Beyond JUnit: Introducing TestNG The Next Generation in Testing

Hani Suleiman

CTO
Formicary
http://www.formicary.net

hani@formicary.net

TS 3097

Testing

- Renewed enthusiasm for testing
- No more ‘real developers’ vs. ‘QA developers’
- Partially due to ‘Extreme programming” (XP)
- Test Driven Development (TDD)
- Testing is cool again!

JUnit

- No real introduction needed!
- Simple JavaTM technology testing framework, based on introspection, test methods, classes, and suites.
- First mainstream testing framework created and now a de facto standard.

JUnit Strengths

- Simple to understand: test methods, classes, suites
- Easy to implement: extend TestCase, prefix method with ‘test’
- Use setUp and tearDown for initialization and cleanup
- Use TestRunner for text or graphical results
- Tests can be combined into suites
- Lots of add-ons: db testing, gui testing, reporting

JUnit Problems

Does this test pass or fail?

```
public class Test1 extends TestCase {
  private int count = 0;

  private void test1() {
    count++;
    assertEquals(count, 1);
  }

  public void test2() {
    count++;
    assertEquals(count, 1);
}
```

It Passes!

- JUnit instantiates your class before invoking each test method
  - By design?
- How do you keep state across invocations?
  - Use statics!
- Many downsides:
  - Not ‘same-Java VM’ friendly
  - Redundant with setUp()
  - Goes against intuitive class/constructor behaviour
- Replacing one flaw (reinstantiation) with another (statics)

JUnit Problems

- How do you run an individual method?
  - Comment out all the other ones!
- How do you keep enable/disable certain methods?
  - Modify your suite, recompile, and rerun
- Note: Existing IDE's and JUnit add-ons help address these issues

JUnit Problems

- Victim of its own success. Initially designed for unit testing only, but now used for all sorts of testing
  - Very limited for anything but unit-testing
- Updates are very few and far between
- Intrusive (forces superlcass and ‘magic’ method naming)
- Static programming model (recompile unnecessarily)
- Doesn’t use latest Java technology features (annotations, asserts)

Introducing TestNG

- Annotations (JDKTM 5 software or JavadocTM tool)
- Flexible runtime configuration (xml)
- Introduces ‘test groups’ , to separate statics (test contents) from dynamics (which tests to run)
- Dependent test methods, parallel testing, load testing, partial failure.
- Flexible plug-in API

TestNG Example (JDK 1.4 software)

```
public class SimpleTest {
  /**
   * @testng.configuration beforeTestClass = true
   */
  public void init() {
    // code that will be invoked when test is instantiated
  }
  /**
   * @testng.test groups = "functest"
   */
  public void serverIsRunning() {
    // your test code
  }
}
```

TestNG Example (JDK 5 software)

```
import org.testng.annotations.*;

public class SimpleTest {
  @Configuration(beforeTestClass = true)
  public void init() {
    // invoked when test is instantiated
  }
  @Test(groups = { "functest" })
  public void serverIsRunning() {
    // your test code
  }
}
```

TestNG Example

- No need to extend a base class
- No magic ‘test’ prefix for method names
- Configuration methods can be given meaningful names (not just setUp and tearDown), you can have as many as you want, and they can be around methods, classes, or suites
- Test methods can be parametrized (not shown, but discussed later)

Running the Test

Runtime specified in testng.xml. Mostly: list of classes and list of groups to include/exclude:

```
<test name="Simple">
    <groups>
      <run>
        <include name="functest"/>
      </run>
    </groups>
    <classes>
      <class name="SimpleTest" />
    </classes>
  </test>
```

Note: testng.xml is optional, can use ant/command line

Annotations

- @Test
  - Identify a test method
- @Configuration
  - Identify a method used to configure tests
- @ExpectedExceptions
  - Indicate that a method is expected to throw one or more exceptions
- @DataProvider
  - Provide parameters to pass to test methods
- @Factory
  - Create your own test objects at runtime

