More On JUnit

- Guidelines for Testable Designs
- Testing Patterns
- JUnit and Ant
- Code Coverage Tools

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Guidelines for Testable Designs

- Koskela recommends the following strategies for writing code that's amenable to unit testing:
  - Favor composition/delegation over inheritance
  - Avoid statics and Singletons
  - Isolate and inject dependencies



These guidelines are from Lasse Koskela's book Test Driven, Manning 2008.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Guidelines for Good Tests

- Kent Beck describes the following attributes of good unit tests:
    - Run fast (they have short setups, run times, and break downs)
    - Run in isolation (you should be able to reorder them)
    - Use data that makes them easy to read and to understand
    - Use real data (e.g. copies of production data) when they need to

This list is adapted from: http://www.agiledata.org/essays/tdd.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unit Testing Patterns

- Koskela defines the following types of unit-testing patterns:
  - Assertion patterns
  - Patterns for test fixtures
  - Miscellaneous patterns

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Assertion Patterns

- These patterns describe common ways to use assertions in test cases:
    - Resulting state assertion
    - Guard assertion
    - Delta assertion

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Resulting State Assertion

- This is the most common type of assertion
- A test case exercises functionality, then asserts expected state of the test fixture aftwards

```
1    @Test
2    public void testGetGpa()
3    {
4      GasPurchase gp = new GasPurchase();
5      gp.setGallons(12);
6      gp.setMilesDriven(360);
7      gp.setPrice(3.87);
8      assertEquals("GPA should be 30.0", 30.0,
9         gp.getGpa());
10   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Guard Assertion

- In this type of assertion pattern, the test case checks the test fixture **before** exercising the functionality to assure that state assumptions are valid

```
1    @Test
2    public void testAddPurchase()
3    {
4      PurchaseList pl = new PurchaseList();
5      assertNull("Purchase list should be empty",
6         pl.getPurchases());
7      GasPurchase gp = new GasPurchase();
8      gp.setGallons(12);
9      gp.setMilesDriven(360);
10     gp.setPrice(3.87);
11     pl.addPurchase(gp);
12     assertNotNull("Purchase list should be non-empty",
13       pl.getPurchases());
14   }
```

Note in this case, we also use the Resulting State Assertion pattern to check the fixture's state after adding the purchase to the list.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Delta Assertion

- In this type of assertion pattern, the test case checks the test fixtures **change** in state after exercising functionality

```
1    private PurchaseList pl;
2
3    // setup method omitted
4
5    @Test
6    public void testAddPurchase()
7    {
8      int sizeBefore = pl.getPurchaseCount();
9      GasPurchase gp = new GasPurchase();
10     gp.setGallons(12);
11     gp.setMilesDriven(360);
12     gp.setPrice(3.87);
13     pl.addPurchase(gp);
14     assertEquals("Size should increment",
15       (sizeBefore+1), pl.getPurchaseCount());
16   }
```

In this example, a setup method (not shown) creates a PurchaseList and adds some GasPurchases to the list.

Our Delta Assertion patterned test first retrieves the starting size of the purchase-list, adds another GasPurchase, then asserts the assumption that the size of the list should increase by one.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Fixture Patterns

- These patterns describe common ways to work with the test fixture:
  - Parameterized creation method
  - Object mother
  - Automated teardown

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Parameterized Creation Method

- This pattern extracts object creation and initialization into a separate method to hide the details
- The pattern lets us hide the initialization of attributes that aren't important for the test case

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Parameterized Creation Method, cont'd

```
1    @Before
2    public void testAddPurchase()
3    {
4      GasPurchase gp1 = createPurchase(12.0, 360);
5      GasPurchase gp2 = createPurchase(13.0, 400);
6      . . .
7    }
8
9    private GasPurchase createPurchase(double gallons,
10      double milesDriven)
11   {
12     GasPurchase gp = new GasPurchase();
13     gp.setGallons(gallons);
14     gp.setMilesDriven(milesDriven);
15     gp.setPrice(0.00);
16     return gp;
17   }
```

In this case, we don't care about the GasPurchase price, since our test only checks the MPG calculation.

This is a form of factory method, which is a Gang of Four design pattern.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Object Mother

- In this pattern, you delegate creation and initialization to a separate object that hides complexity

