CS 535 Object-Oriented Programming
Fall Semester, 2000
Doc 3 Testing
Contents

Testing 2
  Johnson's Law 2
  Why Unit Testing 3
  When to Write Unit Tests 4
JUnit 5
  Using JUnit 6
    Test Fixtures 12
    Suites – Multiple Test Classes 13
  What to Test 17
  The Complete JUnit Example 18

References

JUnit Cookbook Local copy at: http://www.eli.sdsu.edu/java-SDSU/junit/cookbook/cookbook.htm

JUnit Test Infected: Programmers Love Writing Tests Local copy at: http://www.eli.sdsu.edu/java-SDSU/junit/testinfected/testing.htm

JUnit on-line documentation Local copy at: http://www.eli.sdsu.edu/java-SDSU/docs/

Originals of the above can be found at: http://www.junit.org/

Copyright ©, All rights reserved.
2000 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Testing

Johnson's Law

If it is not tested it does not work

Types of tests

- Unit Tests

    Tests individual code segments

- Functional Tests

    Test functionality of an application

Why Unit Testing

If it is not tested it does not work

The more time between coding and testing

- More effort is needed to write tests
- More effort is needed to find bugs
- Fewer bugs are found
- Time is wasted working with buggy code
- Development time increases
- Quality decreases

Without unit tests

- Code integration is a nightmare
- Changing code is a nightmare

When to Write Unit Tests

First write the tests

Then write the code to be tested

Writing tests first saves time

- Makes you clear of the interface & functionality of the code
- Removes temptation to skip tests

JUnit

Framework for unit testing Java code

Available at: http://www.junit.org/

Already installed in JDK 1.2 on rohan and moria

Ports of JUnit are available in

C++	Delphi	Eiffel
Forte 4GL	Objective-C	Perl
PowerBuilder	Python	Ruby
Smalltalk	Visual Basic	

See http://www.xprogramming.com/software.htm to download ports of JUnit

Using JUnit
Example

Goal: Implement a Stack containing integers.

Tests:

Subclass junit.framework.TestCase

Methods starting with 'test" are run by TestRunner

First tests for the constructors:

```
package example;

import junit.framework.TestCase;

public class  StackTest extends TestCase {

  //required constructor
  public StackTest(String name) {
    super(name);
  }

  public void testDefaultConstructor() {
    Stack test = new Stack();
    assert( test.isEmpty() );
  }

  public void testSizeConstructor() {
    Stack test = new Stack(5);
    assert( test.isEmpty() );
  }
}
```

First part of the Stack

```
package example;

public class Stack  {
   int[] elements;
   int topElement = -1;

   public Stack() {
    this(10);
   }

   public Stack(int size) {
      elements = new int[size];
   }

   public boolean isEmpty() {
      return topElement == -1;
   }
}
```

Running JUnit

JUnit has three interfaces

- Text (junit.textui.*)
- AWT (junit.ui.*)
- Swing (junit.swingui.*)

  Shows list of previously run test classes

JUnit has two class loaders

- Normal java class loader (TestRunner)
- junit.util.TestCaseClassLoader (LoadingTestRunner)

