Software Development Tools

COMP220

Seb Coope

Ant, Testing and JUnit (1)

These slides are mainly based on “Java Development with Ant” - E. Hatcher & S.Loughran. Manning Publications, 2003 and “JUnit in Action”, 2nd edition – P. Tahchiev, F. Leme, V. Massol, G. Gregory, Manning Publications, 2011

Testing with JUnit and ANT

"Any program without an automated test simply doesn't exist."

(Extreme Programming Explained, Kent Beck)

- *Software bugs have enormous costs* :

  time, money, frustration, and even lives.

- Creating and *continuously executing test cases* is

  a practical and common *approach to address software bugs*.

- The JUnit testing framework is now the *de facto standard unit testing* API for Java development.

Testing with JUnit and ANT

- You know that Eclipse *integrates* with Junit.
- Ant also *integrates* with Junit. This allows:

  executing tests as part of the build process,

  capturing their output, and

  generating rich colour enhanced *reports* on testing.
- The following several lectures on Ant is eventually aimed to show how to use JUnit from Ant for testing.
- But we will start with JUnit Primer.

JUnit primer
(independent of Eclipse and Ant)

- JUnit is an API that enables developers to easily create *Java test cases*.
- It provides a comprehensive *assertion* facility to verify *expected* versus *actual* results.

Writing a test case (simplified version)

- Let us create a simple *JUnit test case*, e.g. `SimpleTest.java`, (a special version of Java class). Follow *three simple steps*:

1. Import the necessary JUnit classes (see Slide 7) such as

```
import static org.junit.Assert.*;

import org.junit.Test;
```

2. Implement one or more no-argument `void` *methods* `testXXX()` *prefixed by the word* `test` and *annotated* as `@Test`
3. Implement these `@Test` methods by using assertion methods

- *Compare* these steps with the *example* in the next Slide

An example of `SimpleTest` test case:

Create the following file under `C:\Antbook\ch04\test` directory.

- `C:\Antbook\ch04\test\org\example\antbook\junit\SimpleTest.java:`

```
package org.example.antbook.junit;

//import required JUnit4 classes:
import static org.junit.Assert.*;
import org.junit.Test;

public class SimpleTest
{
    @Test
    public void testSomething()
    {
        assertTrue("MULTIPLICATION???",
                   4 == (2 * 2));
    }
}
```

Any your package

Import JUnit classes

Implement a `@Test` annotated `void testXXX()` method

Use *assertion* methods (to be further discussed later).

Imported JUnit Classes

Find the imported classes `org.junit.Assert` and `org.junit.Test` in

`C:\JAVA\junit4.8.2\junit-4.8.2.jar` and

`C:\JAVA\junit4.8.2\junit-4.8.2-src.jar`.)

by using commands

```
jar tf junit-4.8.2.jar
```
or

```
jar tf junit-4.8.2-src.jar,
```

or by using WinZip (with renaming the extension `.jar` by `.zip`;

before renaming, copy these JAR files into some other directory!)

Running a test case

- How to run a test case such as `SimpleTest`?

(~~Recall, this~~ was very easy in Eclipse!)

Note that `SimpleTest` does *not* contain `main` method!
So, it cannot be run in itself.

- Junit 4 provides us with Test Runner Java classes used to execute all `@Test` methods `testXXX()`

Prefixing these method names by the word "`test`" is unnecessary, but it is a good tradition.
Test Runners will run only methods *annotated* as `@Test`, irrespectively how the methods are named.

See Slides 19,20 below on *other* possible JUnit *annotations*.

Running a test case

- JUnit 4 provides different runners for running old style Junit 3.8 tests, new style JUnit 4 tests, and for different kinds of tests.
- The JUnitCore "facade" (which we will actually use)

  `org.junit.runner.JUnitCore`

  operates *instead of* any test runner.

  It determines which runner to use for running your tests.

  It supports running JUnit 3.8 tests, JUnit 4 tests, and mixture of both.
- See also `http://junit.sourceforge.net/javadoc/`

Running a test case (cont.)

- `JUnitCore` Test Runner expects a *name* of a Test Case class as *argument*.
- Then *all the methods* annotated as `@Test` of this subclass (typically having the name like `testXXX()`) and having no arguments *are running*.
- Methods not annotated by `@Test` will not run!
- Test Runner

  prints a *trace of dots* (.....) at the console as the tests `testXXX()` are executed

  followed by a *summary* at the end.

Running a test case (cont.)

