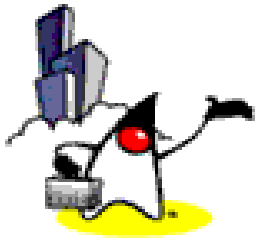
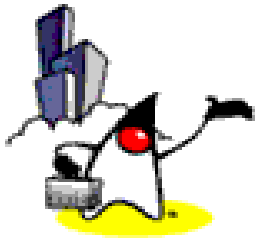
JUnit Testing Framework



Topics

- What is and Why JUnit framework?
- JUnit mechanics
- Fixtures
- Test suites
- Test runners
- JUnit classes
- Best practice guidelines
- Design patterns for testing
- Extensions

Test-Driven Development (TDD)



Acknowledgment

- This presentation is a modified (and enhanced) version of Orlando Java User's Group presentation on the same subject by Matt Weber

Why test?

- Automated tests prove features
- Tests retain their value over time and allows others to prove the software still works (as tested).
- Confidence, quality, sleep
- Get to code sooner by writing tests.

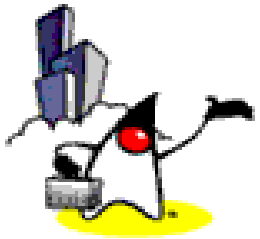
Test-Driven Development (TDD)

- Is a software development technique that involves repeatedly first writing a test case and then implementing only the code necessary to pass the test
- Gives rapid feedback

Unit Testing

- Is a procedure used to validate that individual units of source code are working properly
- A unit is the smallest testable part of an application
  - In procedural programming a unit may be an individual program, function, procedure, web page, menu etc, while in object-oriented programming, the smallest unit is always a Class; which may be a base/super class, abstract class or derived/child class

What is and Why JUnit Testing Framework?



What is JUnit?

- JUnit is a regression testing framework
- Used by developers to implement unit tests in Java (de-facto standard)
- Integrated with Ant
  - Testing is performed as part of nightly build process
- Goal: Accelerate programming and increase the quality of code.
- Part of XUnit family (HTTPUnit, Cactus), CppUnit

Why JUnit?

- Without JUnit, you will have to use println() to print out some result
  - No explicit concept of test passing or failure
  - No mechanism to collect results in a structured fashion
  - No replicability
- JUnit addresses all these issues

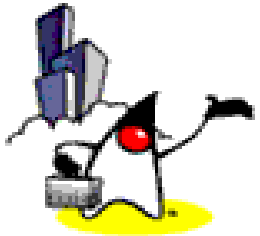
Key Goals of JUnit

- Easy to use to create tests
- Create tests that retain their value over time
- Leverage existing tests to write new ones (reusable)

What does JUnit provide?

- API for easily creating Java test cases
- Comprehensive assertion facilities
  - verify expected versus actual results
- Test runners for running tests
- Aggregation facility (test suites)
- Reporting

How to Write JUnit Testing Code?



How to write JUnit-based testing code (Minimum)

- Include JUnit.jar in the classpath
- Define a subclass of `TestCase` class
- Define one or more public `testXXX()` methods in the subclass
- Write assert methods inside the testXXX methods()
- Optionally define `main()` to run the TestCase in standalone mode

Test methods

- Test methods has name pattern testXXX()
  - XXX reflects the method of the target class
- Test methods must have no arguments
- Test methods are type of void

Example 1: Very Simple Test

```
import junit.framework.TestCase;

public class SimpleTest extends TestCase {

    public SimpleTest(String name) {
        super(name);
    }
    // Test code
    public void testSomething() {
        System.out.println("About to call assertTrue() method...");
        assertTrue(4 == (2 * 2));
    }
    // You don't have to have main() method, use Test runner
    public static void main(String[] args){
        junit.textui.TestRunner.run(SimpleTest.class);
    }
}
```

How to write JUnit-based testing code (More Sophisticated)

- Optionally override the setUp() & tearDown()methods
  - Create common test data
- Optionally define a static suite()factory method
  - Create a TestSuite containing all the tests.

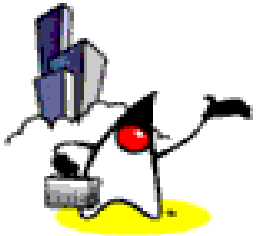
Example 2: More Sophisticated Example

```
// Define a subclass of TestCase
public class StringTest extends TestCase {

    // Create fixtures
    protected void setUp(){ /* run before */}
    protected void tearDown(){ /* after */ }

    // Add testing methods
    public void testSimpleAdd() {
      String s1 = new String("abcd");
      String s2 = new String("abcd");
      assertTrue("Strings not equal",
                 s1.equals(s2));
     }

    // Could run the test in batch mode
    public static void main(String[] args){
     junit.textui.TestRunner.run (suite ());
    }
```

Example 2: Simple Testcase (cont.)

```
// Create TestSuite object
public static Test suite (){
  suite = new TestSuite (”StringTest");
  String tests = System.getProperty("tests");
  if (tests == null){
    suite.addTest(new
       TestSuite(StringTest.class));
  }else{
    StringTokenizer tokens = new
       StringTokenizer(tests, ",");
    while (tokens.hasMoreTokens()){
      suite.addTest(new
      StringTest((String)tokens.nextToken()));
    }
  }
  return suite;
}
```

Assert Statements



Assert statements

- JUnit Assertions are methods starting with assert
- Determines the success or failure of a test
- An assert is simply a comparison between an expected value and an actual value
- Two variants
  - assertXXX(...)
  - assertXXX(String message, ...) - the message is displayed when the assertXXX() fails

Assert statements

- Asserts that a condition is true
  - assertTrue(boolean condition)
  - assertTrue(String message, boolean condition)
- Asserts that a condition is false
  - assertFalse(boolean condition)
  - assertFalse(String message, boolean condition)

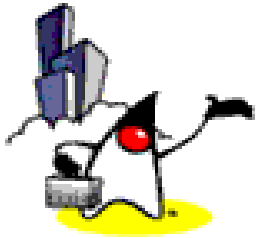
Assert statements

- Asserts expected.equals(actual) behavior
  - assertEquals(expected, actual)
  - assertEquals(String message, expected, actual)
- Asserts expected == actual behavior
  - assertSame(Object expected, Object actual)
  - assertSame(String message, Object expected, Object actual)

Assert statements

- Asserts object reference is null
  - assertNull(Object obj)
  - assertNull(String message, Object obj)
- Asserts object reference is not null
  - assertNotNull(Object obj)
  - assertNotNull(String message, Object obj)
- Forces a failure
  - fail()
  - fail(String message)

Fixtures



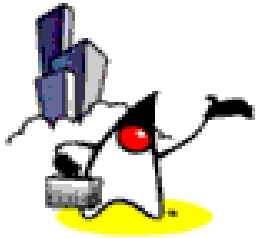
Fixtures

- setUp() and tearDown() used to initialize and release common test data
- setUp() is run before every test invocation & tearDown() is run after every test method

Example: setUp

```
public class MathTest extends TestCase {
    protected double fValue1, fValue2;

    protected void setUp() {
        fValue1= 2.0;
        fValue2= 3.0;
    }

    public void testAdd() {
        double result= fValue1 + fValue2;
        assertTrue(result == 5.0);
    }

    public void testMultiply() {
        double result= fValue1 * fValue2;
        assertTrue(result == 6.0);
    }
}
```

Test Suites



Test Suites

- Used to collect all the test cases
- Suites can contain testCases and testSuites
  - TestSuite(java.lang.Class theClass, <java.lang.String name>)
  - addTest(Test test) or addTestSuite(java.lang.Class testClass)
- Suites can have hierararchy

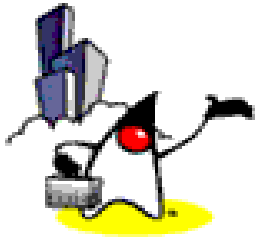
Example: Test Suites

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

public static Test suite (){
   suite = new TestSuite  ("AllTests");
   suite.addTest
      (new TestSuite (AllTests.class));
   suite.addTest (StringTest.suite());
}

public void testAllTests () throws Exception{
     assertTrue (suite != null);
}
```

Example: Test Suites