  Reloads classes without having to restart program

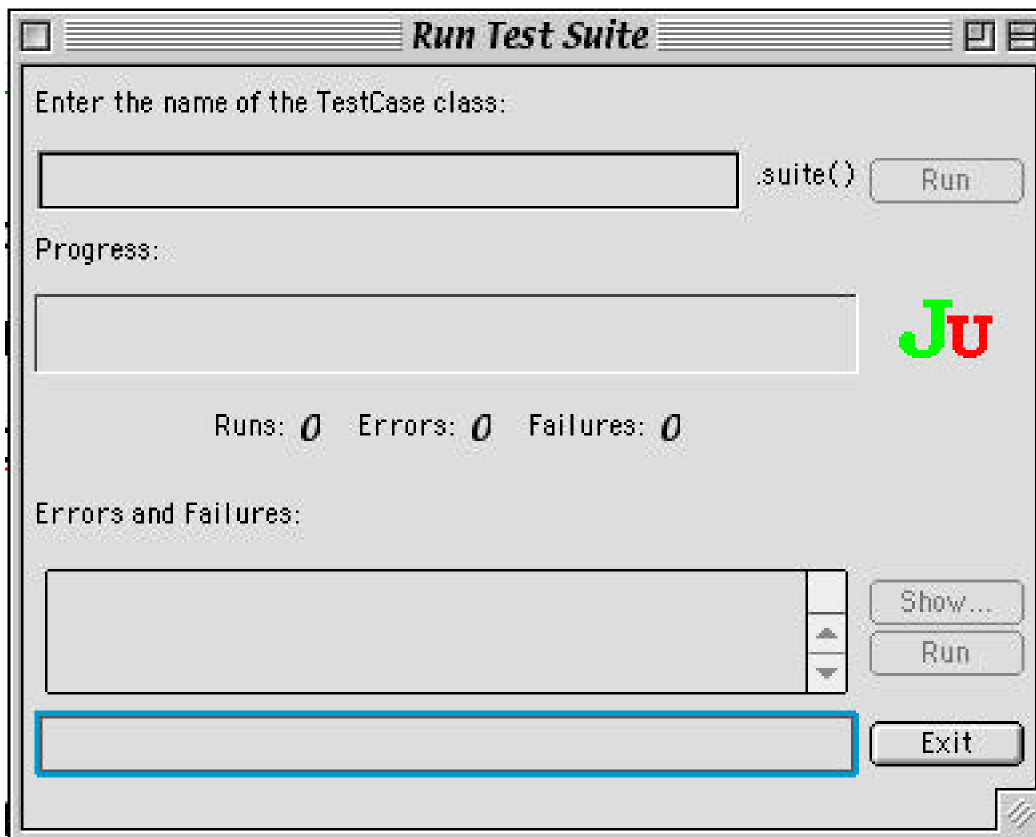
Starting TestRunner

Make sure your classpath includes the code to tested

On Rohan use:

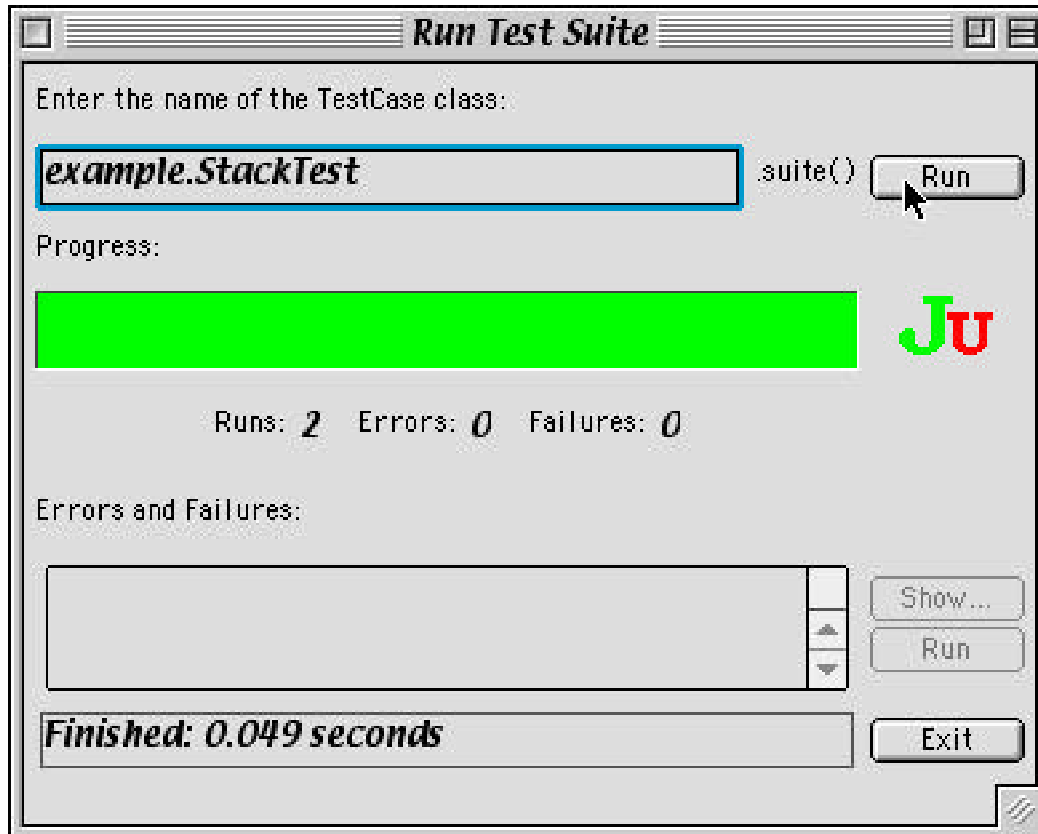
java junit.ui.LoadingTestRunner

You get a window like:



Enter the full name of the test class

Click on the Run button



If there are errors/failures select one and click on Show

You will see a stack trace of the error

With LoadingTestRunner you can recompile the Stack & StackTest classes without exiting LoadingTestRunner

Testing the Tests

If can be useful to modify the code to break the tests

```
package example;

public class Stack  {
   int[] elements;
   int topElement = -1;

   etc.

   public boolean isEmpty() {
      return topElement == 1;
   }
}
```

One company had an automatic build and test cycle that ran at night. The daily build was created and all the tests were run at night. The test results were available first thing in the morning. One night the build process crashed, so the daily build was not made. Hence there was no code to test. Still 70% of the tests passed. If they had tested their tests, they would have discovered immediately that their tests were broken.

