eXtreme Testing

Effective unit testing with JUnit

written by Eric M. Burke

burke_e@ociweb.com

last revised 12 Oct 2000

What is eXtreme Programming (XP)

- A lightweight software development methodology
  - best for small to medium teams (less than 20 people)
  - coding and testing are key activities
    - constant code reviews through **pair programming**
    - **refactoring** is done constantly
    - focus on **simplicity**
    - continuous integration **testing**
    - short iterations
- This is not a presentation on XP

XP Testing

- Programmers write unit tests
  - if the interface for a method is unclear
  - if an implementation will be complicated
  - to test unusual inputs and boundary conditions
  - before refactoring
  - **all unit tests must run at 100%**
- "Customers" write functional tests
  - may require a dedicated tester to help
  - failed functional tests can help determine priorities

Unit Test Process

- Write unit tests <u>before</u> code
  - helps you to think about the interface
  - makes it easy to determine when the code is finished
    - **just write enough code to solve the current problem**
- When a bug occurs
  - first write a unit test to expose the bug
  - then fix the bug
  - then run the test again

XP discourages heavy emphasis on "framework" development early on. Focus on the simplest solution that will solve today's problems. Reusable code will emerge as the result of refactoring.

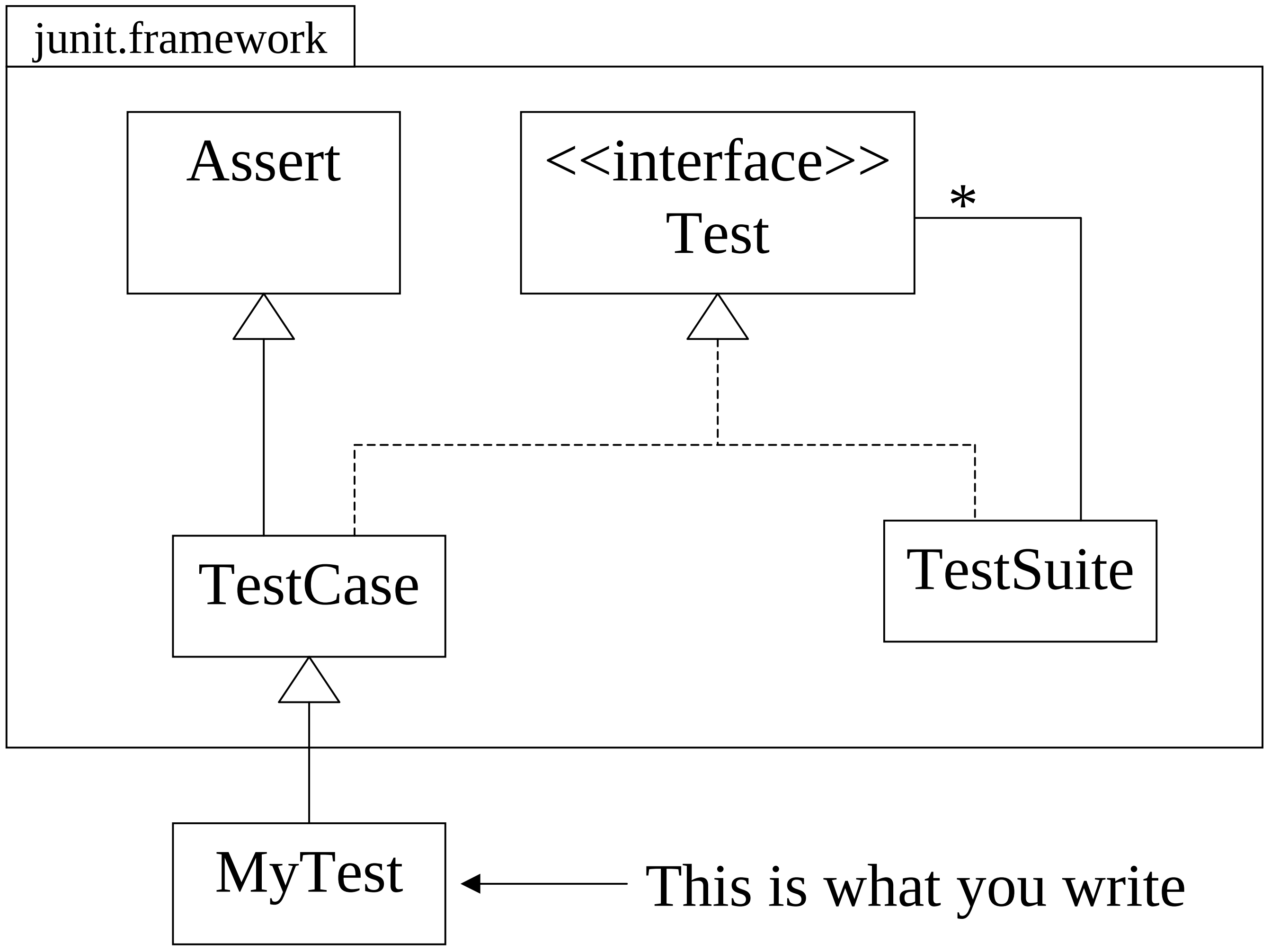
Good Test Design

- Don't test trivial things
  - like simple getter/setter methods
  - will ultimately discourage programmers from writing good tests
- Test anything that may go wrong
- Tests become part of the system documentation
- Tests must be automated