```
public static Test suite() {
        TestSuite suite = new TestSuite(IntTest.class);
        suite.addTest(new TestSuite(FloatTest.class));
        suite.addTest(new TestSuite(BoolTest.class));
        return suite;
 }
```

Test Runners



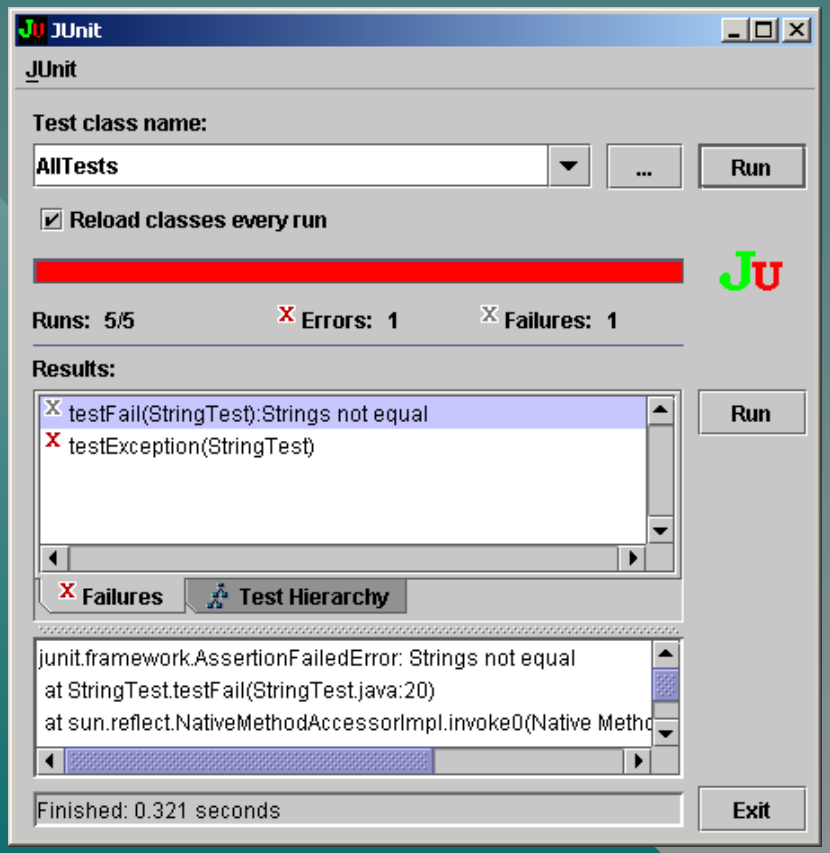
TestRunners

- Text
  - Lightweight, quick quiet
  - Run from command line

```
java StringTest
.......
Time: 0.05
Tests run: 7,  Failures: 0,  Errors: 0
```

TestRunners - Swing

- Run with java junit.swingui.TestRunner



Automating testing (Ant)

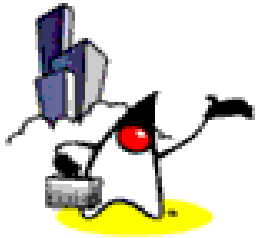
- JUnit Task

```
<target name="test" depends="compile-tests">
  <junit printsummary="yes" fork="yes">
       <classpath>
        <pathelement location="${build}" />
         <pathelement location="${build}/test" />
       </classpath>
       <formatter usefile="yes" type="plain" />
       <test name="AllTests" />
   </junit>
 </target>
```

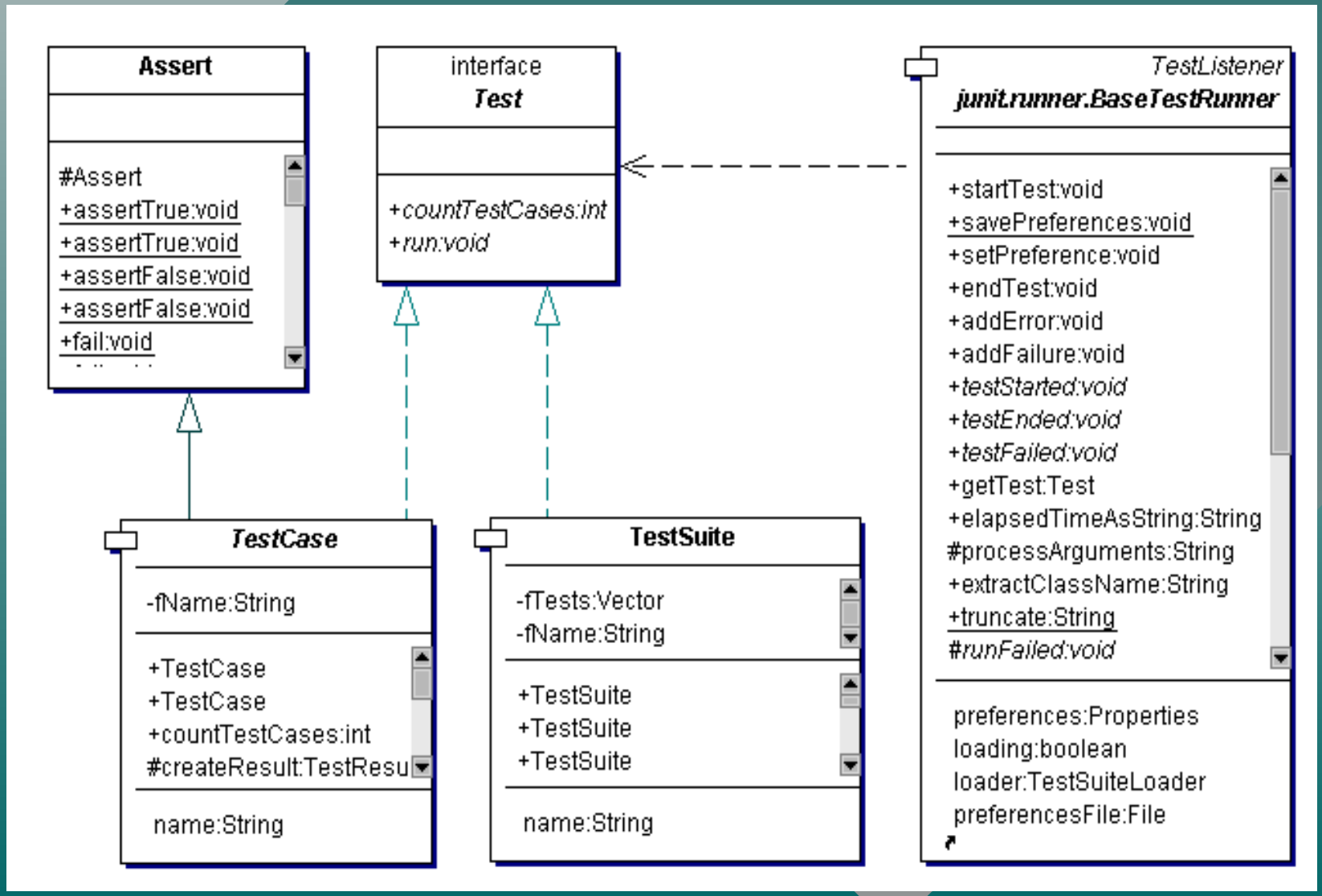
Ant Batch mode

```
<target name="batchtest" depends="compile-tests">
    <junit printsummary="yes" fork="yes" haltonfailure="no">
    <classpath>
        <pathelement location="${build.dir}" />
        <pathelement location="${build.dir}/test" />
     </classpath>
    <formatter type="plain" usefile="yes"/>

      <batchtest fork="yes" todir="">
       <fileset dir="${test.dir}">
            <include name="**/*Test.java" />
       </fileset>
     </batchtest>

   </junit>
 </target>
```

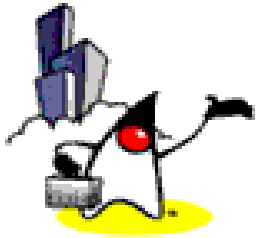
JUnit Classes



JUnit Class Diagram



Best Practice



What should I test?

- Tests things which could break
- Tests should succeed quietly.
  - Don’t print “Doing foo…done with foo!”
  - Negative tests, exceptions and errors
- What shouldn’t I test
  - Don’t test set/get methods
  - Don’t test the compiler

Test first and what to test

- Write your test first, or at least at the same time
- Test what can break
- Create new tests to show bugs then fix the bug
- Test driven development says write the test then make it pass by coding to it.

Testing for Exceptions

```
public void testExpectException()
{
    String s1 = null;
    String s2 = new String("abcd");

    try{
        s1.toString();
        fail("Should see null pointer");
    }
    catch(NullPointerException ex){
        assertTrue(true);
    }
}
```

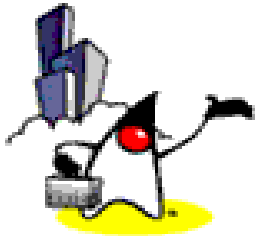
Test then Fix

- Bugs occasionally slip through (gasp!)
- Write a test first which demonstrates the error. Obviously, this test is needed.
- Now, fix the bug and watch the bar go green!
- Your tests assure the bug won't reappear.

Test then Refactor

- Once the code is written you want to improve it.
- Changes for performance, maintainability, readability.
- Tests help you make sure you don’t break it while improving it.
- Small change, test, small change, test...

Design Patterns for Testing



Designing for testing

- Separation of interface and implementation
  - Allows substitution of implementation to tests
- Factory pattern
  - Provides for abstraction of creation of implementations from the tests.
- Strategy pattern
  - Because FactoryFinder dynamically resolves desired factory, implementations are plugable

Design for testing - Factories

- `new` only used in Factory
- Allows writing tests which can be used across multiple implementations.
- Promotes frequent testing by writing tests which work against objects without requiring extensive setUp
  - "extra-container" testing.

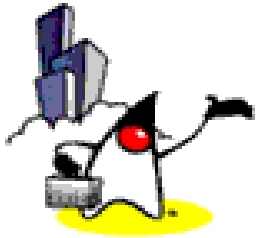
Design for testing - Mock Objects

- When your implementation requires a resource which is unavailable for testing
- External system or database is simulated.
- Another use of Factory, the mock implementation stubs out and returns the anticipated results from a request.

Testing with resources (EJB/DB)

- Use fixtures to request resource connection via factory, could be no-op.
- Use vm args or resource bundle to drive which factory is used.
- Data initialization/clearing handled by fixtures to preserve order independence of tests.

Extensions

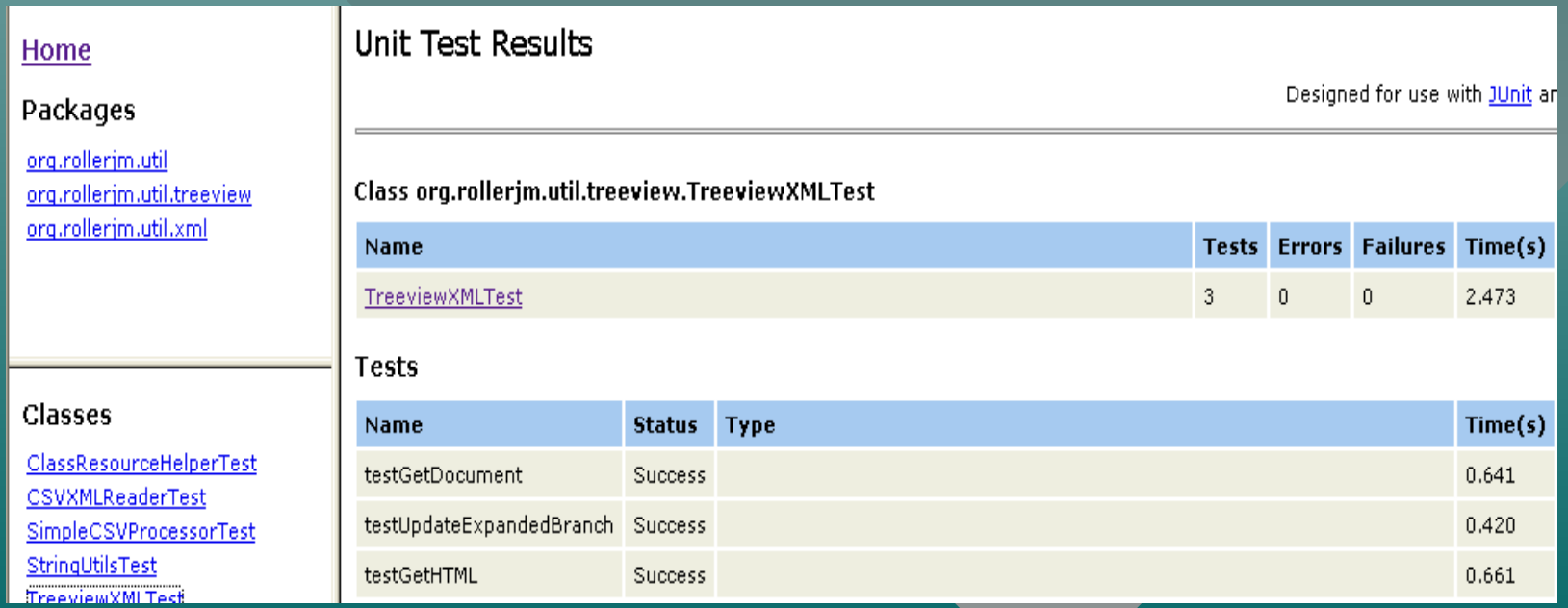


JUnit Extensions

- JUnitReport
- Cactus
- JWebUnit
- XMLUnit
- MockObject
- StrutsTestCase

JUnitReport

- Apache Ant extension task
- Uses XML and XSLT to generate HTML



Home

Packages

org.rollerjm.util
org.rollerjm.util.treeview
org.rollerjm.util.xml

Classes

ClassResourceHelperTest
CSVXMLReaderTest
SimpleCSVProcessorTest
StringUtilsTest
TreeviewXMLTest

Unit Test Results

Designed for use with JUnit ar

Class org.rollerjm.util.treeview.TreeviewXMLTest

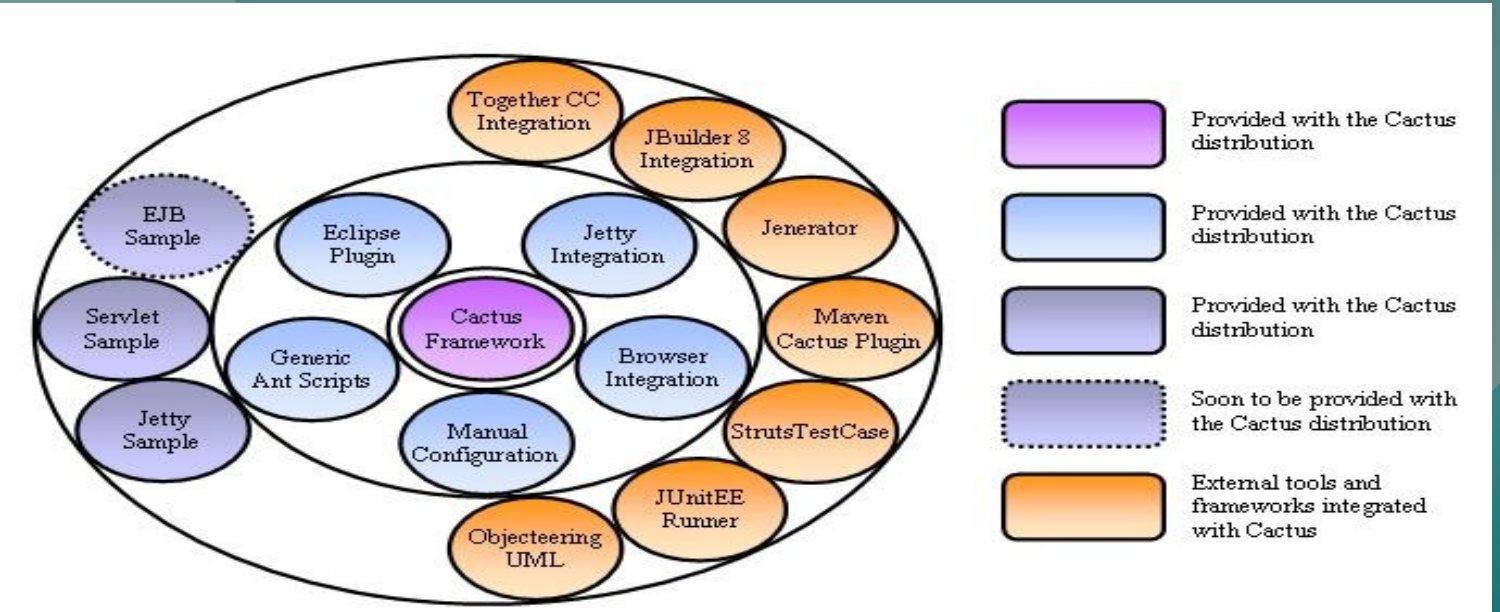
Name	Tests	Errors	Failures	Time(s)
TreeviewXMLTest	3	0	0	2.473

Tests

Name	Status	Type	Time(s)
testGetDocument	Success		0.641
testUpdateExpandedBranch	Success		0.420
testGetHTML	Success		0.661

Cactus (from Jakarta)

- Simple test framework for unit testing server-side Java code



The Cactus Ecosystem

JWebUnit

- Framework that facilitates creation of acceptance tests for web applications

XMLUnit

- Provides an XMLTestCase class which enables assertions to be made about the content and structure of XML
  - Differences between two pieces of XML
  - Validity of a piece of XML
  - Outcome of transforming a piece of XML using XSLT
  - Evaluation of an XPath expression on a piece of XML

Mock Objects

- Generic unit testing framework whose goal is to facilitate developing unit tests in the mock object style
- What is a Mock object?
  - "double agent" used to test the behaviour of other objects
  - Dummy object which mimics the external behaviour of a true implementation
  - observes how other objects interact with its methods and compares actual behaviour with preset expectations

StrutsTestCase

- Extends the JUnit TestCase class that
- Provides facilities for testing code based on the Struts framework
- You can test
  - implementation of your Action objects
  - mappings declarations
  - form beans declarations
  - forwards declarations

JUnit Testing Framework