OBJECTFAB
simple is beautiful.

Testing without excuses

Diligent but routine pieces of work for JUnit should be the computer's task.

By Andreas Braig and Steffen Gemkow

„Why wasn't this tested before?" This question often arises shortly before a deadline. The answer: because testing was not easy enough. Before programmers get infected with testing using JUnit, the additional effort is sometimes used as an excuse. JUnitDoclet is the tool to take over the burden of creating and maintaining unit tests. Suddenly all the excuses don't count any longer.

Unit tests and test-driven development increase the reliability of software and the programmers confidence in her own code significantly. Kent Beck and Erich Gamma created the quasi-standard for automated unit tests in Java, JUnit [1]. On the one hand the implementation using JUnit is really simple, on the other hand starting is still laborious. Prior to writing the actual test, the class skeleton of the *TestCase* has to be hammered in. Each time you are adding or changing methods the appropriate test methods need to be adapted. These chores should not be used any longer as an excuse against unit tests. That's why the authors developed JUnitDoclet [2].

Mode of operation

JUnitDoclet works as a plug-in for SUN's JavaDoc tool but instead of generating HTML documentation it generates test classes and methods. For every public method of every class in the application a JUnit TestCase is created. In addition all TestCases of a package and its subpackages are added to a JUnit TestSuite. Compared to similar functions and extensions of development environments JUnitDoclet has a couple of advantages:

- Independence of IDEs and capability for automation,
- Generation of test classes and methods for complete package hierarchies (including TestSuites),
- Incremental generation through merging with existing tests and updating after simple refactorings,
- Adaptability according to users needs through templates,
- Availability of the source code.

Usage

We will take an example from the programming contest „The Way Out" [3] to demonstrate working with JUnitDoclet. The class *SpaceShip* in listing 1 represents a spaceship in a discrete universe. Position, speed, direction and energy usage can take integer values only. The method *calculateStep* calculates the speed and direction of the next movement while the method *makeStep* executes it, i.e. calculates the next position. Taking this simple and incomplete implemented version, we will generate the TestCase with the following command:

```
javadoc -doclet com.objectfab.tools.junitdoclet.JUnitDoclet -sourcepath .\src -d .\junit de.thewayout.two.SpaceShip
```

The result is shown in listing 3 where a test method was generated for every public method while only the lines printed in italics are implemented manually. Additionally an instance of the class to be tested is created in the method *setUp*, which is then available in every method. For the accessor methods of the attributes *speed*, *direction* and *position* combined test methods are generated containing a type-dependent implementation suggestion. The method *testVault* being generated for every test class will be mentioned later in the article.

Now JUnitDoclet has done the chore and the developer may start implementing unit tests. The numerous comments like *"// JUnitDoclet begin ... "* und *"// JUnitDoclet end ... "* mark protected areas which JUnitDoclet won't overwrite during future invocations. Own source code like the lines printed in italics in listing 3 has to be added within these areas.

```
package de.thewayout.two;

public class SpaceShip
{
  private int direction = 0;
  private long speed = 0, power = 0;
  private long[] position = new long[] {0, 0, 0};









  public int getDirection() { return direction; }
  public void setDirection(int newDirection) {
    direction = newDirection;
  }

  public long getSpeed() { return speed; }
  public void setSpeed(long newSpeed) {
    speed = newSpeed;
  }

  public long[] getPosition() {return position; }
  public void setPosition(long[] newPosition) {
    position[0] = newPosition[0];
    position[1] = newPosition[1];
    position[2] = newPosition[2];
  }

  public long getPowerConsumed() {return power; }
  public void consumePower(long value) {

    power += value;

  }

  public int calculateStep() {
    // Berechne Geschwindigkeit und Richtung
    // für den nächsten Schritt.
  }

  public void makeStep() {
    // Rufe calculateStep() auf und
    // bewege SpaceShip für jeden Punkt
    // Geschwindigkeit ein Feld weiter.
  }
}
```