TestNG Terminology

- Suite—each suite contains:
- Tests—each test contains:
- Classes—each class contains:
- Methods
- @Configuration can wrap each of the scopes above

testng.xml

```
<suite name="TestNG JDK 1.5" parallel="true" thread-count="5">
  <test name="Nopackage">
    <classes>
      <class name="NoPackageTest1" />
      <class name="NoPackageTest2" />
    </classes>
  </test>
  <test name="Regression1" >
    <packages>
      <package name="test.regression.*" />
    </package>
  </test>
</suite>
```

TestNG Annotations

- @Configuration
  - beforeTestMethod/afterTestMethod
    - JUnit: setup/tearDown
  - beforeTestClass/afterTestClass
    - JUnit: No equivalent
  - beforeSuite/afterSuite
    - JUnit: No equivalent
  - beforeTest/afterTest
    - Junit: No equivalent
- You can have multiple @Configuration methods
- @Configuration methods can be grouped/have dependencies

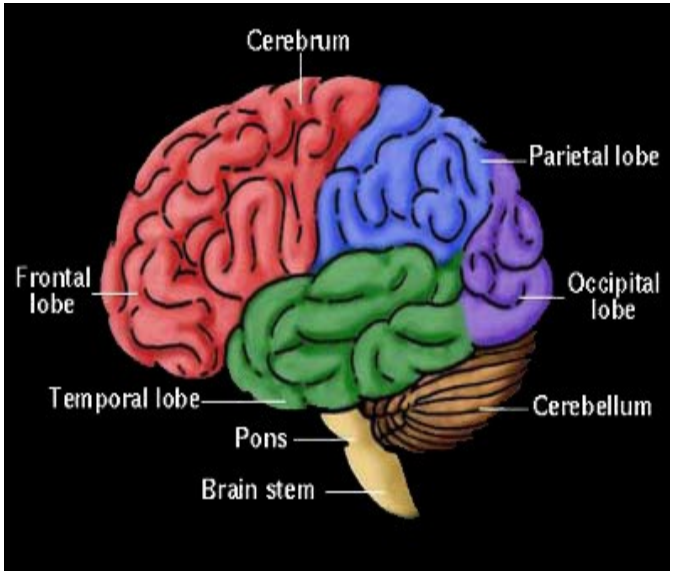
TestNG Annotations

- @Test
  - Groups: The groups this method belongs to
  - Parameters: The parameters that will be passed to your test methods, as declared in testng.xml
  - DependsOnGroups: The list of groups this method depends on
  - Timeout: The maximum duration of this test before it is considered a failure

```
@Test(groups = { "functional" }, timeOut = 10000,
      dependsOnGroups = "serverIsUp")
public void sendHttpRequest() {
  // …
}
```