- *Compiling* our `SimpleTest.java` test case to directory `build\test` (create it yourselves), and then
- *running* the `JUnitCore` Test Runner from the command line, with `SimpleTest.class` as *argument* goes *as follows:*

```
C:\Antbook\ch04>javac -d build\test tes
antbook\junit\SimpleTest.java

C:\Antbook\ch04>java
-cp build\test;C:\JAVA\junit4.8.2\junit-4.8.2.jar org.
junit.runner.JUnitCore org.example.antbook.junit.
SimpleTest
JUnit version 4.8.2
.
Time: 0.01
OK (1 test)
```

Compiling destination

what to compile

- The *dot character* (.) indicates *one* `@Test` *method* `testXXX()` *being run successfully*.
- In this example *only one* `@Test` *method exists*, `testSomething`. TRY this with several `testXXX()` methods (annotated as `@Test` or not) or with several `assertTrue`. How many dots will you see?

Running a test case (-classpath)

```
C:\Antbook\ch04>javac -d build\test test\org\example\
antbook\junit\SimpleTest.java


C:\Antbook\ch04>java
-cp build\test;C:\JAVA\junit4.8.2\junit-4.8.2.jar org.
junit.runner.JUnitCore org.example.antbook.junit.
SimpleTest
JUnit version 4.8.2
.
Time: 0
OK (1 test)
```

`-cp (-classpath)` overrides system `CLASSPATH` environment variable!

That is why, besides the location

`build\test`

of `org.example.antbook.junit.SimpleTest`, the path

`C:\JAVA\junit4.8.2\junit-4.8.2.jar`

to JUnit JAR file containing `JUnitCore` (which will run `SimpleTest`) is also necessary (even if it was in `CLASSPATH`).

Directory Structure in `ch04`

The *directory structure* in `ch04` is as follows:

`C:\Antbook\ch04` - *base* directory (`basedir="."`)

`C:\Antbook\ch04\src` - *source* directory (`${src.dir}`)

`C:\Antbook\ch04\test` - *test* directory (`${src.test.dir}`) containing (deeper) *JUnit test classes*

`C:\Antbook\ch04\build` - build directory (`${build.dir}`)

`C:\Antbook\ch04\build\classes` - for *compiled source files* (`${build.classes.dir}`)

`C:\Antbook\ch04\build\test` - for *compiled JUnit classes* (`${build.test.dir}`; to be considered later).

`Red coloured` (underlined) directories and their content should be created by yourself. Other highlighted directories `build\classes` and `\build\test` will be created automatically by your Ant build file.

Invoking Test Runner from build file with <java> task

It is more convenient to apply Test Runner `JUnitCore` to `SimpleTest` *from* Ant build file `mybuild.xml` (in `C:\Antbook\ch04`) *containing*

```
<path id="test.classpath">
   <pathelement location="${build.test.dir}"/>
   <!-- More path elements? Add yourself! -->
</path>

<target name="junit-TestRunner"
        depends="test-compile">
  <java classname="org.junit.runner.JUnitCore"
        classpathref="test.classpath">

    <arg value="org.example.antbook.junit.
  </java>
</target>
```

`build\test` for *compiled* test cases. Should also be created in `mybuild.xml` by some target `test-init`!

This target is also required before running `SimpleTest`! Which else?

Test Runner

Class path. It may also be required in target `test-compile`! We will see!

Test Case to run

We name the target as `junit-TestRunner` because it *imitates* command-line execution of Test Runner `JUnitCore` with the *argument* `SimpleTest`:

```
ch04>java -cp build\test;C:\JAVA\junit4.8.2\junit-4.8.2.jar
org.junit.runner.JUnitCore org.example.antbook.junit
SimpleTest
```

See Slides 11,12

Invoking Test Runner from build file with <java> task in the Lab

- Continue working yourselves on mybuild.xml in C:\Antbook\ch04

  Set additional essential *properties* in mybuild.xml for all required directories (like src, test, build\classes, build\test, etc). See Slide 13.

  Use always these properties in mybuild.xml instead of usual directory names. This is a good practice.

  To avoid usual misprints, copy-and-paste long property names.

  Create other necessary *targets* (using templates from old files)

  test-init, test-compile, clean, etc.

  Complete definitions of the *path* with id="test.classpath", if required,

  Check carefully all the relevant details in mybuild.xml, and

  RUN the above target junit-TestRunner

  with preliminary cleaning build directory:

Invoking `TestRunner` from build file with `<java>`

After completing definition of the *path* with `id="test.classpath"`:

```
C:\Antbook\ch04>ant -f mybuild.xml clean junit-TestRunner
Buildfile: C:\Antbook\ch04\mybuild.xml
     [echo] Building Testing Examples
clean:
   [delete] Deleting directory C:\Antbook\ch04\build
init:
    [mkdir] Created dir: C:\Antbook\ch04\build\classes
compile:

test-init:
    [mkdir] Created dir: C:\Antbook\ch04\build\test
test-compile:
    [javac] Compiling 1 source file to C:\Antbook\ch04\build\test

junit-TestRunner:
     [java] JUnit version 4.8.2
     [java] .
     [java] Time:  0.016
     [java]
     [java] OK (1 test)
     [java]
     [java] Java Result: 1
BUILD SUCCESSFUL
Total time: 2 seconds
```

Currently no Java files in `src`. Nothing to compile.

There is currently 1 test file `SimpleTest.java` in `ch04\test` *compiled* to `ch04\build\test`

Ignore

Asserting desired results

- The mechanism by which JUnit determines the success or failure of a test is via *assertion statements* like

```
assertTrue(4 == (2 * 2))
```

- Other *assert* methods simply
  - compare between expected value and actual value, and
    generate appropriate messages helping us to find out why a method does not pass the test.
- These expected and actual values can have any of the *types:*
  any *primitive datatype*,
  `java.lang.String`,
  `java.lang.Object`.

- For each type there are two *variants of the assert methods*, each with the *signatures* like

```
assertEquals(expected, actual)
assertEquals(String message, expected, actual)
```

- The *second signature* for each datatype allows a `message` to be inserted into the results of testing in *case of failure*.
  It can help to identify which assertion failed.

Some JUnit Assert Statements

based on http://www.vogella.de/articles/JUnit/article.html

Statement	Description
assertTrue([String message], boolean condition)	Check if the boolean condition is true.
assertEquals([String message], expected, actual)	Test if the values are equal: `expected.equals(actual)` Note: for arrays the reference is checked not the content of the arrays
assertArrayEquals([String message], expected, actual)	Asserts the equality of two arrays (of their lengths and elements)
assertEquals([String message], expected, actual, tolerance)	Usage for float and double; the tolerance is the maximal allowed difference between expected and actual.
assertSame([String], expected, actual)	Check if both variables refer to the same object `expected == actual`
assertNotSame([String], expected, actual)	Check that both variables refer not to the same object.
assertNull([message], object)	Checks if the object is null
assertNotNull([message], object)	Check if the object is not null.
fail([message])	Lets the test method fail; might be usable to check that a certain part of the code is not reached.

Some Junit4 Annotations

based on http://www.vogella.de/articles/JUnit/article.html

Annotation	Description
@Test public void method() See also Slides 5,6 above.	Annotation @Test identifies that this method is a test method.
@Before public void method() See also Slides 4-6 in part 12. Ant and JUnit. of these lectures.	Will perform the method() before each test. This method can prepare the test environment, e.g. read input data, initialize the class) This method() is usually called setUp().
@After public void method() See slides as above.	This method() must start after each @Test method This method() is usually called tearDown().
@BeforeClass public void method()	Will perform the method before the start of all tests. This can be used to perform time intensive activities for example be used to connect to a database
@AfterClass public void method()	Will perform the method after all tests have finished. This can be used to perform clean-up activities for example be used to disconnect to a database
@Ignore	Will ignore the test method. E.g. useful if the underlying code has been changed and the test has not yet been adapted or if the runtime of this test is just too long to be included. Console will show "I" instead of dot ".".
@Test(expected=IllegalArgumentException.class)	Tests if the method throws the named exception
@Test(timeout=100)	Fails if the method takes longer then 100 milliseconds

Some Junit4 Annotations

Annotation	Description
@RunWith(value=Suite.class)	Annotation @RunWith of a test class declaration says which Test Runner (here `org.junit.runners.Suite.class`) should be used by `JUnitCore` facade to run this test class. See Junit in Action 2nd Ed., pages 21-23 and also Slides 9, 10 in Part 12. Ant and JUnit. of these lectures.
@SuiteClasses(value={FirstTest.class, SecondTest.class,...}) Public class AllTests{}	Annotation @SuiteClasses is used to create a Suite of Tests Classes(or Suite of Suites). Here, the Suite created is called AllTests. The above annotation @ RunWith(value=Suite.class) should be also used here before @SuiteClasses. See Junit in Action 2nd Ed., pages 21-23 and also Slides 9 , 10 in Part 12. Ant and JUnit. of these lectures.
@Parameters Details and examples of @Parameters not considered in these lectures	Annotation @Parameters is used to create a Collection of arrays of parameters so that a test can be applied to each of this parameter array. Arrays must be of identical length to be substitutable for variables used in the test. The class of such a ParametrizedTest should be annotated as @RunWith(value=Parametrized.class). More details in Junit in Action 2nd Ed., pages 17-19.

Failure or error?

- JUnit uses the term *failure* for a test that fails expectedly, meaning that

  an assertion (like those above) was not valid or

  a `fail()` was encountered.

  We *expect* these failures and therefore set up these assertions.

- The term *error* refers to an unexpected error (such as a `NullPointerException` or the like).

Running in Ant JUnit test case for `FilePersistenceServices.java`

Recall that while working on "Eclipse and JUnit" the class

`FilePersistenceServices.java`

for *writing* and *reading* a data to/from a file was partly implemented and tested by JUnit test case

`FilePersistenceServicesTest.java`

Both classes declared joint `package` (in your case – your personal)

```
package org.eclipseguide.persistence;
```

Let us copy them from your Eclipse workspace, respectively, to:

`C:\Antbook\ch04\src\ org\eclipseguide\persistence\ FilePersistenceServices.java`

and

`C:\Antbook\ch04\test\`

`org\eclipseguide\persistence\FilePersistenceServicesTest.java`

You should use as complete as possible versions of these files created in lectures and your Labs!

java
and FilePersistenceServicesTest.java

C:\Antbook\ch04\src\ org\eclipseguide\persistence\
FilePersistenceServices.java

```
package org.eclipseguide.persistence;

import java.util.StringTokenizer;
import java.util.Vector;

public class FilePersistenceServices
{
   public static boolean write(String fileName, int key, Vector<String> v)
   {
    return false; // false: not yet implemented
   }
   public static Vector<String> read
                     (String fileName, int key)
   {
    return null; // null: just to return anything (not yet implemented)
   }
```

We omitted the most of the details of this file, except red coloured package declaration and method names. Please, recall what they were intended to do.

But you should use the complete version of this file!

Continued on the next slides...

Continuation

```
   public static String vector2String(Vector<String> v, int key)
   {
    return null; // null: just to return anything (not yet implemented)
   }

   public static Vector<String> string2Vector(String s)
   {
    return null; // null: just to return anything (not yet implemented)


   }
   public static int getKey(String s)
   {
    return -1; // -1: just to return anything (not yet implemented)
             // should return key, a positive integer;
   }
}
```

End of file

You should use the complete versions of these three methods!

In the real file, methods `vector2String`, `string2Vector` and `getKey` were fully implemented in the Labs and returned something more meaningful than `null`.

If you implemented these three methods correctly, they even should pass our tests!

C:\Antbook\ch04\test\org\eclipseguide\persistence\FilePersistenceServicesTest.java

Read and do this yourself in the lab

JUnit test case for testing FilePersistenceServices.java

```
package org.eclipseguide.persistence;
// Junit4 packages:
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.util.Vector;

public class FilePersistenceServicesTest
{
   Vector<String> v1;
   String s1 = "\"1\",\"One\",\"Two\",\"Three\"";

   @Before    //Runs before each @Test method
   public void setUp() throws Exception
   {v1 = new Vector<String>();
    v1.addElement("One"); v1.addElement("Two"); v1.addElement("Three");
   }
   @After        //Runs after each @Test method
   public void tearDown() throws Exception { v1 = null; }
```

Setting up the fixture v1

Releasing the fixture

(continued on the next slide)

```
@Test
public void testWrite()
{
  // fail("Not yet implemented");
  assertTrue("NOT WRITTEN???",
          FilePersistenceServices.write("TestTable", 1, v1));
}
@Test
public void testRead()
{
  // fail("Not yet implemented");
  FilePersistenceServices.write("TestTable", 1, v1);
  Vector<String> w = FilePersistenceServices.read("T
  // assertNotNull("NULL OBTA
  assertEquals(v1, w);
}
@Test
public void testVector2String()
{ assertEquals(s1, FilePersistence 2String(v1, 1)); }
@Test
public void testString2Vector()
{ assertEquals(v1, FilePersistence 2Vector(s1)); }
@Test
public vo
{ assertE rsistenc s.getKey(s1)); }
```

Test method

Read and do this yourself in the lab

Test method

Correction: this line commented

Test method

Test method

Test method

1 is expected

actual

Enf of file

Comments

- Running the unit test `FilePersistenceServicesTest` now should evidently fail on its test methods `read()` and `write()` which are still wrongly implemented,

  until we *provide correct implementation* of all the tested methods.
- We are omitting the implementation details of `read()` and `write()` as this is beyond the scope of the testing *tools*.
- However, this is, of course, in the scope of the testing *practice* for which we have insufficient time in this module COMP220 (although we had some such a practice while working with Eclipse).
- NOW, COMPILE and then RUN `FilePersistenceServicesTest` by using JUnit `Test Runner` *from* Ant's build file `mybuild.xml` *appropriately extended.*

Extending `path id="test.classpath"` >

- Now, after above copying, we have two more classes, one in `src`, and another in `test` directories.
- Let us add new *argument* `FilePersistenceServicesTest` to `<java>` in target `unit-TestRunner` and repeat the command

```
ant -f mybuild.xml clean junit-TestRunner
```

from Slide 16 above.