Listing 1: First concept of the class SpaceShip

```
package de.thewayout.two;

public class SpaceShip
{
  private int direction = 0;
  private long speed = 0, power = 0;
  private long[] position = new long[] {0, 0, 0};
  private Steering steering = null;

  public Steering getSteering(){return steering;}
  public void setSteering(Steering theSteering){
    steering = theSteering;
  }

  public int getDirection() { return direction; }
  public void setDirection(int newDirection) {
    direction = newDirection;
  }

  public long getSpeed() { return speed; }
  public void setSpeed(long newSpeed) {
    speed = newSpeed;
  }

  public long[] getPosition() {return position; }
  public void setPosition(long[] newPosition) {
    position[0] = newPosition[0];
    position[1] = newPosition[1];
    position[2] = newPosition[2];
  }

  public long getPowerConsumed() {return power; }
  public void consumePower(long value,
                           String cause) {
    power += value;
    System.out.println(power + " : " + cause);
  }









  public void makeStep() {
    // Rufe calculateStep() auf und
    // bewege SpaceShip für jeden Punkt
    // Geschwindigkeit ein Feld weiter.
  }
}
```

Listing 2: Updated version of the class SpaceShip

Refactoring

Everything changes, that holds true for software as well. During development countless small and large refactorings that may affect tests are done to classes of any application. Taking a close look on the affected classes all refactorings can be reduced to the following changes:

- Adding methods:
  In this trivial case a new test method is created.
- Deleting and renaming methods:
  The corresponding test method is not generated during the next run of JUnitDoclet. The test implementation inside "// JUnitDoclet begin ... " and "// JUnitDoclet end ... " is moved to the method *testVault* and thus saved from being lost. In case the method was renamed a new test method is generated. The implementation can be moved manually from the method *testVault* to this place.
- Changing signatures and overloading methods:
  In this case JUnitDoclet doesn't change anything because methods are identified be their names only.

```
package de.thewayout.two;
import junit.framework.TestCase;
// JUnitDoclet begin import
// JUnitDoclet end import

public class SpaceShipTest
// JUnitDoclet begin extends_implements
    extends TestCase
// JUnitDoclet end extends_implements
{
  // JUnitDoclet begin class
  SpaceShip spaceship = null;
  // JUnitDoclet end class

  public SpaceShipTest(String name) {
  // JUnitDoclet begin method SpaceShipTest
    super(name);
  // JUnitDoclet end method SpaceShipTest
  }
  public SpaceShip createInstance() {
  // JUnitDoclet begin method testcase.createInstance
    return new SpaceShip();
  // JUnitDoclet end method testcase.createInstance
  }
  protected void setUp() throws Exception {
  // JUnitDoclet begin method testcase.setUp
    super.setUp();
    spaceship = createInstance();
  // JUnitDoclet end method testcase.setUp
  }
  protected void tearDown() throws Exception {
  // JUnitDoclet begin method testcase.tearDown
    spaceship = null;
    super.tearDown();
  // JUnitDoclet end method testcase.tearDown
  }
  public void testSetGetDirection() throws Exception {
  // JUnitDoclet begin method setDirection getDirection
    int[] tests = {Integer.MIN_VALUE, -1, 0, 1,
        Integer.MAX_VALUE};
    for (int i=0; i<tests.length; i++) {
      spaceship.setDirection(tests[i]);
      assertEquals(tests[i],spaceship.getDirection());
    }
  // JUnitDoclet end method setDirection getDirection
  }
  public void testSetGetSpeed() throws Exception {
  // JUnitDoclet begin method setSpeed getSpeed
    long[] tests = {Long.MIN_VALUE, -1, 0, 1,
        Long.MAX_VALUE};
    for (int i=0; i<tests.length; i++) {
      spaceship.setDirection(tests[i]);
      assertEquals(tests[i],spaceship.getDirection());
    }
  // JUnitDoclet end method setSpeed getSpeed
  }
```

```
  public void testSetGetPosition() throws Exception {
  // JUnitDoclet begin method setPosition getPosition
    long[][] tests = {null, new long[]{}};
    for (int i=0; i<tests.length; i++) {
      spaceship.setPosition(tests[i]);
      assertEquals(tests[i], spaceship.getPosition());
    }
  // JUnitDoclet end method setPosition getPosition
  }

  public void testGetPowerConsumed() throws Exception {
  // JUnitDoclet begin method getPowerConsumed
  // JUnitDoclet end method getPowerConsumed
  }

  public void testConsumePower() throws Exception {
  // JUnitDoclet begin method consumePower
    long powerBefore = spaceship.getPowerConsumed();
    spaceship.consumePower(100);
    assertEquals("Unexpected power consumption",
       powerBefore + 100, spaceship.getPowerConsumed());
  // JUnitDoclet end method consumePower
  }

  public void testCalculateStep() throws Exception {
  // JUnitDoclet begin method calculateStep
    long[] p1, p2;
    p1= new long[] { 100, 100, 100 };
    spaceship.setPosition(p1);
    spaceship.setSpeed(1);
    spaceship.calculateStep();
    spaceship.goAhead();
    p2 = spaceship.getPosition();
    assertEquals("Moved unexpected distance.", 1,
        Math.abs(p2[0] - p1[0] +
                 p2[1] - p1[1] +
                 p2[2] - p1[2]));
  // JUnitDoclet end method calculateStep
  }

  public void testMakeStep() throws Exception {
  // JUnitDoclet begin method makeStep
  // JUnitDoclet end method makeStep
  }

  public void testVault() throws Exception {
  // JUnitDoclet begin method testcase.testVault
  // JUnitDoclet end method testcase.testVault
  }

  public static void main(String[] args) {
  // JUnitDoclet begin method testcase.main
    junit.textui.TestRunner.run(SpaceShipTest.class);
  // JUnitDoclet end method testcase.main
  }
}
```