```
1    @Before
2    public void setUp()
3    {
4      PurchaseListFactory mother =
5        new PurchaseListFactory();
6      PurchaseList pl =
7        mother.getPurchaseList();
8    }
```

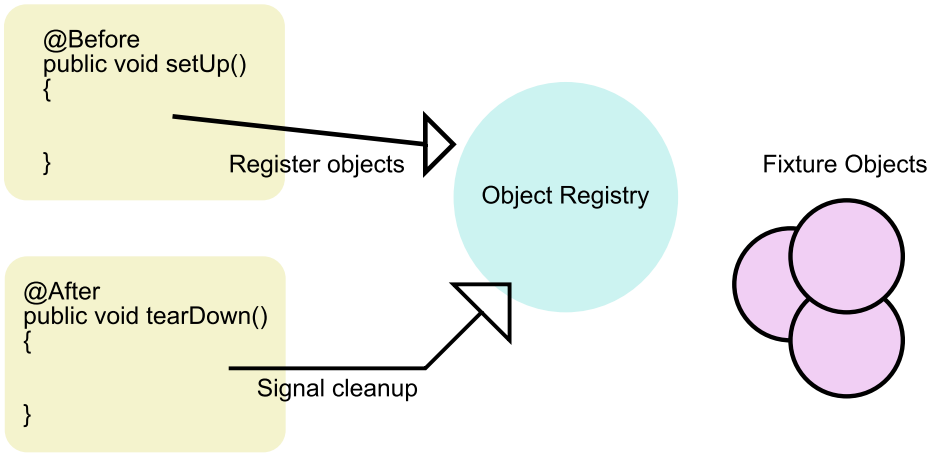
This pattern is a logical extension of the Parameterized Creation Method pattern, which delegates the entire creation of all or part of the test fixture to a separate class.

The downside to this pattern is that you have to write the "mother" class, which adds to the burden of unit testing.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Automated Teardown

- In this pattern, you delegate clean up tasks to a separate class that maintains an "object registry" and knows the details of cleaning up after the test



If your test fixture has non-trivial clean-up requirements, your test case can delegate the clean up to a separate class (this is logically the inverse of Object Mother).

One common way to write the cleanup class is to have it implement a registry of objects that need some cleanup. In your "setUp" method, as you create the test fixture, add such objects to the registry, and then in your test case's "tearDown" method, signal the registry to do its work.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Automated Teardown, cont'd

```
1    private PurchaseObjectRegistry registry =
2        new PurchaseObjectRegistry();
3
4    @Before
5    public void setUp()
6    {
7      GasPurchase gp1 = new GasPurchase();
8      . . .
9      registry.addObject(gp1);
10
11     GasPurchase gpN = new GasPurchase();
12     . . .
13     registry.addObject(gpN);
14   }
15
16   @After
17   public void tearDown()
18   {
19     registry.cleanUp();
20   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Apache Ant

- Apache Ant is a Java-based scripting tool that you can use to automate many common Java tasks, including running JUnit testing
- To use Ant, write an XML file that contains **targets** for jobs that you want the script to accomplish
- Ant provides numerous **core tasks** to let you perform operations; in addition, Ant supports extension or optional tasks



Ant is included with popular integrated develompment environments like Eclipse.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Ant Script

```
1    <?xml version="1.0"?>
2    <project default="deploy">
3       <property name="project-name" value="lab11"/>
4
5       <target name="makewar">
6         <delete file="${project-name}.war"
7                failonerror="false"/>
8          <war warfile="${project-name}.war"
9               webxml="web.xml">
10            <fileset dir="." includes="*.jsp"/>
11            <classes dir="." includes="**/*.class"/>
12         </war>
13      </target>
14
15      <target name="deploy" depends="makewar">
16         <copy file="${project-name}.war"
17               todir="/appserver/deploy"/>
18      </target>
19   </project>
```

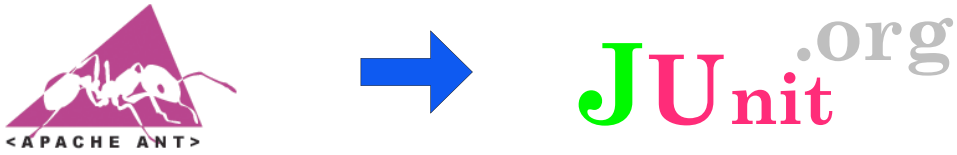
C:> ant -f build.xml

The common filename for Ant scripts is build.xml.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JUnit and Apache Ant

- The JUnit Ant task lets you run JUnit tests from an Ant script
- The JUnit Ant task provides many different ways to produce output including plain text and XML
- To use the JUnit Ant task, you must configure the JUnit JAR file's location in the Ant script



The JUnit Ant task is not a core task that's shipped with Ant itself. Instead, it's a custom task whose code is located in the junit.jar.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample JUnit Ant Script

```
<?xml version="1.0" ?>
<project name="myproject" default="output">
  <property name="junit4-jar"
    value="/junit4_4.3.1/junit.jar"/>

  <target name="output">
   <junit printsummary="false">

     <formatter type="brief" usefile="false"/>

      <classpath path="bin" />
      <classpath location="${junit4-jar}"/>

     <test name="purchase.TestGasPurchase" />

   </junit>
  </target>
</project>
```

output to console

where to find class under test

test-case classname

```
[junit] Testsuite: purchase.TestGasPurchase
[junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.063 sec
[junit] Testcase: testGetGpa(purchase.TestGasPurchase): FAILED
[junit] GPA should be 30.0 expected:<30.0> but was:<31.0>
```


This Ant script uses the junit task to generate "brief" formatted output to the console. The "usefile" attribute setting tells the task whether or not to send the output to a file; the default is true.

