Automation, Quality of testing

Pieter van den Hombergh,Ferd van Odenhoven

14 maart 2006

1. Contents this week
2. Simple build and test automation
3. build tools
   - Ant
   - Build file
4. Quality of testing
   - The flow of code
   - Coverage frameworks
   - Testing fairy tale: Hansel
   - Demonstration
     - Coverage test suite

Using build tools and scripts

In a software development process where testing is considered cruicial, the following steps are typical to build and test the software:

1. Create a fresh (empty) directory. (Or delete the contents of an old one).
2. Checkout the version you want to build from the version control system
3. Build the system using the/a build script (which also comes from the repository).
4. Test the built components and assemblies (unit testing, integration testing)
5. If all tests pass, package the system and TAG it with an appropriate TAG.
6. If any tests fails, bug the developers, e.g. by mailing them

Simple build tools

Well known tools are

Make

- Make (gnu make, microsoft nmake). A `Makefile` with a peculiar but simple syntax is used.
  - IDE either use then under the water level or is at least able to produce them.
  - Generated Makefiles are difficult to understand. The IDE's typically throw in every possible option, without really needing them all.

The power of an ant

Simple build tools (2)

Well known tools are

(n)Ant

- Ant is the **make** replacement in the java world. Open (apache.org) source. and available on any platform that supports java (and the java XML libs).
- To be found at `http://ant.apache.org`

Simple build tools (2)

- Ant defines so called tasks. You can define your own, but there is a great ant community with a lot of predefined tasks. The only problem is finding them (try google) and getting the documentation. (A weak spot in open source).
- Ant build scripts are written in XML
- Ant has access to the java VM.
- Ant extensions can be defined in java.
- Typical ant tasks are build and unit testing. But much more can be done.
- There is a .NET counterpart nowadays. It is called, guess what, Nant.

Ant build script build.xml, excerpts properties

Extract from build.xml (for the shoppingcart example).

```
<property name="appname"
  description="name of the application and jar file"
  value="shoppingcart"
/>
<!-- customize for the fully qualified
     Classname with THE main() method -->
<property name="appmain"
  description="class with the main method to start the application"
  value="shop.Main"
/>

<!-- Location of the system jar files.
   Set for /usr/share/java
   which is the default in
   SuSE linux variants and maybe more. -->
<property name="javalib"
  description="Location of the jar libs."
  location="/usr/share/java"
/>
```

Compile target

```
<target name="compile" depends="initbuild"
  description="Compile the sources without tests">
  <javac srcdir="${src}"
       debug="false"
       destdir="${build}" />
</target>

<target name="compiletests" depends="compile,initbuild"
    description="Compiles the sources with tests">
  <javac srcdir="test"
       debug="false"
       destdir="${testbuild}">
    <classpath refid="project.classpath"/>
  </javac>
</target>
```

Ant test target

```
<target name="unittest"
    depends="compiletests"
    description="Runs the tests">
  <junit haltonfailure="true">
    <classpath
        refid="project.classpath"/>
    <formatter type="brief"
        usefile="false"/>
    <batchtest>
      <fileset dir="${testbuild}"
          includes="**/*Test.class"/>
    </batchtest>
  </junit>
</target>
```

Ant install file

Yes I know it is a simple jar file. But this is java. Other people call it **xcopy** deploy and think it is fancy.

```
<target name="jar"
  depends="compile"
  description="Compile the sources and makes an executable j
  <mkdir dir="${jar}" />
  <jar jarfile="${jarfile}" basedir="${build}">
    <manifest>
      <attribute name="Main-Class"
        value="${appmain}"/>
    </manifest>
  </jar>
  <echo message="appmain=${appmain}, appname=${appname}"/>
  <echo message="run application with command on next line"/
  <echo message="java -jar ${jarfile}"/>
</target>
```

There is little ant cannot do

```
<target name="publish"
  description="Bundle the whole stuff and publish to
  depends="jar,javadoc">
  <copy todir="${publish.dir}/api">
    <fileset dir="api/"/>
  </copy>
  <copy todir="${publish.dir}">
    <fileset file="jar/*.jar"/>
  </copy>
  <zip destfile="${publish.dir}/${zip.filename}">
    <fileset dir="../${appname}"
        excludes="**/CVS,${zip.excludes}"/>
  </zip>
</target>
```

Using build properties

- The build.xml file can define placeholders with default values.
- These properties can be override/redefined in a build.properties file.
- This allows the use of a generic build.xml file and reuse it in various projects.
- With `ant -p` or `ant -projecthelp` you get a list of possible ant targets.

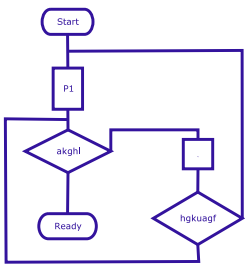
Example of build.properties

```
# specific for shoppingcart job
appmain=shop.Shop
appname=shop
publish.dir=/home/www/public_html/sen1/shop
zip.filename=shop.zip
zip.excludes=**/*.class Solution.java
```

Quality of your code

- You now know how to determine the quality of your code
- how about the quality of your test
  - Do you test all that is testable?
  - Do you take into account al possible combinations of preconditions and the ensuing postconditions?
  - And of course is the amount of testing within an acceptable range

Code flow



- Did you test all of this or only the 'correct' or logical path?
- Or
- Are all possible paths in the code executed?

Test coverage

- Test coverage is the degree in which you covered the code in the class under test, i.e. exercised all possible paths.
- Full coverage (100%) is not always possible, because of some wierd combination of paths in the code and combination of preconditions to reach those paths.
- For instance Exceptions that are hard to trigger.
- Nonetheless you should try.
- There are various open source coverage aids:
  - Quilt
  - EMMA
  - NoUnit
  - InsECT
  - Hansel
  - Jester
  - etc

see
`http://java-source.net/open-source/code-coverage`

This is not a fairy tail

Testing the tests with Hansel

For the combination of `java` and `ant` there are several frameworks that allow you to test your tests.

- Hansel (of Hansel and Gretel) is such a framework (at least one that is simple to use and I used succesfully).
- Works on the byte code level, i.e. insert so called instrumentation into the class objects during class loading.
- Determines from the test class and the class under test if all execution paths in the clas under test are used duering one or more tests.
- A hansel test fails if there are non-executed paths.
- `http://hansel.sourceforge.net/`

Demonstration time

Ingredients:

- Hansel.jar on your classpath. From `http://hansel.sourceforge.net/`. I used version 1.02.
- The Byte Code Engineering Library in bcel.jar. Version 5.1. Standard installed on our SuSE linux boxes.
- The appropriate target in ant.
- Or a XXTestCoverage.java test suite class that defines what has to be tested.

Coverage suite

```
package shop;
import junit.framework.Test;
import junit.framework.TestSuite;
import org.hansel.CoverageDecorator;
/**
 * ShoppingTestCoverage.java.
 * Coverage suite for shop package.
 * @author Pieter van den Hombergh (hom)
 */
public class ShoppingTestCoverage extends TestSuite {
    public static Test suite(){
        TestSuite suiteWithCoverage = new TestSuite();
        CoverageDecorator coverageTest = new CoverageDecorator(
                new Class[]{ItemImpl.class,
                        ShoppingCartImpl.class,
                        FreeBubbleGum.class});
        coverageTest.addTestClass( ItemImplTest.class );
        coverageTest.addTestClass( ShoppingCartImplTest.class );
        coverageTest.addTestClass( FreeBubbleGumTest.class );
        suiteWithCoverage.addTest( coverageTest );
        return suiteWithCoverage;
    }
}
```