JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Code Generators

Ausarbeitung
(WS 2004)
(KV: Testen von Softwaresystemen- 339.404)

Bearbeiter
Klaus Almhofer, 9956531
Fischerlehner Christoph, 9956548
Fuchs Georg-Josef, 0056570

Betreuer
Dr. Christoph Steindl

Inhaltsverzeichnis

Einleitung

Auf den folgenden Seiten widmen wir uns sieben Tools, die alle die Entwicklung von Testfällen für Software erleichtern sollen. Alle diese Generatortools können als Open Source Software gratis bezogen werden und sind von der Homepage zu JUnit aus erreichbar (http://www.junit.org/news/extension/testcase_generation/index.htm).
Die Downloadadressen werden wir jeweils auch auf den folgenden Seiten direkt bei der Erläuterung der einzelnen Tools angeben.
Die Dateigröße der Werkzeuge ist sehr bescheiden, von einigen Kilobyte zu wenigen Megabyte, sodass sie auch über langsame Verbindungen heruntergeladen werden können.
Obwohl alle für dieselbe Programmiersprache nämlich Java konzipiert worden sind, wird dieser Ansatz in allen getesteten Tools anders und mit unterschiedlichem Aufwand umgesetzt.
Es ist weiters zu beachten, dass sich einige der TestGeneratoren noch in einem Entwicklungsprozess befinden, es sich auch teilweise um Betaversionen handelt, daher ist bei manchen Tools noch nicht das stimmige Gesamtkonzept, das angestrebt wird, bereits als ganzes schon umgesetzt. Einige Tools sind auch schon mit ausführbaren Batch-Dateien ausgestattet um den Einstieg komfortabler zu gestalten.
Die sieben Werkzeuge sind „Cricket Cage“, der „Jenerator“, der „JunitDoclet – Test Suite Generator“ , der „JUnit Test Case Builder“, der „Java Bean Tester“, der „WTTOOLS JUnit Test Generator“ und der „JUnit Test Generator“ von Webmind inc.
Gemein ist allen sieben Generatortools, dass sie ohne grafische Oberfläche auskommen und über die Shell ausgeführt bzw. in Integrierte Entwicklungsumgebungen eingebunden werden.
Der Einarbeitungs- bzw. Installationsaufwand für die einzelnen Produkte war aber mitunter sehr unterschiedlich, während manche Produkte relativ einfach sind und mit wenigen Schritten zum Laufen gebracht werden können, ist die Struktur von anderen hingegen unübersichtlich, bzw. verfügen sie über keine entsprechende Dokumentation.
Weiters ist auch die Kapselung der Tools ganz unterschiedlich, so sind manche alleine lauffähig während andere Datenbanken und konfigurierte XML-Dateien benötigen um lauffähig zu sein.
Leider waren unsere Versuche mit den einzelnen Werkzeuge Testcode zu erzeugen nicht bei allen der besprochenen Tools erfolgreich, zwei der Tools konnten nicht in Gang gebracht werden, gleichermaßen aufgrund der äußerst schlechten Dokumentation als auch den Vorraussetzungen, die für den Betrieb vonnöten sind.
Der Jenerator zum Beispiel ist ein Werkzeug das speziell für den Einsatz mit der J2EE konzipiert ist. Es soll mit J2EE vertrauten Praktikern helfen für ihre Geschäftsanwendungen Testfälle automatisch generieren zu lassen. Da das verlangte Detailwissen in unserem Testteam nicht vorhanden war und ein entsprechendes Tutorial zur J2EE etwa 500 Seiten hat, konnte der Jenerator keinem Praxistest unterzogen werden.
Nichtsdestotrotz soll auch das Konzept des Jenerators auf den folgenden Seiten nicht außen vor bleiben.
Generell gute Erfahrungen haben wir mit dem „JUnit test case Builder“ gemacht weil sich einerseits die Installation relativ einfach gestaltete sich andererseits die Einbindung in Eclipse als besonders praxisfreundlich herausstellte. Weiters ist das Tool im Gegensatz zu anderen auch nicht auf spezielle Programmstrukturen wie Java Beans eingeschränkt und kann daher für jede Art von Javaprogramm zur Erstellung von Testfällen angewendet werden.
Einige weitere Tools haben sich als ebenso brauchbar herausgestellt doch bleibt in diesen Fällen die umständlichere Bedienung über die Shell das Manko.
Alle Werkzeuge wurden auf Windows XP Rechnern getestet, ein Betrieb auf Linuxsystemen und anderen Betriebssystemen hätte aber sicherlich sehr ähnliche Ergebnisse gebracht, da aufgetretene Schwierigkeiten mit den Werkzeugen nicht auf das Betriebssystem zurückgeführt werden konnten.