The "printsummary" attribute tells the junit task not to itself display any output -- if you set that option to true, you will get duplicate output, since the formatter also produces output.

Note that we need to specify locations of the class under test and the JUnit JAR. We also need to specify the name of the test-case class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The junitreport Ant Task

- To directly generate HTML from XML-formatted Ant output, you can use the **junitreport** task

```
1    <target name="output">
2        <junit printsummary="false">
3            <formatter type="xml"/>
4            <classpath path="bin" />
5            <classpath location="${junit4-jar}"/>
6            <test name="purchase.TestGasPurchase" />
7        </junit>
8    </target>
9
10   <target name="report" depends="output">
11       <junitreport todir=".">
12           <fileset dir=".">
13               <include name="TEST-*.xml"/>
14           </fileset>
15           <report format="frames" todir="."/>
16       </junitreport>
17   </target>
```

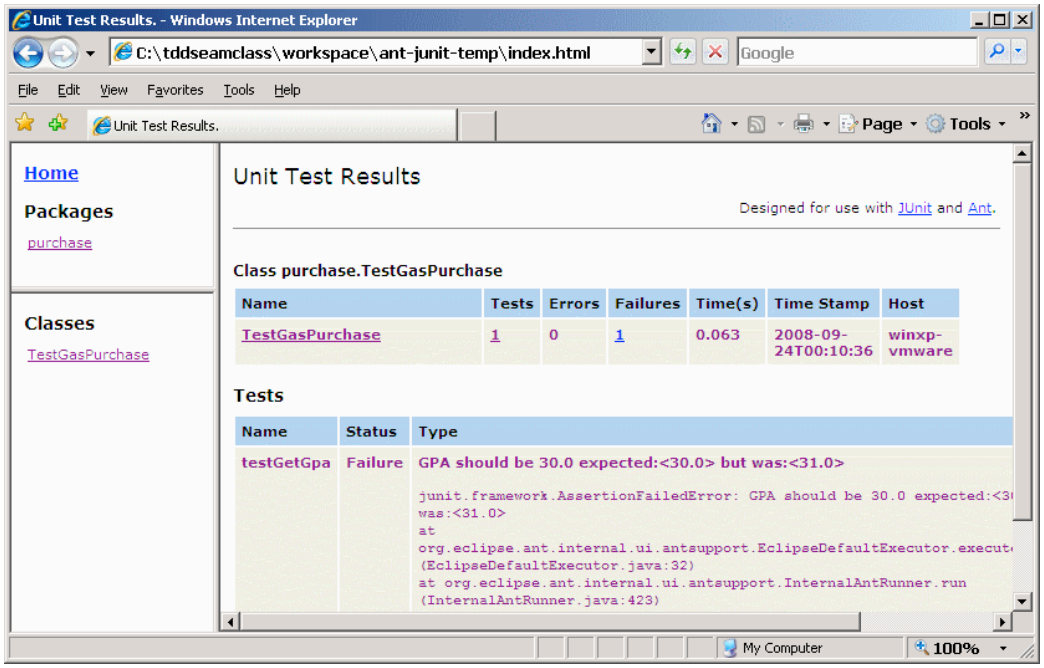

Here we show a fragment of an Ant script. This script is similar to the previous example, but in line 3, generates XML output to a file rather than "brief" output to the console. Note that it's possible to use more than one "format" element, so we could get both XML files and the console output we did in the last script.

We then use the junitreport task to transform the XML file(s) generated in the "output" target into HTML. In line 15 we specify that the HTML should use frames to make it easier to navigate.

Note that since we didn't specify anything about the names of the XML (or HTML) files, the script will generate files with default names. For the XML, the default names have the form of TEST-yyyyy.xml.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The junitreport Ant Task, cont'd



This screenshot shows what the HTML report looks like.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

EMMA: Code Coverage

- EMMA is a code-coverage tool that **instruments** your code at the byte-code level to determine if all of your lines of code have been exercised by your test cases
- Emma is available from

http://emma.sourceforge.net/



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Jester: Code Coverage

- Jester is a code-coverage tool that produces a modified version of your source and then runs test cases to see if the change causes a failure (it should)
- Jester reports the code that **doesn't fail** as a possible missed coverage
- Jester is available from http://jester.sourceforge.net/
- There is a good article about running Jester by Elliotte Rusty Harold at http://www-128.ibm.com/developerworks/library/j-jester/



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned

- About common patterns used in writing tests
- How to integrate JUnit and Apache Ant
- About code-coverage tools

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum