JUnit

Tom Reichlmayr
Software Engineering Department
Rochester Institute of Technology
Agenda

- Background – Unit Testing
- JUnit Features - SimpleTest
- Looking Under the Hood
- More JUnit Examples
Testing, 1 – 2 – 3 – 4, Testing...
What Does a Unit Test Test?

- The term “unit” predates the O-O era.
- Unit – “natural” abstraction unit of an O-O system: class or its instantiated form, object.
- Unit Tests – verify a small chunk of code, typically a path through a method or function.
- Not application level functionality.
How Do We Unit Test?

- Print Statements (diffs against benchmarks)
- Debuggers – examine variables, observe execution paths.
- Typically done by unit developer.
- Best benefit if running of tests is *automated*.
- Tests best run in isolation from one another.
- Tests built incrementally as product code is developed.
The Typical Test Cycle

- Develop a suite of test cases
- Create some test fixtures to support the running of each test case.
- Run the test – capture test results.
- Clean-up fixtures, if necessary.
- Report and analyze the test results.
Why is Unit Testing Good?

- Identifies defects early in the development cycle.
- Many small bugs ultimately leads to chaotic system behavior.
- Testing affects the design of your code.
- Successful tests breed confidence.
- Testing forces us to read our own code – spend more time reading than writing.
- Automated tests support maintainability and extendibility.
Why Don’t We Unit Test?

- “Coding unit tests takes too much time”
- “I’m too busy fixing bugs to write tests”
- “Testing is boring – it stifles my creativity”
- “My code is virtually flawless…”
- “Testing is better done by the testing department”
- “We’ll go back and write unit tests after we get the code working”
What is JUnit?

JUnit is an open source Java testing framework used to write and run repeatable tests.

It is an instance of the xUnit architecture for unit testing frameworks.

JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners
The JUnit Test Template

- Create a test class that extends TestCase
- Create a testXXX() method for each individual test to be run.
- Create a test fixture – resources needed to support the running of the test.
- Write the test, collect interesting test behavior
- Tear down the fixture (if needed)
- Run the tests with a text or Swing interface.
import java.util.*;
import junit.framework.*;

public class SimpleTest extends TestCase{

    public void testEmptyCollection() {
        Collection testCollection = new ArrayList();
        assertTrue( testCollection.isEmpty());
    }

    public static void main(String args[]){
        junit.textui.TestRunner.run(SimpleTest.class);
    }
}
Key JUnit Concepts

- **assert** -
  - `assertEquals( expected, actual )` – also `NotEquals`
  - `assertNull( actual result )` – also `NotNull`
  - `assertTrue( actual result )` – also `False`

- **failures** –
  - Exceptions raised by asserts (expected)

- **errors** –
  - Java runtime exceptions (not expected)
JUnit supports test hierarchies

- Test Suite-A
  - Test Case1
  - Test Case2
- Test Suite-B
  - Test Case3
- Test Suite-C

(and so on …)
JUnit Under the Hood

Originally written by Kent Beck and Erich Gamma. – design patterns.

An offspring of a similar framework for Smalltalk (SUnit)

A common xUnit architecture has evolved and has been implemented in a variety of languages.
JUnit Design Objectives

- A simple framework that encourages developers to write unit tests.
- Minimalist framework – essential features, easier to learn, more likely to be used, flexible
- Test Cases & Test Results are objects
- Patterns – high “density” of patterns around key abstractions: mature framework
See the JUnit Cook’s Tour for a full pattern analysis:
http://junit.sourceforge.net/doc/cookstour/cookstour.htm
TestCase

- **TestCase.run()** applies **Template Method** pattern

```java
public void run()
{
    setup();
    runTest();
    tearDown();
}
```

“Define the skeleton of an algorithm in an operation, deferring some steps to subclasses”
TestResult

TestResult applies Collecting Parameter pattern

```java
class TestResult {
  public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
      runTest();
    } catch (AssertionFailedError e) {
      result.addFailure(this, e);
    }
    catch (Throwable e) {
      result.addError(this, e);
    }
    finally {
      tearDown();
    }
  }
}
```

“When you need to collect results over several methods, add a parameter to the method and pass an object that will collect results for you”

(Beck – Smalltalk Best Practice Patterns)
TestSuite applies **Composite** pattern

“Composite objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly”
More…

- TDD / TFD ???
  - Test Driven Design
  - Test First Design
  - JUnit provides support for these agile techniques, but JUnit is lifecycle agnostic
- Extensions for J2EE applications
- What about GUI’s? – JUnit limited
Resources

- JUnit: [www.junit.org](http://www.junit.org)
- cppUnit: [http://cppunit.sourceforge.net/doc/1.8.0/index.html](http://cppunit.sourceforge.net/doc/1.8.0/index.html)
- Unit Testing in Java – How Tests Drive the Code, Johannes Link
- Test-Driven Development by Example, Kent Beck
- Pragmatic Unit Testing in Java with JUnit, Andy Hunt & Dave Thomas