing methods in TestCase

- Pattern follows ***programming by contract*** paradigm:
  - Set up **preconditions**
  - Exercise functionality being tested
  - Check **postconditions**
- Example:

```
public void testEmptyBowl() {
    Bowl emptyBowl = new Bowl();
    assertEquals(“Size of an empty bowl should be zero.”,
                 0, emptyBowl.size());
    assertTrue(“An empty bowl should report empty.”,
    emptyBowl.isEmpty());
}
```

- Things to notice:
  - Specific method signature – public void ***test***Whatever()
    - Allows them to be found and collected automatically by JUnit
  - Coding follows pattern
  - Notice the assert-type calls…

Assert methods

- Each assert method has parameters like these: *message*, *expected-value*, *actual-value*
- Assert methods dealing with floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
  - messages helps documents the tests
  - messages provide additional information when reading failure logs

Assert methods

- assertTrue(String *message*, Boolean *test*)
- assertFalse(String *message*, Boolean *test*)
- assertNull(String *message*, Object *object*)
- assertNotNull(String *message*, Object *object*)
- assertEquals(String *message*, Object *expected*, Object *actual*) (uses equals method)
- assertSame(String *message*, Object *expected*, Object *actual*) (uses == operator)
- assertNotSame(String *message*, Object *expected*, Object *actual*)

More stuff in test classes

- Suppose you want to test a class Counter
- public class CounterTest
  extends junit.framework.TestCase {
  - This is the unit test for the Counter class
- public CounterTest() { } //Default constructor
- protected void setUp()
  - Test *fixture* creates and initializes instance variables, etc.
- protected void tearDown()
  - Releases any system resources used by the test fixture
- public void testIncrement(), public void testDecrement()
  - These methods contain tests for the Counter methods increment(), decrement(), etc.
  - Note capitalization convention

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
    public CounterTest() { }    // default constructor

    protected void setUp() {    // creates a (simple) test fixture
        counter1 = new Counter();
    }

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

    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

TestSuites

- TestSuites collect a selection of tests to run them as a unit
- Collections automatically use TestSuites, however to specify the order in which tests are run, write your own:

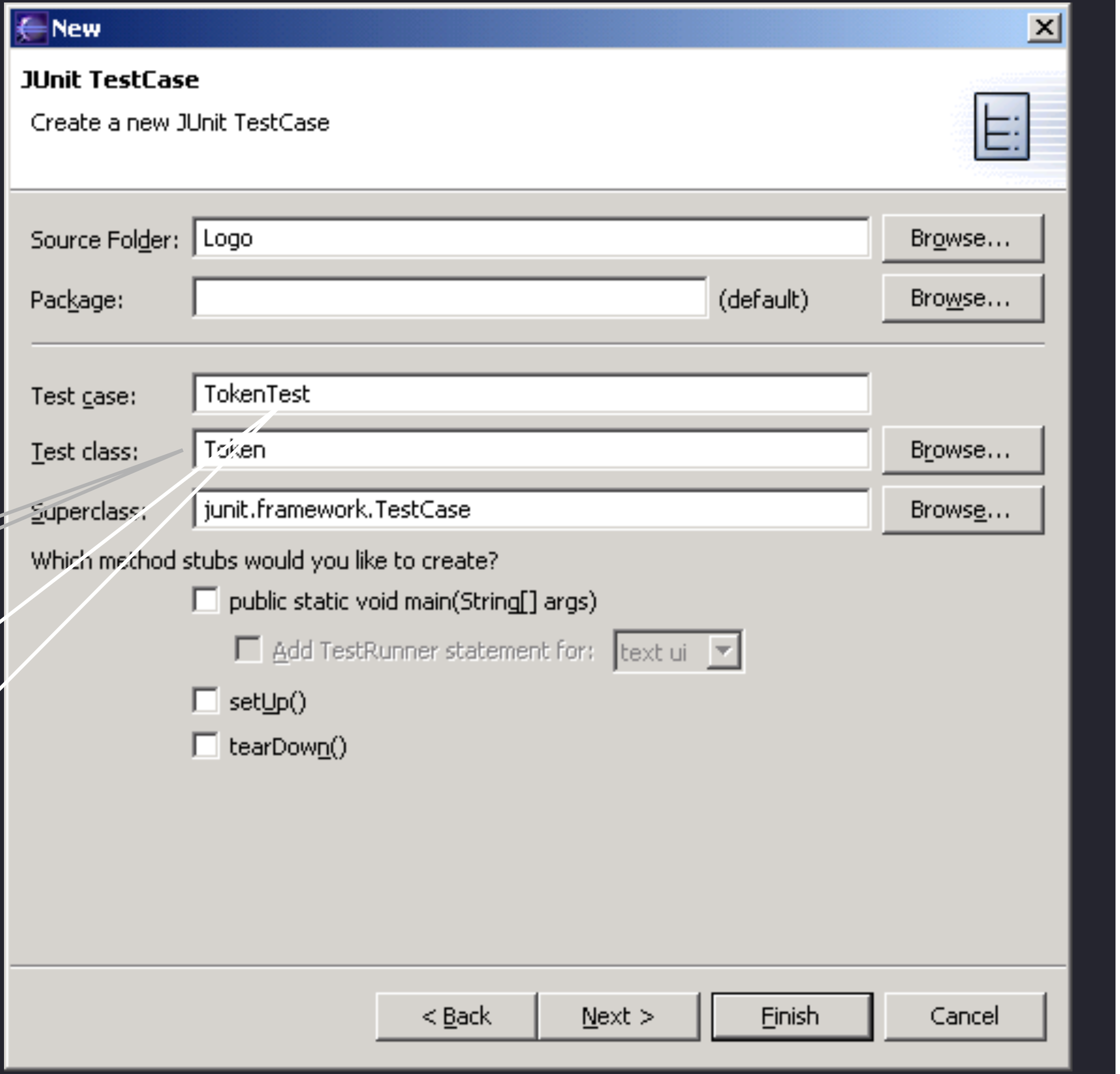
```
public static Test suite() {
        suite.addTest(new TestBowl(“testBowl”));
        suite.addTest(new TestBowl(“testAdding”));
        return suite;
}
```