  - pass/fail
  - don't make someone interpret the results

JUnit Overview

- A free Java tool for writing unit tests
  - www.junit.org
  - visit www.xprogramming.com for other languages
    - C++, Delphi, Eiffel, Forte 4GL, Objective-C, Perl, PowerBuilder, Python, Smalltalk, Visual Basic, others...
- Supports batch mode and GUI mode testing
- Test cases can be grouped into test suites
- All tests are pass/fail type tests
  - stack traces and error messages indicate where failures occur

JUnit Design



Writing a Unit Test

- Create a subclass of TestCase

```
import junit.framework.*;
public class HelloWorldTest extends TestCase {
    public HelloWorldTest(String name) {
        super(name);
    }
```

- Write individual test methods
  - numerous assert(...) methods are available

```
    public void testSample1() {
        String s1 = "someString";
        String s2 = "someString";
        assertEquals(s1, s2);
    }
```

Writing a Unit Test (Cont'd)

- Rules for test methods
  - must be public, and should not take arguments
  - should begin with the name "test"
    - so reflection can determine which methods to call
  - may optionally throw exceptions
    - will be reported as **errors** by JUnit
      - asserts are reported as **failures**
  - it is OK to use assert several times in the same method
    - when an assert condition is not met, the current method is finished

Writing a Unit Test (Cont'd)

- Include a method called suite()
  - again, reflection locates this method

```
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new HelloWorldTest("testSample1"));
        suite.addTest(new HelloWorldTest("testSample2"));
        suite.addTest(new HelloWorldTest("testSample3"));
        return suite;
    }
```

