Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Test med JUnit 3

Denne artikel introducerer JUnit 3.

Den forklarer ideen med JUnit. Og den viser hvordan man konkret bruger det.

Den forudsætter kendskab til Java og noget generel udviklings erfaring.

Skrevet den **17. Feb 2010** af **arne_v** I kategorien **Programmering / Java** | ★★★☆☆

Historie:
V1.0 - 17/01/2004 - original
V1.1 - 31/01/2004 - forbedret formatering + lidt mere forklaring
V1.2 - 16/02/2010 - smårettelser

Versioner

Denne artikel omhandler JUnit 3.x. JUnit 3.x er tildels forældet idag.

For samme artikel omkring JUnit 4.x se:
http://www.eksperten.dk/guide/1347

Unit test

Unit test er ikke en ny opfindelse. Det har man brugt i mange år. Developere skulle teste deres lille stump inden hele projektet blev afleveret til test.

Men der er kommet rigtig meget fokus på den i de sidste 10-15 år. Det er ikke længere bare noget som man principielt burde huske at gøre men en meget vigtig del af hele udviklings processen.

Det hænger lidt sammen med begreber som XP (eXtreme Programming), Refactoring, Agile Software Development, Test Driven Development etc.. Men er også accepteret udenfor disse metodikker.

Man er beyndt at skrive unit test kode til at teste systematisk og automatisk.

Det med automatisk er uhyre vigtigt i. Hvis man har en stor suite af unit tests som tester en given kode, så kan man nemmere tillade sig at rette i den, fordi man kan teste store dele af funktionaliteten på få sekunder/minutter.

Samtidigt har unit test også flyttet sig tidsmæssigt.

Engang skrev man unit test kode til sidst efter at man havde skrevet

selve koden. Det er nu almindeligt accepteret, at det ikke er så godt. Folk skriver nemlig så unit test koden efter hvordan de ved koden virker d.v.s. at de finder færre fejl.

Så nu er det alment accepteret at man bør skrive unit test koden først udfra hvad koden bør kunne og ikke hvad den kan.

Nogen er endda begyndt at flytte det at skrive unit teste kode op som en del af design. Det er jo en præcis måde at beskrive hvad koden skal gøre og dermed dets interface udadtil.

JUnit

Unit test i Java er næsten synonymt med JUnit. Næsten alle bruger JUnit eller en overbygning på JUnit.

JUnit kan hentes her:
http://www.junit.org/

Og ZIP filen skal bare unzippes og så er man kørende.

De fleste Java IDE'er kommer idag med JUnit og indbyggede måder at køre JUNit på. Men det vil jeg dog ikke komme ind på her.

Eksempel

Lad os tage et simpelt eksempel for at illustrere hvordan det ser ud i praksis.

Vi har følgende kode som skal testes:

```
public class MathVector {
   private int[] v;
   public MathVector(int n) {
       v = new int[n];
   }
   public MathVector(int[] v) {
      this.v = v;
   }
   public MathVector add(int k) {
       int[] res = new int[v.length];
       for(int i = 0; i < v.length; i++) res[i] = v[i] + k;
       return new MathVector(res);
   }
   public MathVector sub(int k) {
       int[] res = new int[v.length];
       for(int i = 0; i < v.length; i++) res[i] = v[i] - k;
       return new MathVector(res);
   }
   public MathVector mul(int k) {
       int[] res = new int[v.length];
       for(int i = 0; i < v.length; i++) res[i] = v[i] * k;
       return new MathVector(res);
   }
```

```
    public MathVector div(int k) {
        int[] res = new int[v.length];
        for(int i = 0; i < v.length; i++) res[i] = v[i] / k;
        return new MathVector(res);
    }
    public MathVector mod(int k) {
        int[] res = new int[v.length];
        for(int i = 0; i < v.length; i++) res[i] = v[i] - (v[i]/k);
        return new MathVector(res);
    }
    public int size() {
        return v.length;
    }
    public int[] getV() {
        return v;
    }
}
```

(der er en simpel fejl !)

Vi laver nu følgende test kode:

```
import junit.framework.*;

public class TestMathVector extends TestCase {
    public TestMathVector(String s) {
        super(s);
    }
    protected void setUp() {
    }
    protected void tearDown() {
    }
    public void testAdd() {
        int[] v = { -1000000, -1, 0, 1, 1000000 };
        MathVector mv = new MathVector(v);
        MathVector mv2 = mv.add(777);
        assertEquals("add size", mv.size(), mv2.size());
        for(int i = 0; i < mv2.size(); i++) {
            assertEquals("add " + i, mv.getV()[i] + 777, mv2.getV()[i]);
        }
    }
    public void testSub() {
        int[] v = { -1000000, -1, 0, 1, 1000000 };
        MathVector mv = new MathVector(v);
        MathVector mv2 = mv.sub(777);
        assertEquals("sub size", mv.size(), mv2.size());
        for(int i = 0; i < mv2.size(); i++) {
            assertEquals("sub " + i, mv.getV()[i] - 777, mv2.getV()[i]);
        }
    }
    public void testMul() {
        int[] v = { -1000000, -1, 0, 1, 1000000 };
        MathVector mv = new MathVector(v);
```

```
        MathVector mv2 = mv.mul(777);
        assertEquals("mul size", mv.size(), mv2.size());
        for(int i = 0; i < mv2.size(); i++) {
            assertEquals("mul " + i, mv.getV()[i] * 777, mv2.getV()[i]);
        }
    }
    public void testDiv() {
        int[] v = { -1000000, -1, 0, 1, 1000000 };
        MathVector mv = new MathVector(v);
        MathVector mv2 = mv.div(777);
        assertEquals("div size", mv.size(), mv2.size());
        for(int i = 0; i < mv2.size(); i++) {
            assertEquals("div " + i, mv.getV()[i] / 777, mv2.getV()[i]);
        }
    }
    public void testMod() {
        int[] v = { -1000000, -1, 0, 1, 1000000 };
        MathVector mv = new MathVector(v);
        MathVector mv2 = mv.mod(777);
        assertEquals("mod size", mv.size(), mv2.size());
        for(int i = 0; i < mv2.size(); i++) {
            assertEquals("mod " + i, mv.getV()[i] % 777, mv2.getV()[i]);
        }
    }
}
```

Bemærk navne konventionen:
- klassen Xyz testet i klasse TestXyz
- metode abc testes i metoden testAbc

Jeg kalder assertEquals med 3 argumenter:
- tekst som identificerer stedet i koden
- forventet værdi
- faktisk værdi

(jeg anbefaler klart at man bruger versionen med 3 argumenter, da den identificerende tekst ofte er meget nyttig)

Der findes også assertTrue, assertNotNull etc. metoder. De kan slåes op i JavaDoc for JUnit.

Og når vi kører med:

```
javac -classpath . MathVector.java
javac -classpath .;/junit3.8.1/junit.jar TestMathVector.java
java -classpath .;/junit3.8.1/junit.jar junit.textui.TestRunner TestMathVector
```

Får vi følgende fejl:

```
.....F
Time: 0
There was 1 failure:
1) testMod(TestMathVector)junit.framework.AssertionFailedError: mod 0
```

```
expected:<-1> but was:<-998713>
        at TestMathVector.testMod(TestMathVector.java:53)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:25)

FAILURES!!!
Tests run: 5,  Failures: 1,  Errors: 0
```

Vi kigger lidt i koden og opdager at der mangler en multiplikation,
så vi retter koden til:

```
   public MathVector mod(int k) {
       int[] res = new int[v.length];
       for(int i = 0; i < v.length; i++) res[i] = v[i] - (v[i]/k)*k;
       return new MathVector(res);
   }
```

Og nu får vi:

```
.....
Time: 0

OK (5 tests)
```

Hvis man bruger ant, så ser build.xml ud som:

```
<project name="junitdemo" default="testrun">
    <property name="junitlib" value="/junit3.8.1/junit.jar"/>
    <target name="compile">
        <javac classpath="." srcdir="." destdir="."/>
    </target>
    <target name="testcompile" depends="compile">
        <javac classpath=".;${junitlib}" srcdir="." destdir="."/>
    </target>
    <target name="testrun" depends="testcompile">
        <junit fork="on">
            <classpath path=".;${junitlib}"/>
            <formatter type="plain" usefile="false"/>
            <test name="TestMathVector"/>
        </junit>
    </target>
</project>
```

og output ser ud som:

```
Buildfile: build.xml
```

```
compile:

testcompile:

testrun:
    [junit] Testsuite: TestMathVector
    [junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0.015 sec
    [junit] Testcase: testAdd took 0 sec
    [junit] Testcase: testSub took 0 sec
    [junit] Testcase: testMul took 0 sec
    [junit] Testcase: testDiv took 0 sec
    [junit] Testcase: testMod took 0 sec

BUILD SUCCESSFUL
Total time: 1 second
```

Det kan anbefales at bruge ant.

Vigtigt:
- junit.jar skal koieres til ant's lib directory
- fork="on"

Man kan så samle flere test cases i en test suite som:

```
import junit.framework.*;

public class AllTests extends TestCase {
    public AllTests(String name) {
        super(name);
    }
    public static Test suite() {
        TestSuite test = new TestSuite();
        test.addTestSuite(TestMathVector.class);
        test.addTestSuite(Abc.class);
        test.addTestSuite(Xyz.class);
        return test;
    }
}
```

Og teste den.

Man kan også erstatte junit.textui.TestRunner med junit.swingui.TestRunner
og få en lille grafisk vindue med grøn og rød op.

Det her var J2SE - man kan naturligvis også bruge JUnit til J2EE.
Remote interfaces til EJB's kan testes umiddelbart i JUnit. Det er
lidt vanskeligere med local interfaces og JCA connectorer, da de kun
kan kaldes inde fra. Til det formål er der lavet overbygninger til
JUnit bl.a. Apache Cactus:
http://jakarta.apache.org/cactus/index.html

God test lyst.

Kommentar af simonvalter d. 31. Jan 2004 | 1

Godt forklaret, jeg fik da fat i det :)

Kommentar af mikkelbm d. 25. May 2004 | 2

En god artikel, som fint beskriver brugen af JUnit.
Jeg kunne dog godt tænke mig lidt mere vægt på, at JUnit ikke er en mirakelkur. Forstået på den måde, at man virkelig skal gennemtænke enhver tænkelig situation, før en testcase er brugbar.