- Should seldom have to write your own TestSuites as each method in your TestCase should be independent of all others
- Can create TestSuites that test a whole package:

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestBowl.class);
        suite.addTestSuite(TestFruit.class);
        return suite;
}
```

JUnit in Eclipse

- To create a test class, select File→ New→ Other... → Java, JUnit, TestCase and enter the name of the *class* you will test



New

JUnit TestCase

Create a new JUnit TestCase

Source Folder: Logo

Package: (default)

Test case: TokenTest

Test class: Token

Superclass: junit.framework.TestCase

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Add TestRunner statement for: text ui

□ setUp()

□ tearDown()

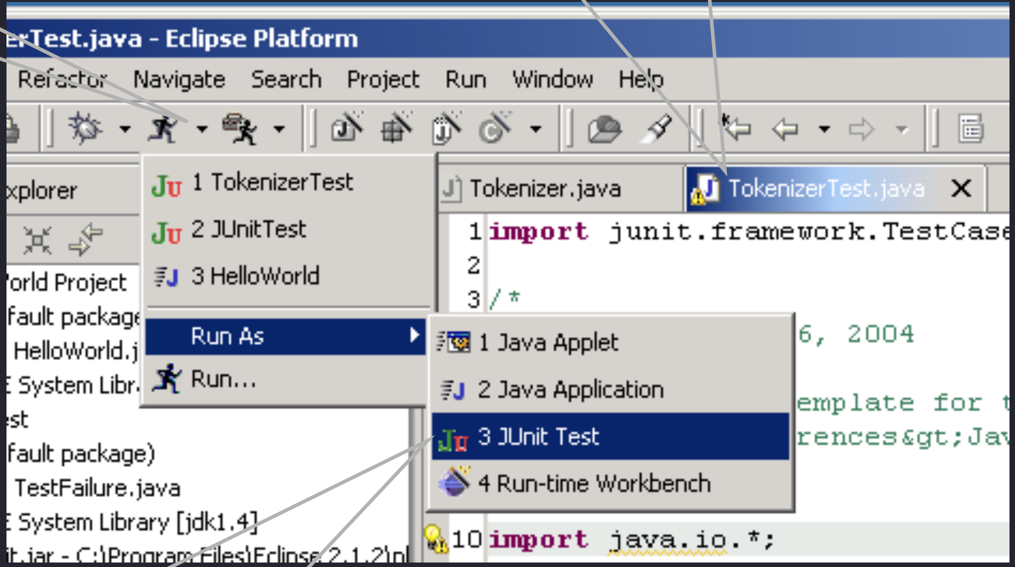
Fill this in

This will be filled in *automatically*

Running JUnit

Second, use this pulldown menu

First, select a *Test* class

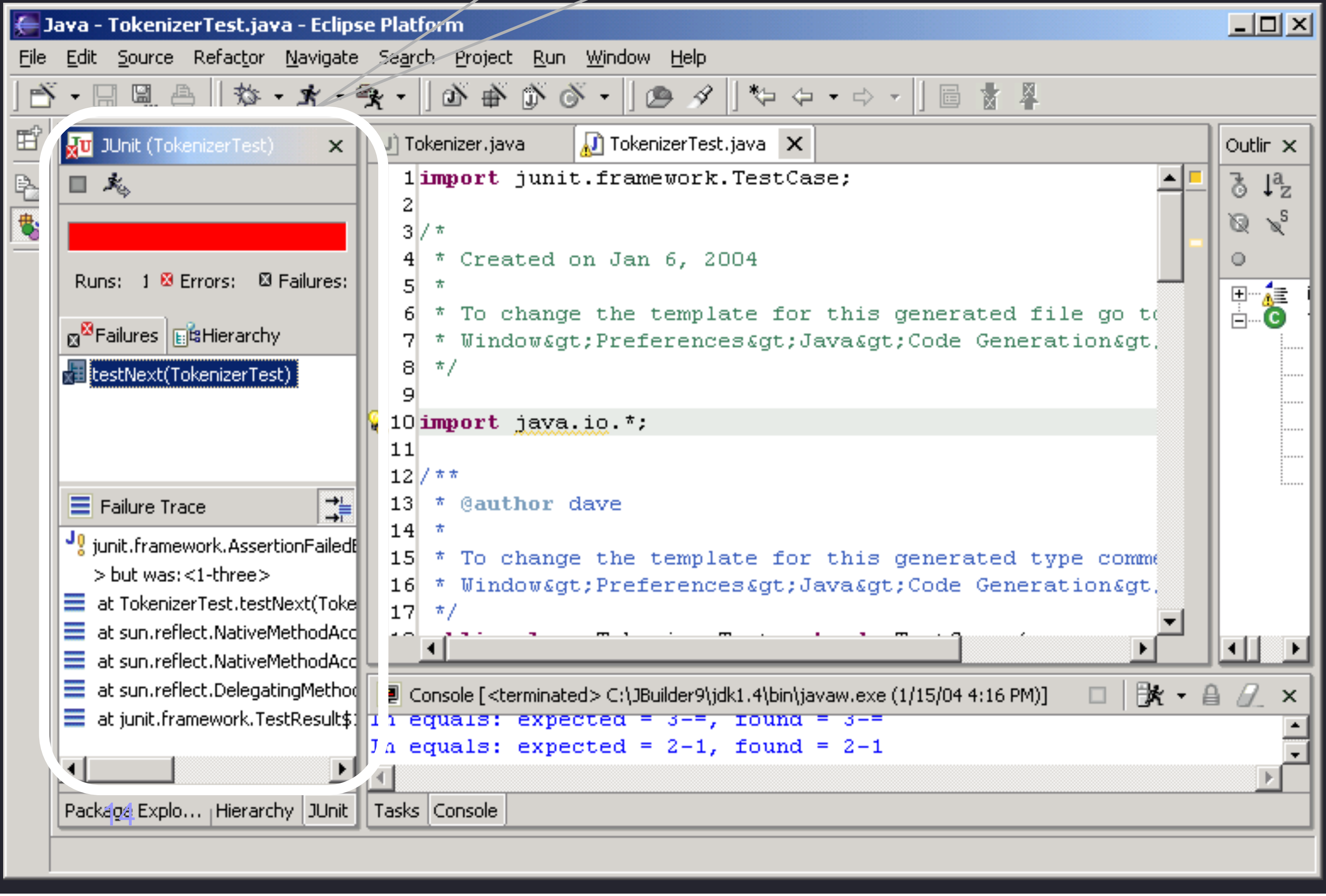