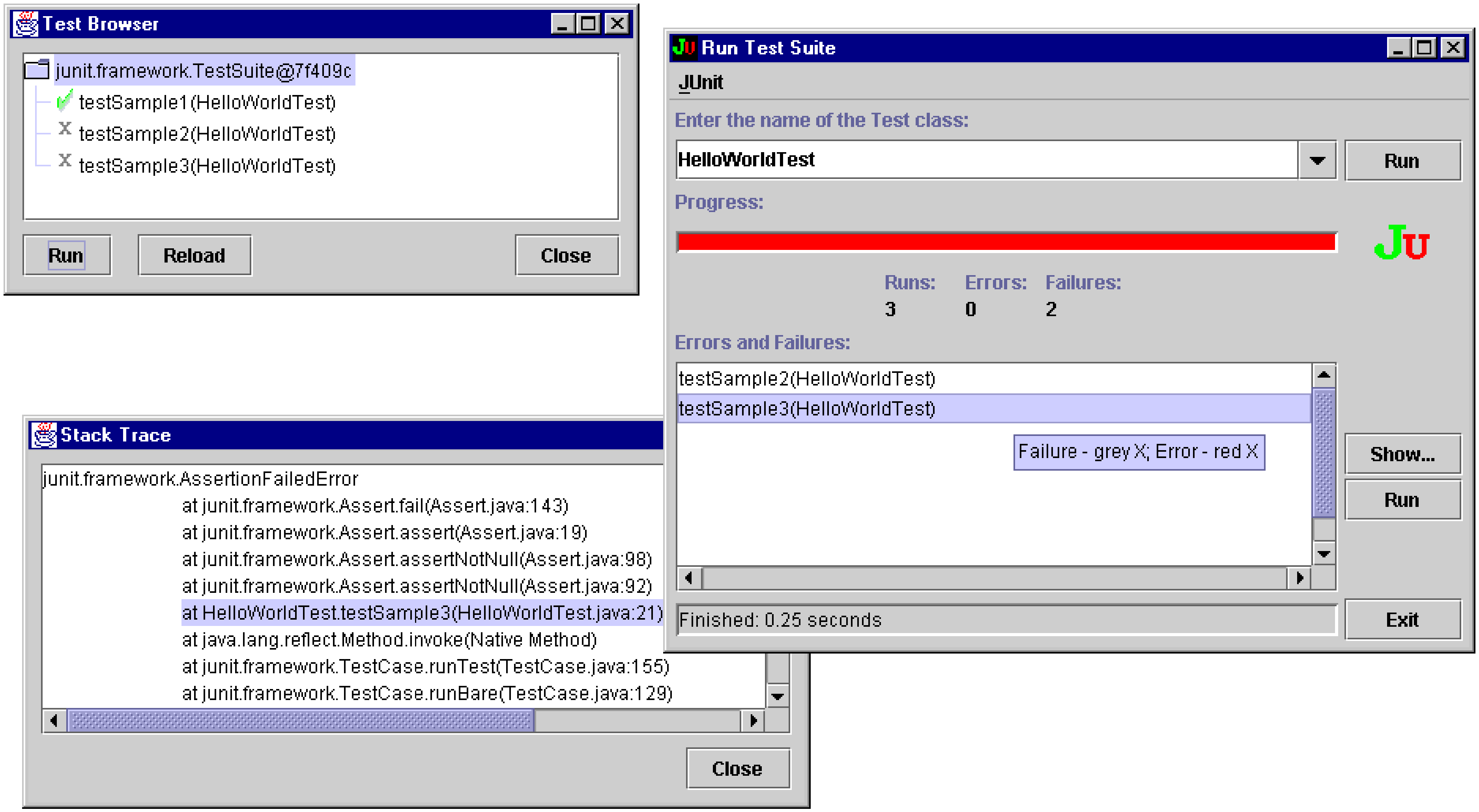
- A main(...) method is convenient for batch tests

```
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Running the Tests

- For batch mode testing
  - `java HelloWorldTest`
    - uses the main() method shown on the previous page
  - `java junit.textui.TestRunner HelloWorldTest`
- For Swing GUI testing
  - `java junit.swingui.TestRunner SomeTest`
  - `java junit.swingui.LoadingTestRunner SomeTest`
    - reloads classes when they change; won't work with JARs
- Older AWT GUI is in the junit.ui package
    - no dropdown list of previous tests
    - no test browser capability

JUnit Demo



```
junit.framework.TestSuite@7f409c
  ✓ testSample1(HelloWorldTest)
  X testSample2(HelloWorldTest)
  X testSample3(HelloWorldTest)
```

```
HelloWorldTest

Runs:  Errors:  Failures:
3      0        2

testSample2(HelloWorldTest)
testSample3(HelloWorldTest)