Test Fixtures

Before each test setUp() is run

After each test tearDown() is run

```
package example;

import junit.framework.TestCase;

public class  StackTest extends TestCase {
   Stack test;

   public StackTest(String name) {
      super(name);
   }

   public void setUp() {
      test = new Stack(5);
      for (int k = 1; k <=5;k++)
         test.push( k);
   }

   public void testPushPop() {
      for (int k = 5; k >= 1; k--)
         assert( "Popping element " + k,  test.pop() == k);
   }
}
```

Suites – Multiple Test Classes

Multiple test classes can be run at the same time

Running AllTests in TestRunner runs the test in

StackTest
QueueTest

```
package example;
import junit.framework.TestSuite;

public class AllTests
{
  static public TestSuite suite()
    {
      TestSuite suite= new TestSuite();
    try
      {
      suite.addTest(new TestSuite(StackTest.class));
      suite.addTest(new TestSuite(QueueTest.class));
      }
    catch (Exception e)
      {
      }
      return suite;
    }
}
```

Using Main

We can use main to run the test via textui.TestRunner

The command:

```
java example.AllTests
```

will run all the tests in StackTest & QueueTest

```
package example;

import junit.framework.TestSuite;
import junit.textui.TestRunner;

public class AllTests
   {
   static public void main(String[] args)
      {
      TestRunner.main(args);
      }

   static public TestSuite suite()
      {
      same as last page
      }
   }
```

Just For Completeness

The QueueTest and Queue classes exist but don't do much

```
package example;

import junit.framework.TestCase;

public class QueueTest extends TestCase
   {
   public QueueTest( String name)
      {
      super(name);
      }
   public void testConstructor()
      {
      Queue test = new Queue();
      assert( test.isEmpty());
      }
   }

package example;

import java.util.Vector;

public class  Queue
   {
   Vector elements = new Vector();
   public boolean isEmpty()
      {
      return elements.isEmpty();
      }
   }
```

Why not just use print statements?

Using print statements does not scale

```
package example;

public class StackTest {

   public static void main( String[] args ) {
      Stack test = new Stack();
      System.out.println( "Expect: true Result: " +
         test.isEmpty());

      test = new Stack(5);
      System.out.println( "Expect: true Result: " +
         test.isEmpty();
   }
}
```

What to Test

Everything that could possibly break

Test values

- Inside valid range
- Outside valid range
- On the boundary between valid/invalid

GUIs are very hard to test

- Keep GUI layer very thin
- Unit test program behind the GUI, not the GUI

The Complete JUnit Example
Stack

```
package example;

public class Stack  {
  int[] elements;
  int topElement = -1;

  public Stack()   { this(10); }

  public Stack(int size) { elements = new int[size]; }

  public boolean isEmpty() {
    return topElement == -1;
  }

  public boolean isFull() {
    return topElement == elements.length-1;
    }

  public void push( int element) {
    topElement++;
    elements[topElement] = element;
    }

  public int pop() {
    return elements[topElement--];
    }
  }
```

StackTest

```
package example;

import junit.framework.TestCase;

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

  public void testDefaultConstructor() {
    Stack test = new Stack();
    assert( test.isEmpty() );
  }

  public void testSizeConstructor() {
    Stack test = new Stack(5);
    assert( test.isEmpty() );
  }

  public void testUnderflow() {
    Stack test = new Stack(512345);
    test.push( 1);
    test.pop();
    try {
      test.pop();
      fail( "Pop on empty stack passed");
    }
    catch (Exception overflow) {
    }
  }
```

```
  public void testPushAndFull() {
    Stack test = new Stack(5);
    assert("Is empty", test.isEmpty() );
    assert("Empty stack claims full", test.isFull() == false);
    for (int k = 1; k <=5;k++)
      test.push( k);

    assert("Should not be empty", test.isEmpty() == false);
    assert( "Is full", test.isFull() );
    try {
      test.push( 6);
      fail("Full stack accepted element");
    }
    catch (Exception overflow) {
    }
  }

  public void testPushPop() {
    Stack test = new Stack(5);
    for (int k = 1; k <=5;k++)
      test.push( k);

    for (int k = 5; k >= 1; k--)
      assert( "Pop fail on element " + k,  test.pop() == k);
  }
}
```