```
import junit.framework.TestCase
/*
6, 2004
emplate for t
rences&gt;Jav
import java.io.*;
```

Third, Run As → JUnit Test

Results

Your results are here



Runs: 1 Errors: Failures:

testNext(TokenizerTest)

```
import junit.framework.TestCase;

/*
 * Created on Jan 6, 2004
 *
 * To change the template for this generated file go t
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;
 */

import java.io.*;

/**
 * @author dave
 *
 * To change the template for this generated type comme
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;
 */
```

```
junit.framework.AssertionFailedE
> but was:<1-three>
at TokenizerTest.testNext(Toke
at sun.reflect.NativeMethodAcc
at sun.reflect.NativeMethodAcc
at sun.reflect.DelegatingMethoc
at junit.framework.TestResult$
```

Console [<terminated> C:\JBuilder9\jdk1.4\bin\javaw.exe (1/15/04 4:16 PM)]

```
In equals: expected = 3-=, found = 3-=
In equals: expected = 2-1, found = 2-1
```

Unit testing for other languages

- Unit testing tools differentiate between:
  - Errors (unanticipated problems caught by exceptions)
  - Failures (anticipated problems checked with assertions)
- Basic unit of testing:
  - *CPPUNIT_ASSERT(Bool)* examines an expression
- CPPUnit has variety of test classes (e.g. *TestFixture*)
  - Inherit from them and overload methods

More Information

- http://www.junit.org
  - Download of JUnit
  - Lots of information on using JUnit
- http://sourceforge.net/projects/cppunit
  - C++ port of Junit
- http://www.thecoadletter.com
  - Information on Test-Driven Development