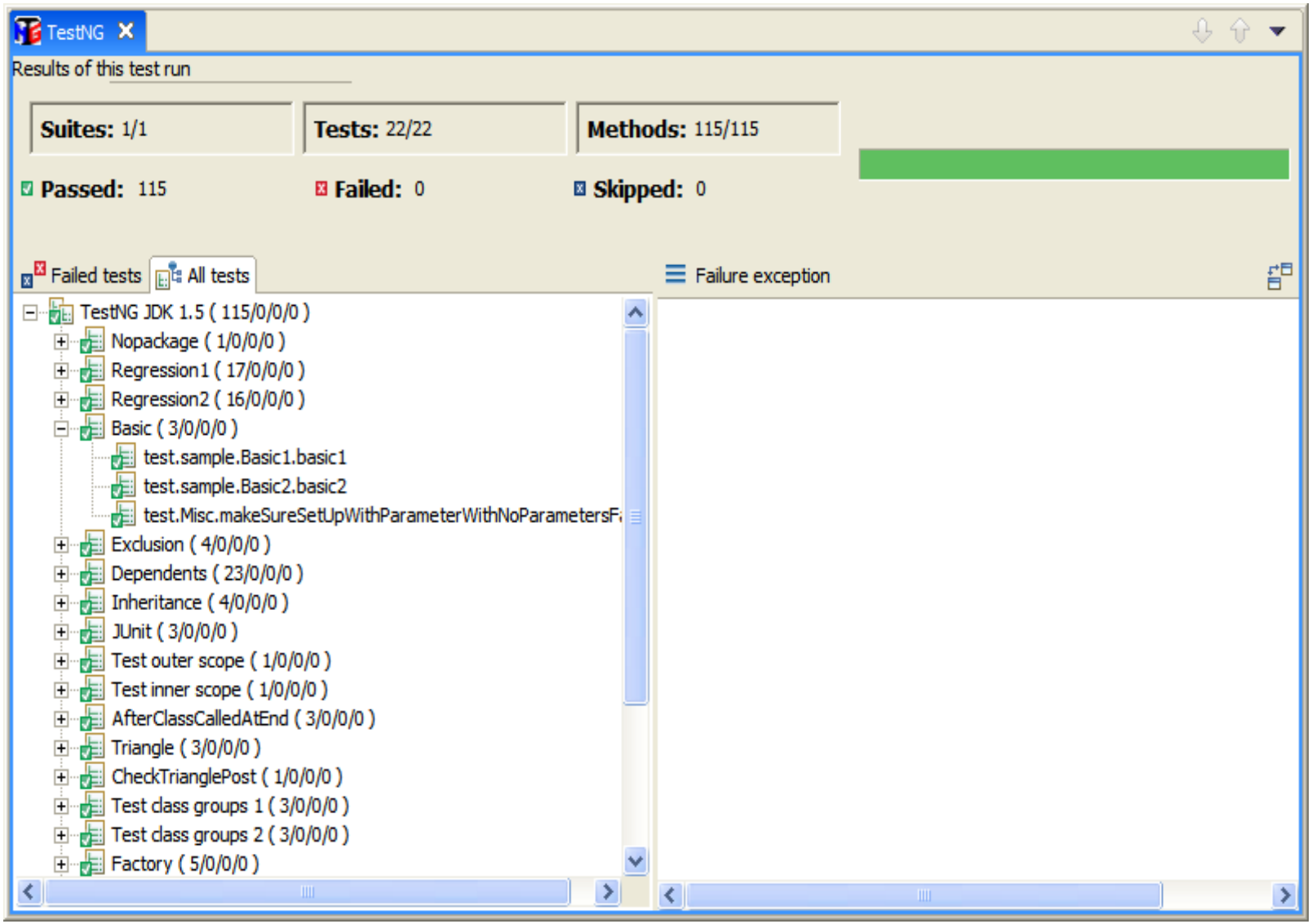
testng.xml

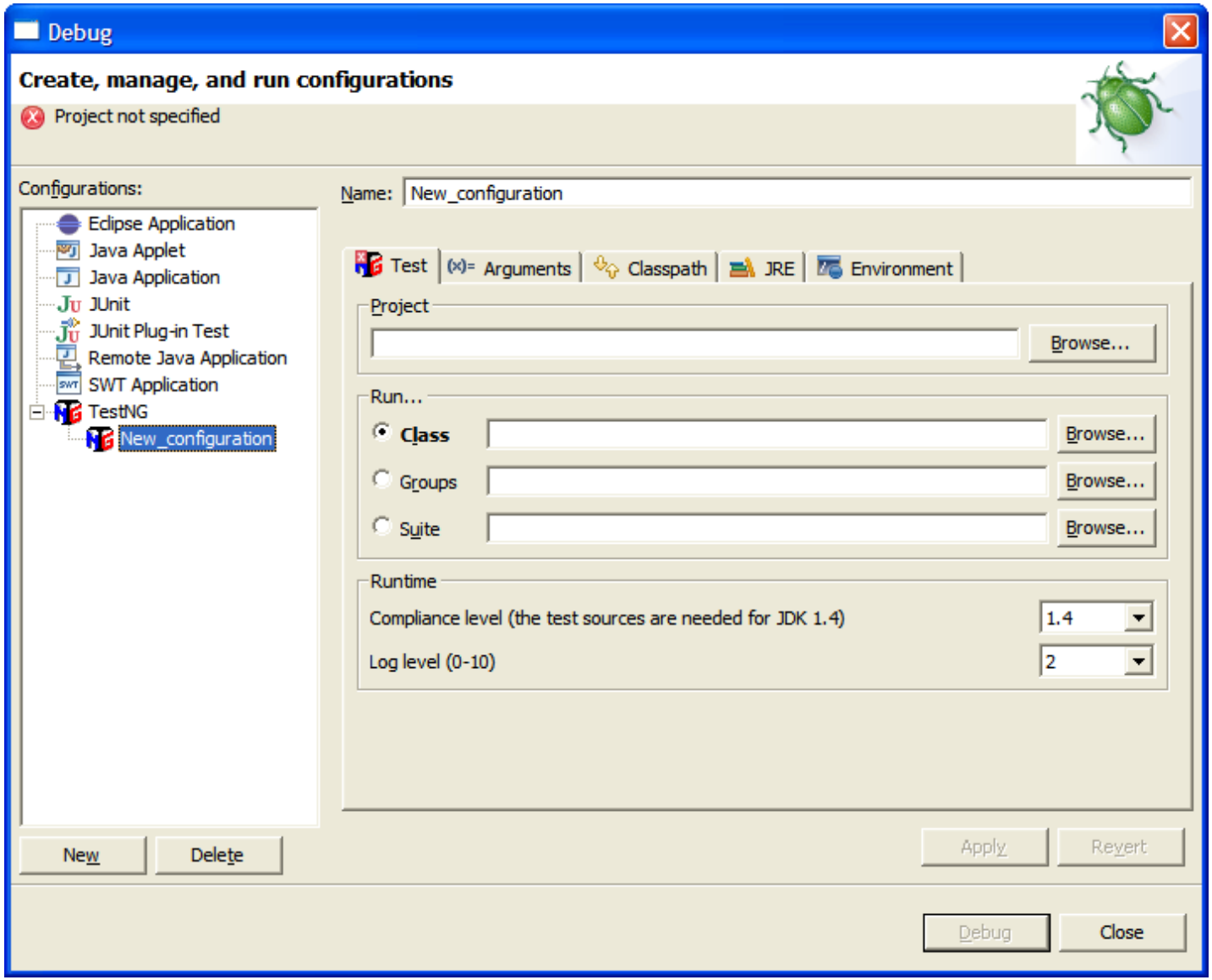
- Where all runtime information goes:
  - Test methods, classes, and packages
  - Which groups should be run (include-groups)
  - Which groups should be excluded (exclude-groups)
  - Define additional groups (groups of groups)
  - Whether tests should be run in parallel
  - Parameters
  - JUnit mode