Jenerator

Einleitung:

Das Werkzeug Jenerator kann nur im Kontext mit der J2EE verwendet werden. Bei diesem Tool handelt es sich nicht nur um einen ausschließlichen Generator für Testfälle, sondern es wird auch Code für ganz andere Anwendungsfälle erzeugt.
Weil es sich nicht um ein allgemein zu verwendendes Tool handelt wird im folgenden nur kurz auf die Möglichkeiten des Werkzeuges, ein längere Beschreibung würde den Rahmen sprengen.
Laut Dokumentation kann man damit die folgenden Arten von Code erzeugen:

Features:

Codeerzeugung für:
- Enterprise Java Beans (Remote, Local und Home Schnittstellen)
- Entity Beans (CMP und BMP)
- Value Object for Entity Beans
- Session Beans (zustandslos und zustandsbehaftet)
- Message Driven Beans
- EJB Deployment Descriptor
- Unit Tests für alle Arten von EJB für JUnit und Cactus
- ANT Build Dateien für den erzeugten Code
- Servlets

Jenerator ist also ein Generatortool für die Erzeugung von J2EE Code. Dabei kann man mehrere Datenquellen wie eine Datenbank oder eine XML-Datei verwenden, die zuerst in ein XML-Sheet umgewandelt werden worauf wiederum mehrere XSL Anweisungen durchgeführt werden. Das Ergebnis wird dann gemäß den Jenerator Desciptor Files in ausführbaren Code umgesetzt. Sozusagen als Nebenprodukt wird dabei für manche Codearten auch TestCode für JUnit bzw. Cactus erzeugt. Es ist also wichtig zu verstehen, dass Jenerator somit ungeeignet ist bestehenden Code in Testfälle umzusetzen.

Installation:

Da für den Betrieb von Jenerator eine umfangreiche Umgebung nötig ist(Datenbank etc.) ist das Tool nicht sehr einfach zu installieren. Nachdem die ZIP-Datei von der Herstellerseite heruntergeladen wurde müssen verschiedenste Umgebungsvariablen gesetzt werden JEN_HOME für das Stammverzeichnis des Werkzeuges und JDBC_DRIVER für die JDBC Treiberbibliothek. Danach muss in der Datei config.xml der Ausgabepfad vermerkt werden. Die folgenden Schritte sind je nach erzeugtem Code unterschiedlich und sollten daher im User Guide der sich im Unterverzeichnis docs\userguide befindet nachgelesen werden.

Verwendete Module:

Apache Xalan XSLT Umsetzer
Apache Xerces XML Parser
Apache Log4j Logging Framework

JUnit Doclet Test Suite Generator

Einleitung:

Für das Junit Doclet wurde ein interessanter Ansatz durch die Einbindung der Javadoc gewählt. Im Prinzip ist das JUnit Doclet ein Plugin des Javadoc Tools, doch anstelle einer HTML Dokumenationen werden eben Testfälle generiert. JavaDoc wird genutzt um die Struktur der Java Applikation zu extrahieren und daraus die Skeletons für Testfälle und Testsuiten zu generieren.
JunitDoclet steht derzeit in der Version 1.0.2 zum Download bereit. Diese Version ist zwar auch schon vom 30.11.2002, allerdings gab es davor bereits mehrere Updates die die meisten Bugs beseitigt haben dürften. Auch bei unseren Tests sind keine Probleme aufgetreten. Angeblich arbeiten die Entwickler derzeit intensiv an JunitDoclet 2.0, dessen Preview-Releasetermin (Mai 2004) allerdings schon verstrichen ist.