Listing 3: Generated test class SpaceShiptTest

In the example the method *calculateStep* is moved to a new class *Steering*. This is done by deleting the method in the class *SpaceShip* and adding a new attribute of the type *Steering* including accessor methods. Next, the signature of the method *consumePower* is changed by adding a parameter, which states the reason for the energy consumption (e.g. propulsion system, life support or energy shields). Listing 2 shows the new version of the class *SpaceShip* and listing 4 the changes within the test classes after running JUnitDoclet. Aside from adding methods the test classes won't compile, because the implementations expect the original version of the applications class. Comfortable development environments are able to rename methods and change method signatures throughout the whole source code. Otherwise a careful and global find and replace is required. The important fact is that implemented tests won't get lost. Without tight integration into an IDE, the most feasible solution is moving the test implementation to the method *testVault* and copying it manually to the correct test method afterwards. JUnitDoclet behaves defensive and prints out warnings whenever a test implementation is moved to the method *testVault*. In case a warning is overlooked and the test class is compileable despite the changes, the tests are executed and may fail. (Because of the new marker style in JUnitDoclet 1.0 tools like IntelliJ IDEA can refactor the markers when renaming methods too. The test code will be placed at the correct position right away, instead of temporary moving it to the method *testVault*.)

Integration

To unfold its full power JUnitDoclet needs to integrate easily and smoothly with the developers other tools. Using it in a script for the popular build tool ANT [4] is shown in listing 5. Calling the target *junitdoclet* all test classes are generated, with *junitcompile* compiled and with *junittest* executed. JUnitDoclet is called here for whole package hierarchies. Thereby it creates *TestSuites* for packages combining all contained *TestCases* and *TestSuites* of subpackages. This assures that global test runs execute all *TestCases*. Otherwise an exisiting test, which was accidentally not included in a *TestSuite*, could easily make someone think everything is alright. Of course the same result can be achieved without ANT using a batch file.

Special care is required in combination with version control systems. JUnitDoclet always creates all *TestCases* and *TestSuites*. Possibly another developer has already implemented and checked in some *TestCases* colliding with locally created versions. The authors recommend checking in new classes and packages always together with their appropriate *TestCases* and *TestSuites*, respectively.