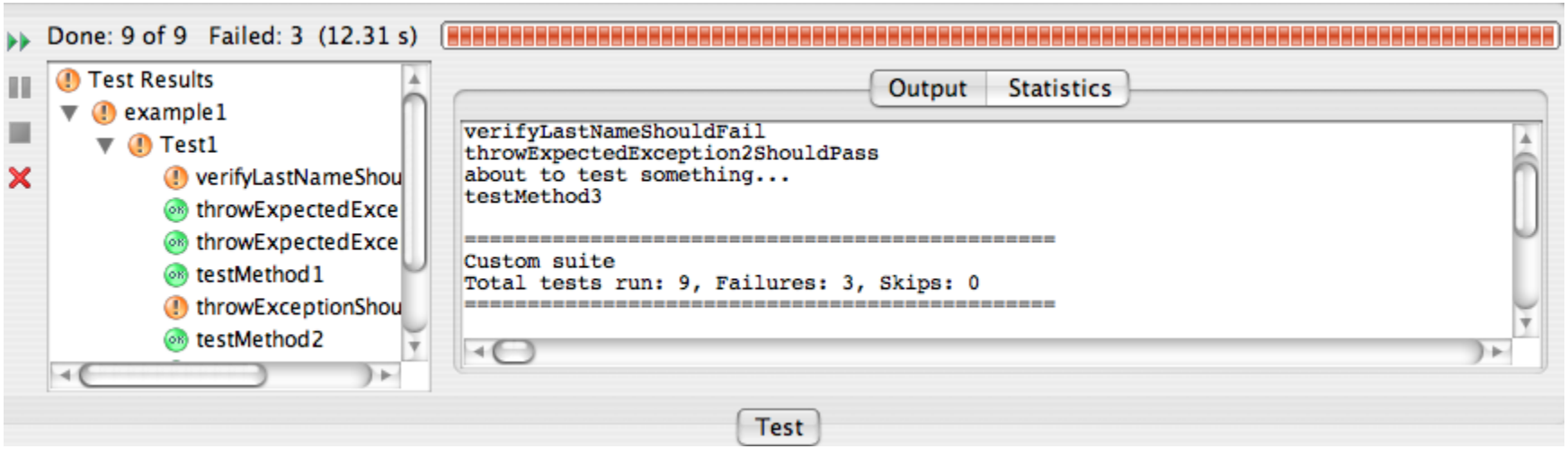


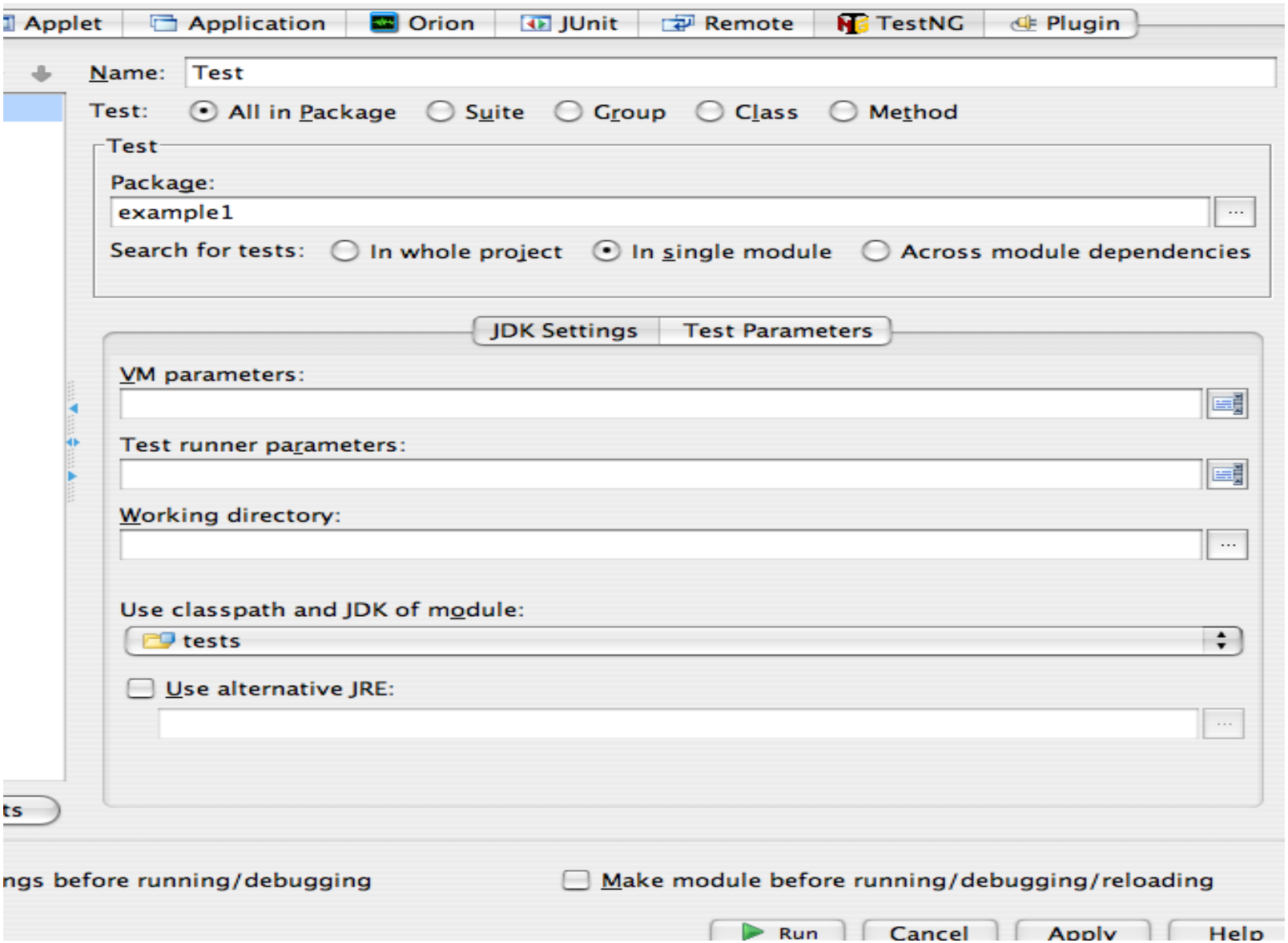
Eclipse and IDEA

- TestNG Plug-in exists for Eclipse and IDEA
- Run a test class, method, or group
- Easy selection of groups and testng.xml files
- Visual feedback (red/green bar, just like JUnit!)
- Directly jump to failures
- ‘quickfix/intentions’ to one-click migrate JUnit tests


TestNG
Results of this test run
Suites: 1/1
Tests: 22/22
Methods: 115/115
Passed: 115
Failed: 0
Skipped: 0
Failed tests
All tests
Failure exception
TestNG JDK 1.5 (115/0/0/0)
Nopackage (1/0/0/0)
Regression1 (17/0/0/0)
Regression2 (16/0/0/0)
Basic (3/0/0/0)
test.sample.Basic1.basic1
test.sample.Basic2.basic2
test.Misc.makeSureSetUpWithParameterWithNoParametersF
Exclusion (4/0/0/0)
Dependents (23/0/0/0)
Inheritance (4/0/0/0)
JUnit (3/0/0/0)
Test outer scope (1/0/0/0)
Test inner scope (1/0/0/0)
AfterClassCalledAtEnd (3/0/0/0)
Triangle (3/0/0/0)
CheckTrianglePost (1/0/0/0)
Test class groups 1 (3/0/0/0)
Test class groups 2 (3/0/0/0)
Factory (5/0/0/0)


Debug
Create, manage, and run configurations
Project not specified
Configurations:
Eclipse Application
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Remote Java Application
SWT Application
TestNG
New_configuration
Name: New_configuration
Test
Arguments
Classpath
JRE
Environment
Project
Browse...
Run...
Class
Groups
Suite
Browse...
Browse...
Browse...
Runtime
Compliance level (the test sources are needed for JDK 1.4)
1.4
Log level (0-10)
2
New
Delete
Apply
Revert
Debug
Close


Done: 9 of 9 Failed: 3 (12.31 s)
Test Results
example1
Test1
verifyLastNameShou
throwExpectedExce
throwExpectedExce
testMethod1
throwExceptionShou
testMethod2
Output
Statistics
verifyLastNameShouldFail
throwExpectedException2ShouldPass
about to test something...
testMethod3
Custom suite
Total tests run: 9, Failures: 3, Skips: 0
Test


Applet
Application
Orion
JUnit
Remote
TestNG
Plugin
Name: Test
Test: All in Package Suite Group Class Method
Test
Package:
example1
Search for tests: In whole project In single module Across module dependencies
JDK Settings
Test Parameters
VM parameters:
Test runner parameters:
Working directory:
Use classpath and JDK of module:
tests
Use alternative JRE:
ngs before running/debugging
Make module before running/debugging/reloading
Run
Cancel
Apply
Help