Added line

```
test-compile:
    [javac] Compiling 2 source files to C:\Antbook\ch04\build\test
    [javac] C:\Antbook\ch04\test\org\eclipseguide\persistence\
FilePersistenceServicesTest.java:63:  cannot find symbol
[javac] symbol : variable FilePersistence[illegible]ices
```

- Note that BUILD FAILED since compiling does not work for new files now. Why? (In particular `unit-TestRunner` will not start.)

It looks like the compiler requires(!?) and does not know where to find `FilePersistenceServices.class`

Hence, we should further extend `<path id="test.classpath>` from Slide 14. HOW? DO IT to have the build success.

For Lab: Solution to the previous slide

Extend in `mybuild.xml` both `<path id="test.classpath">` element and target `test-compile` as follows:

```
<path id="test.classpath">
  <pathelement location="${build.test.dir}"/> <!-- build/test -->
  <pathelement location="C:\JAVA\junit4.8.2\junit-4.8.2.jar"/>
  <pathelement location="${build.classes.dir}"/>
                    <!-- build/classes: here is the required class! -->
</path>

<target name="test-compile" depends="compile,test-init">
   <javac includeAntRuntime="false"
          srcdir="${src.test.dir}"
          destdir="${build.test.dir}"
          classpathref="test.classpath">
   </javac>
</target>
```

New path element

This target `test-compile` is also required. (Which else ?)

Compiling from `ch04\test` to `ch04\build\test`

Add `classpathref`

Adding `classpathref="test.classpath"` is required for compiling `FilePersistenceServicesTest.java` because the latter actually refers to `FilePersistenceServices.class` in another dir. `build/classes`.

RUN again `ant -f mybuild.xml clean junit-TestRunner` and try to understand the output (ignoring some inessential parts).

Invoking `TestRunner` from build file with `<java>`

After above changes, RUN it again:

```
C:\Antbook\ch04>ant -f mybuild.xml clean junit-TestRunner > output.txt
Buildfile: C:\Antbook\ch04\mybuild.xml
     [echo] Building Testing Examples
clean:
   [delete] Deleting directory C:\Antbook\ch04\build
init:
    [mkdir] Created dir: C:\Antbook\ch04\build\classes
compile:
    [javac] Compiling 1 source file to C:\Antbook\ch04\build\classes
test-init:
    [mkdir] Created dir: C:\An
test-compile:
    [javac] Compiling 2 source files to C:\Antbook\ch04\build\test

junit-TestRunner:
     [java] JUnit version 4.8.2
     [java] ..E.E...
     [java] Time: 0.037
     [java] There were 2 failures:
     [java] 1) testWrite(org.eclipseguid
FilePersistenceServicesTest)
     [java] java.lang.AssertionError: NOT WRITTEN???
```

Sending output into file `output.txt` if it is too long.

1 Java file `FilePersistenceServices.java` in `src`

There is currently 2 test files `SimpleTest.java` and `FilePersistenceServicesTest.java` in `ch04\test` *compiled* to `ch04\build\test`

Please ANSWER: Which "." corresponds to which test method and in which test case? What each "E" means?

continued

Invoking `TestRunner` from build file with `<java>`

CONTINUATION

```
<MANY LINES OMITTED>

     [java] 2) testRead(org.eclipseguide.persistence.
FilePersistenceServicesTest)
     [java] java.lang.AssertionError: expected:<[One, Two, Three]> but
was:<null>

<MANY LINES OMITTED>

     [java] FAILURES!!!
     [java] Tests run: 6,  Failures: 2
     [java]
     [java] Java Result: 1

BUILD SUCCESSFUL
Total time: 2 seconds
```

ignore

Compare this output with that in Slide 38 of part 5. Eclipse and Junit.

Is there any difference? Why?

Thus, `FilePersistenceServicesTest.testWrite` and `testRead` failed. But formally "BUILD SUCCESSFUL".

One of the reasons why `<java>` task for running tests is *not so good*.