Customizing

In every project there are different notions how unit tests should be implemented. This ranges from source code conventions over different versions of JUnit to own extensions of the JUnit framework. Therefore JUnitDoclet offers a template-based mechanism to adapt the code being generated to your own needs. For special requirements the source code is available.

```
package de.thewayout.two;
import junit.framework.TestCase;
// JUnitDoclet begin import
// JUnitDoclet end import

public class SpaceShipTest {

// ... no changes ...

  public void testSetGetSteering() throws Exception {
  // JUnitDoclet begin method setSteering getSteering
    Steering[] tests = {null, new Steering()};
    for (int i=0; i<tests.length; i++) {
      spaceship.setSteering(tests[i]);
      assertEquals(tests[i], spaceship.getSteering());
    }
  // JUnitDoclet end method setSteering getSteering
  }

// ... no changes ...

  public void testConsumePower() throws Exception {
  // JUnitDoclet begin method consumePower
    long powerBefore = spaceship.getPowerConsumed();
    spaceship.consumePower(100);
    assertEquals("Unexpected power consumption",
        powerBefore + 100,
        spaceship.getPowerConsumed());
  // JUnitDoclet end method consumePower
  }

// ... no changes ...

  public void testVault() throws Exception {
  // JUnitDoclet begin method testcase.testVault
// JUnitDoclet begin method calculateStep
    long[] p1, p2;
    p1 = new long[] { 100, 100, 100 };
    spaceship.setPosition(p1);
    spaceship.setSpeed(1);
    spaceship.calculateStep();
    spaceship.goAhead();
    p2 = spaceship.getPosition();
    assertEquals("Moved unexpected distance.", 1,
        Math.abs(p2[0] - p1[0] +
                 p2[1] - p1[1] +
                 p2[2] - p1[2]));
// JUnitDoclet end method calculateStep
  // JUnitDoclet end method testcase.testVault
  }

// ... no changes ...

}
```

Listing 4: SpaceShiptTest after rerunning JUnitDoclet

Already existing *TestCases* can be used along with newly generated ones without any problems. JUnitDoclet does not overwrite files, which were not generated by itself and *TestSuites* contain protected areas where own *TestCases* may be added.

Summary

Using JUnitDoclet a lot of test methods can be generated quickly for existing and new projects. JUnit counts every test method as test case, but this number says nothing about the gained test coverage since most of the test cases are empty and can't fail. Only the manually implemented test cases may be taken into account to estimate test coverage. JUnitDoclet makes it easier to concentrate on the actual work – to implement test cases – but cannot take over this burden.

The first serious field test of JUnitDoclet was in the programming contest "The Way Out" where we provided it to all participants. As expected, experience shows that implementing unit tests is clearly faster with JUnitDoclet. The developer's flow of thoughts is not interrupted when switching between the application and unit tests, because he does not have to implement the test class bodies. Both leads to more fun when implementing unit tests und therefore higher test coverage and test quality.

```
<project name="junitdoclet-sample" default="all">
...
  <target name="junitdoclet" depends="compile">
    <javadoc
      packagenames = " de.thewayout.two.*"
      sourcepath = "./src"
      defaultexcludes = "yes"
      doclet =
      "com.objectfab.tools.junitdoclet.JUnitDoclet"
      docletpathref = "classpath_default"
      additionalparam = "-d ./junit -buildall">
      <classpath refid = "classpath_default" />
    </javadoc>
  </target>

  <target name="junitcompile" depends="junitdoclet">
    <javac srcdir="./junit" destdir="./classes">
      <classpath refid="classpath_default" />
    </javac>
  </target>

  <target name="junittest" depends="junitcompile">
    <junit printsummary="no" fork="yes"
       haltonfailure="no">
      <formatter type="plain" usefile="no"/>
      <test name=" de.thewayout.two.TwoSuite"/>
      <classpath refid="classpath_default" />
    </junit>
  </target>
...
</project>
```

Listing 5: Integration into an ant build script

To gain this experience yourself you can extend the class *SpaceShip* with methods to accelerate and brake and their appropriate test methods checking the energy consumption. The sample source code is available for downloading at [2].

About the Authors

Andreas Braig and Steffen Gemkow are working as consultants running their own Company: ObjectFab GmbH. Focussed on serverside Java and related technologies they provide support in many areas of software development, from developing to coaching, from reviews to methodology. Please visit the website [5] to learn more.

Links

[1] http://www.junit.org/

[2] http://www.junitdoclet.org/

[3] http://www.thewayout.de/

[4] http://jakarta.apache.org/ant/

[5] http://www.objectfab.de/