Integration with Other Frameworks

- Maven plugin (v1 and v2)
- Spring framework
- Glassfish Unified Test Framework
  - TestNG Based
  - Development Ongoing
  - Quality Portal at http://wiki.java.net/bin/view/Projects/GlassFishQuality
  - Mailing List: quality@glassfish.dev.java.net
  - Comparisons with JUnit 4
- Integration is straightforward in most cases (setUp/tearDown)

Converting from JUnit

- JUnitConverter can convert entire codebase to TestNG in a few seconds
- Plugin in Eclipse or IDEA can also do so

Expected Exceptions

- Throwing exceptions is common from Java code
- Easy to test with TestNG:

```
@ExpectedExceptions({
   java.lang.NumberFormatException.class })
@Test
public void validateNumber() {
   …
}
```

Rerunning Failed Tests

- Most of our work is fixing tests that fail
- TestNG knows which tests failed in a run and makes it easy to rerun just these tests.
  - testng-failed.xml

- Typical session:

```
$ java org.testng.TestNG testng.xml
$ java org.testng.TestNG testng-failed.xml
```

DataProviders

- TestNG supports Data-Driven testing
- Example: Testing a parser

```
@Test
public void parseGoodString(String s) {
 new Parser().parse(s);
}
@ExpectedExceptions({ ParserException.class })
@Test
public void parseBadString(String s) {
 new Parser().parse(s);
}
```

DataProviders

```
@Test(dataProvider = "good-strings")
public void parseGoodString(String s) {
  new Parser().parse(s);
}
@DataProvider(name = "good-strings")
public Object[][] createGoodStrings() {
  return new Object[][] {
    new Object[] { "2 * 2" },
    new Object[] { "3 + 2"  };
   }
}
```

DataProviders

- DataProviders allow you to separate data from logic
- Data can come from Java technology, flat file, database, network, etc.
- You can have as many DataProviders as you like; for example, “good-strings”, “bad-strings”, etc.

Testing Thread Safety

- Not even sure how to do this in JUnit:

```
public void testCachePutShouldBeThreadSafe() {
  // Create pool of threads (5? 10? 50?)
  // Create workers invoking cache.put() (100? 200?)
  // Launch the pool
  // Wait for termination
  // Abort laggards (exception if any)
  // If no exception verify that no data has been thrashed
}
```

Phew!

Testing Thread Safety

- TestNG:

```
@Test(invocationCount=1000, threadPoolSize=10)
public void cachePut() {
  m_cache.put("foo", "bar");
}
```

Need a timeout?

```
@Test(invocationCount=1000, threadPoolSize=10,
      timeOut=10000 /* 10 seconds */)
```

Excluding Groups

- Sometimes, tests break and you cannot fix them immediately
- JUnit: comment them out or rename method, and with luck, turn them back on before shipping
- TestNG: define “broken” group and have it excluded in all test runs. Move any fails to this group
- Later: Look for tests in “broken” group and fix.

```
<test name="DBTest">
    <groups>
      <run>
        <exclude name="broken.*" />
        <include name="functional.*" />
      </run>
    </groups>
```

Programmatic Invocation

- Convenient for in-container testing

```
TestNG testng = new TestNG();
testng.setTestClasses(new Class[] {
  Run1.class, Run2.class
});
testng.setGroups(new String[] { "functional", "db"});
TestListenerAdapter tla = new TestListenerAdapter();
testng.addListener(tla);
testng.run();
// inspect results in tla
```

Beanshell

- Sometimes, including and excluding groups is not enough
- What if we need to run certain tests during the week, and a different set during the weekend?
  - Use beanshell!

```
<test name="Basic">
  <script language="beanshell"><![CDATA[
     int today = Calendar.getInstance().
       get(Calendar.DAY_OF_WEEK);
    return today == Calendar.SATURDAY ||
                today == Calendar.SUNDAY;
  ]]></script>
```