Failure - grey X; Error - red X

Finished: 0.25 seconds
```

```
junit.framework.AssertionFailedError
        at junit.framework.Assert.fail(Assert.java:143)
        at junit.framework.Assert.assert(Assert.java:19)
        at junit.framework.Assert.assertNotNull(Assert.java:98)
        at junit.framework.Assert.assertNotNull(Assert.java:92)
        at HelloWorldTest.testSample3(HelloWorldTest.java:21)
        at java.lang.reflect.Method.invoke(Native Method)
        at junit.framework.TestCase.runTest(TestCase.java:155)
        at junit.framework.TestCase.runBare(TestCase.java:129)
```

Other JUnit Features

- Fixtures
  - a convenient way to set up data for multiple tests
  - the fixture can fail just like a test
    - by calling assert, fail, or by throwing an exception
  - see example code on next page
- Using Java reflection to construct a TestSuite
  - you can create a TestSuite in a single line of code
  - automatically adds all test* methods

```
Test mySuite = new TestSuite(MyTester.class);
```

Example Fixture

- The setUp and tearDown methods are called before an after every test method invocation

```
public class FixtureDemo extends TestCase {
    private FileReader demoReader;

    // IOException is only required for this example
    protected void setUp() throws IOException {
        demoReader = new FileReader("demo.dat");
    }

    protected void tearDown() throws IOException {
        demoReader.close();
    }
```

junit.extensions.TestSetup

- Allows you to `setUp` once before a batch of tests and `tearDown` once at the end...

```
public static Test suite() {
    TestSuite suite = new TestSuite(TestSetupDemo.class);
    return new TestSetup(suite) {
        public void setUp() {
            System.out.println("oneTimeSetup");
        }
        public void tearDown() {
            System.out.println("oneTimeTeardown");
        }
    };
}
```

Setting up a RepeatingTest

- Use junit.extensions.RepeatedTest

```
public static void main(String[] args) {
    TestSuite mySuite = new TestSuite();
    mySuite.addTest(new MyTest("testSample1"));
    mySuite.addTest(new MyTest("testSample3"));

    // first parameter is a Test, so it doesn't
    // have to be a TestSuite
    Test repeater = new RepeatedTest(mySuite, 1000);
    junit.textui.TestRunner.run(repeater);
}
```

- output:

```
................................
Time: 0.741
OK (2000 tests)
```

Threading

- Use junit.extensions.ActiveTest
  - could not find examples; documentation is sparse
  - won't work with the command line TestRunner
    - as soon as the test is finished, TestRunner executes System.exit(0)
      - the threaded test appears to finish immediately, as soon as the Thread object is started
  - Swing GUI doesn't repaint properly if individual threads run very fast
    - when I inserted sleep() statements, the GUI repainted

Threading Example

```
public static Test suite() {
    Test repeater1 = new RepeatedTest(
        new ThreadDemo("testSample1"), 100);
    Test repeater2 = new RepeatedTest(
        new ThreadDemo("testSample2"), 50);

    // run the repeaters in parallel
    Test thread1 = new ActiveTest(repeater1);
    Test thread2 = new ActiveTest(repeater2);

    TestSuite suite = new TestSuite();
    suite.addTest(thread1);
    suite.addTest(thread2);

    return suite;
}
```

Assert Summary

- assert(boolean condition) // ensures that condition is true
  - assert(String message, boolean condition)
    - all remaining assert(...) methods have an optional message parameter
- assertEquals(double expected, double actual)
  - overloaded for long, Object
- assertNotNull(Object obj)
- assertNull(Object obj)
- assertSame(Object expected, Object actual) ← == comparison
- fail()
- fail(String message)

Testing Tips

- JUnit is really only designed for unit tests
  - you may need other tools for integration testing or load testing
- Some things, like Servlets, are very hard to test
  - you could simulate an HTTP request to the server
    - your test program is pretending to be a browser
  - or create an implementation of the HttpServletRequest interface, and invoke your Servlet without a server
    - the discussion forum mentioned on the next page has links to example code which does this
  - you should separate business logic from your Servlet anyway
    - most tests will be written against the domain objects, rather than the Servlet itself

Learning More

- eXtreme Programming eXplained
  - Kent Beck, Addison Wesley
  - a short chapter on unit testing; JUnit is not covered
- Refactoring
  - Martin Fowler, Addison Wesley
  - the chapter on unit testing is an excellent JUnit intro
- http://www.c2.com/cgi/wiki?JavaUnit
  - discussion forum
- http://www.extremeprogramming.org