Partial Failure

- “invocationCount” allows us to specify the number of times to execute a test.
- Used with “successPercentage” allows us to define partial failure tests:

```
@Test(invocationCount = 1000,
      successPercentage = 98)
public void sendSmsMessage(String msg)
{ … }
```

Plug-in API

- TestNG exposes a plug-in API that makes it easy to monitor test progress and invocation:
  - Test started
  - Test ended
  - Test result…etc
- Possible to also modify TestNG core
- Four proofs of concept:
  - JUnit mode
  - Default HTML reports
  - JUnitReport HTML plug-in
  - TestNG's own testing

Dependent Methods

- Problem: certain methods depend on the success of previous methods
- Example: testing a server:
  - One test method to launch the server: launch()
  - One test method to ping the server: ping()
  - Twenty methods to verify server functionality test1()…test20()
- Problem: Server is launched but ping() fails
- Scenario difficult to achieve with Junit
- Result: 1 PASSED and 21 FAILURES
- Result: QA freaks out and calls you on Sunday during your golf game

Dependent Methods

- Need a way to order methods;
  not just individual methods, but collections of methods grouped logically
- Need a mechanism to accurately report failures due to failed dependency (avoid PRFTS: Post Run Failure Trauma Syndrome)

Dependent Methods

- Back to the server testing example:
  - Dependencies: launch → init → tests

```
@Test(groups = "launch")
public void launchServer() {…}
@Test(groups = "init", dependsOnGroups = { "launch" })
public void ping() { …}
@Test(dependsOnGroups = { "init" })
public void test1() { … }
```

- Outcome: 1 SUCCESS, 1 FAIL, 20 SKIPS

Reporting

- TestNG issues an HTML report by default
- Plug-in API makes it easy to write your own report (for example, JUnitReporter plug-in)

Results for TestNG JDK 1.5

34 tests, 22 groups, 172 methods, 96 classes.

- Abstract classes (2/0/0) Results
- AfterClassCalledAtEnd (3/0/0) Results
- Basic (3/0/0) Results
- CheckTrianglePost (2/0/0) Results
- DataProvider (0/0/0) Results
- Dependents (28/0/0) Results
- Excluded methods (3/0/0) Results
- Exclusion (4/0/0) Results
- Factory (9/0/0) Results

Dependents

Tests passed/Failed/Skipped: 28/0/0
Started on: Fri Nov 18 18:53:35 PST 2005
Total time: 0 seconds
Included groups:
Excluded groups:

PASSED TESTS

Test method	Time (seconds)	Exception
test.dependent.BaseOrderMethodTest.a_second0()	0	none
test.dependent.OrderMethodTest.a_second1()	0	none
test.dependent.SampleDependentMethods.canBeRunAnytime()	0	none
test.dependent.SampleDependentMethods2.canBeRunAnytime()	0	none
test.dependent.DependentTest.dependentMethods()	0	none
		org.testng.TestNGException: Cyclic graph of methods at org.testng.interne at org.testng.interne at org.testng.interne at org.testng.interne at org.testng.interne at org.testng.TestRur at org.testng.TestRur at org.testng.TestRur at org.testng.TestRur at test.BaseTest$Inte at org.testng.SuiteRu at org.testng.SuiteRu at org.testng.SuiteRu

Summary

- JUnit has the right idea, but suffers from old age and limitations for real (non-unit) testing.
- TestNG offers the following benefits:
  - Non-intrusive (annotations)
  - Clean separation of programming model from runtime
  - Covers more than unit testing, with features like dependencies, groups, parameters, and partial failure
  - Powerful plugin API for custom reports or modifications to core functionality
- Whether you choose TestNG or JUnit, think differently about testing!

For More Information

- Hosted on java.net, available via CVS
- Download and documentation available on http://testng.org
- Authors:
  - Cedric Beust (cbeust@google.com)
  - Alexandru Popescu (the_mindstorm@evolva.ro)

Q&A

Beyond JUnit: Introducing TestNG The Next Generation in Testing

Hani Suleiman

CTO
Formicary
http://www.formicary.net

hani@formicary.net

TS 3097