Features:

Testfälle werden für alle public Methoden generiert, außerdem können mit dem Junit Doclet auch Testsuiten erstellt werden in der sich dann die Testfälle für das gesamte Package befinden.
Ein besonders interessantes Feature ist die Unterstützung von **Refactoring** bei der durch Änderungen im Quelltext die Testfälle automatisch angepasst werden. Konkret werden folgende 3 Fälle unterstützt:
Nachträgliches Hinzufügen von Methoden:
In diesem einfachen Fall wird einfach ein neuer Testfall hinzugefügt
Löschen oder Umbenennen von Methoden:
Wird eine Methode gelöscht wird beim nächsten Ausführen des Tools der entsprechende Testfall nicht mehr generiert. Allerdings wird der manuell erstellte Code aus diesem Testfall nicht mitgelöscht sondern zur Sicherheit in die testVault() Methode verschoben damit dieser Code für den Programmierer nicht verloren ist. Beim Umbennen der Methode wird zusätzlich ein neuer Testfall für die umbennante Methode erstellt. Der Programmierer kann dann den beispielsweise den gesicherten Code aus *testVault()* in den umbenannten Testfall zurückkopieren.
Beispiel:
int calulateStep() wird in calculate FirstStep umbennant. Der Inhalt der händischen Codeimplementierung (alles zwischen "*// JunitDoclet begin Method*" und "*// JunitDoclet end method*") wird in *testVault()* gesichert. Der Testfall *testCalculateStep()* wird gelöscht und ein neuer Testfall *testCalculateFirstStep()* erzeugt. Der in *testVault()* gesicherte Code muß allerdings (falls gewünscht) händisch in die neue Testklasse kopiert werden.

Methode aus einem Java Programm:

```
public int calculateStep() {
    // do something important
}
```

Auszug aus der Testklasse die JunitDoclet zu dieser Methode erzeugt:

```
public void testCalculateStep() throws Exception {
// JUnitDoclet begin method calculateStep

   // Raum für den eigenen Testcode z.b:
   assertEquals(1,  calculateStep());

// JunitDoclet end method calculateStep
```

```
}

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

Methode aus Java Programm wird umbennant:

```
public int calculateFirstStep() {
    // do something important
}
```

Auszug aus der Testklasse die JunitDoclet nach der Unbennenung der Methode erzeugt:

```
public void testCalculateFirstStep() throws Exception {
// JUnitDoclet begin method calculateStep

// JunitDoclet end method calculateStep
}

public void testVault() throws Exception {
 // JunitDoclet begin method testcase.testVault
// JunitDoclet begin method calculateStep

   // Raum für den eigenen Testcode z.b:
   assertEquals(1,  calculateStep());

//JunitDoclet end method calculateStep
  // JunitDoclet end method testcase.testVailt
}
```

Ändern von Methodenparametern und Überladen von Methoden:
Die Methoden werden nur anhand ihres Namens identifiziert daher muß Junit Doclet hier keine Änderungen vornehmen.

Installation:

Das Junit Doclet ist Komandozeilen-basierter Code Generator, was den Vorteil der Unabhängigkeit von IDEs hat. Trotzdem ist aber analog zu Javadoc eine Integration in IDEs möglich und zwar als externes Tool. Dies wird in diesem Abschnitt am Beispiel von Eclipse gezeigt.
Ebenso kann man JunitDoclet aber auch in Ant-Skripte integrieren

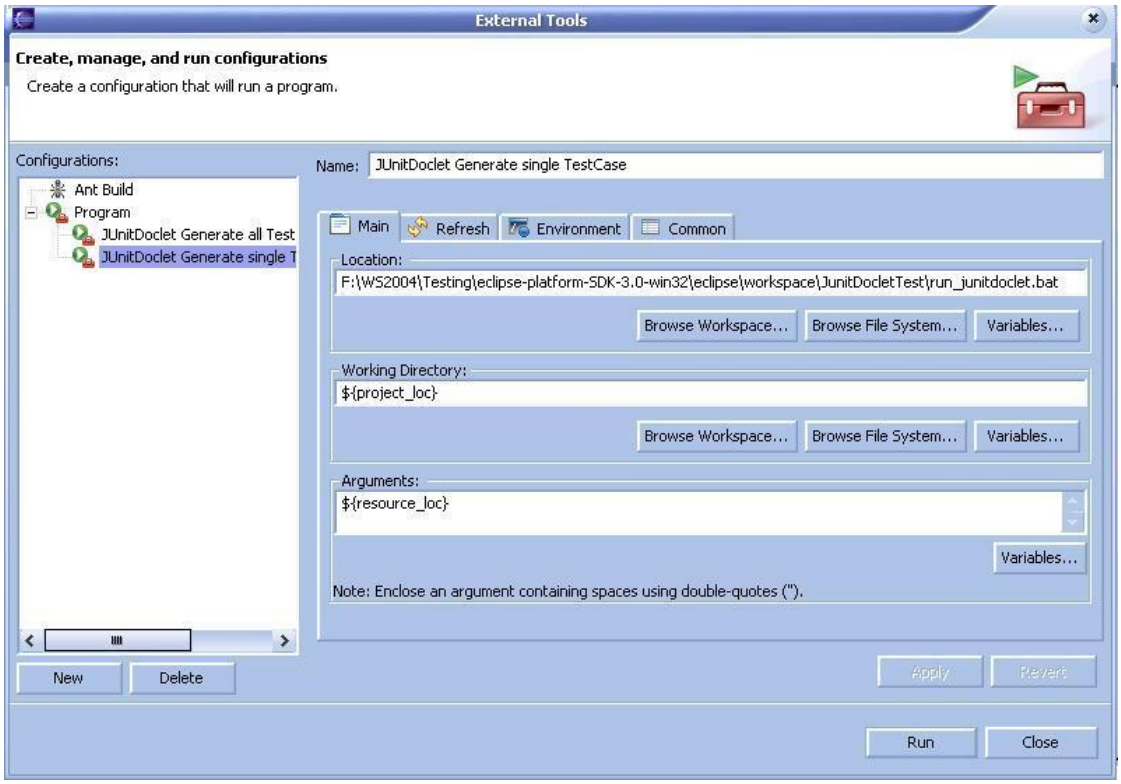
Ant Integration:
Beispiel für die Einbindung in ein Ant-Build Script:

```
<project name="junitdoclet-sample" default="all">
...
<target name="junitdoclet" depends="compile">
<javadoc
packagenames = " testen.von.softwaressystemen.*"
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>
```

Eclipse Integration:

Zur Integration in Eclipse kann man auf drei beim Download enthaltene Batch Dateien zurückgreifen. In „*run_junitdoclet_setenv.bat*“ müssen zuerst einige Pfadanpassungen (JDK Path, JunitDoclet Path,…) vorgenommen werden. Danach können in Eclipse unter „*Run → External Tools → External Tools…*“ das Batch File „*run_junitdoclet_recursive.bat*“ (benötigt den Projektpfad *${project_loc}* als Working-Directory) und „*run_junitdoclet.bat*“ (benötigt den Projektpfad *${project_loc}* als Workung-Directory und die markierte Datei *${resource_loc}* als Argument) eingebunden werden.



Über „*Run → External Tools*“ kann man nun die Batchdateien jederzeit starten.
Um eine Testsuite mit Testfällen für ein gesamtes Package zu erstellen muss dieses im Package-Explorer ausgewählt sein und „*run_junitdoclet_recursive.bat*“ gestartet werden.
Um Testfälle aus einer einzelnen Java Datei zu generieren muß diese ebenfalls im Package-Explorer makiert sein, das zu startende Batch File ist dann aber „*run_junitdoclet.bat*“

JUnit Test Case Builder (Jub):

Einleitung:

Dieses Code Generator Framework ist als Erweiterung für IDE entwickelt worden, mit dem Ziel Testfälle innerhalb der IDE erstellen zu lassen und dort gleich bearbeiten zu können. Mit Jub sollte somit bewusst ein anderer Weg zu den verbreiteten Kommandozeilen-basierten Generatoren gewählt werden.
Jub ist in Java programmiert worden und derzeit erst in der Version 0.1.2 verfügbar. Diese Version stammt allerdings schon aus dem Jahr 2002 was darauf schließen lässt das Jub nicht oder kaum noch weiterentwickelt wird, obwohl dies aufgrund einige ärgerliche Bugs bei Installation und Bedienung dringend erforderlich wäre.
Daher ist die Integration von Jub auch erst in 2 IDEs überhaupt möglich, nämlich in Eclipse (als Plugin) und in ‚VisualAge for Java' (als IDE utility Class Libary).
In dieser Ausarbeitung wurde die Funktionsweise von Jub (nur) mit Eclipse getestet

Features:

JUB kann neben den üblichen Testfälle für alle public Methoden auch Testfälle für protected und default Methoden generieren.
Weiters ist JUB auch in der Lage mehrfache Konstruktoren und überladene Methoden zu berücksichtigen.
Im Gegensatz zu anderen Testgeneratoren versucht JUB neben den Methoden auch tatsächlichen Testcode zu generieren. Der Testcode wird auf Basis der Methodenparameter erzeugt. Den Eingangsparameter wird der Wert *0* oder *null* zugewiesen. Die zu testende Methode wird mit diesen Eingangsparametern ausgeführt und das Ergebnis wird dem Ausgangsparameter zugewiesen.
Zum Erstellen von einfachen Tests kann dieses Feature durchaus hilfreich sein da nur mehr die Parameterwerte angepasst und die entsprechende assert-Abfrage eingebaut werden muss. Für aufwendige Tests ist der generierte Code aber natürlich unbrauchbar, da der Methodeninhalt bei der Generierung nicht berücksichtigt wird.

Methode aus einem Java Programm:

```
public String getRightAlignedIntString(int len, int a) {
// doesn't matter for JUB whats happening here
}
```

JUB erzeugt daraus folgenden Test:

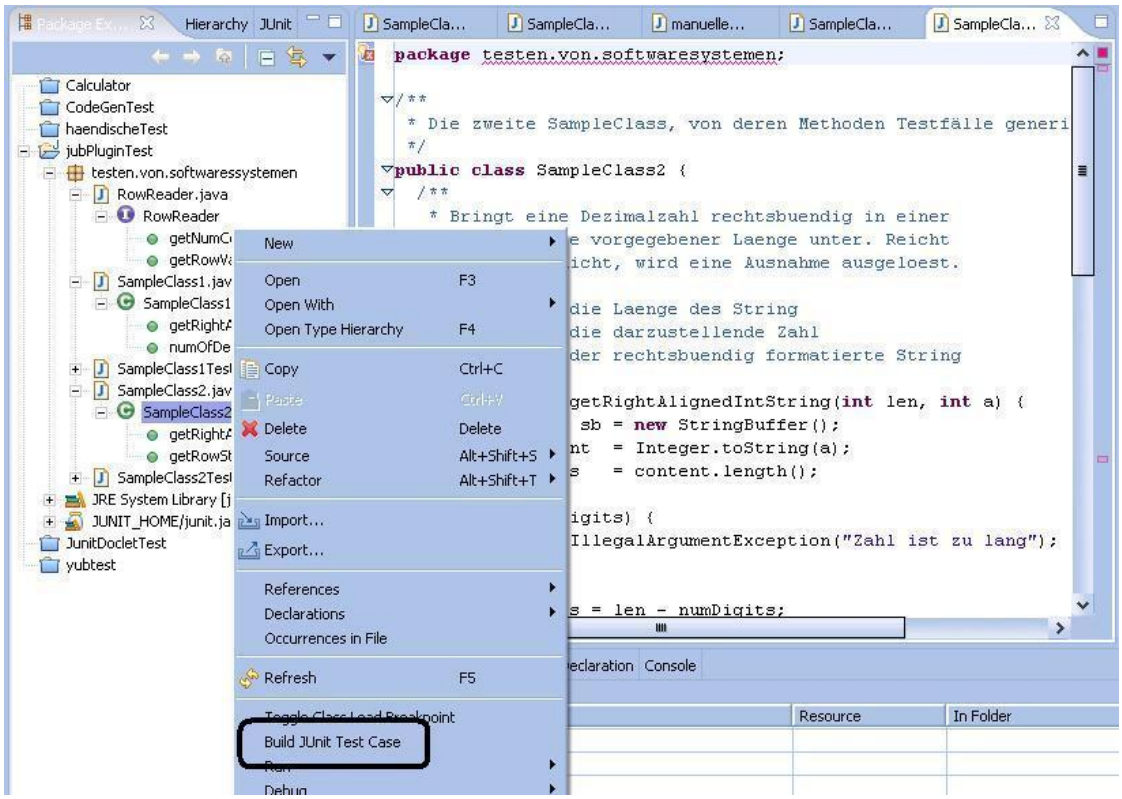
```
public void testGetRightAlignedIntString() {
fail("Newly generated method - fix or disable");

//insert code testing basic functionality
  SampleClass2 var0 = null;
  int var1 = 0;
  int var2 = 0;
  String var3 = null;
  var3 =
  var0.getRightAlignedIntString(var1,
  var2);
}
```

Installation:

Die Installtion und Integration in Eclipse sollte schon aufgrund der Zielsetzung des Projekts sehr einfach vonstatten gehen. Tatsächlich ist es ausreichend im Eclipse Plugin Ordner das Verzeichnis "*net.sourceforge.jub*" zu erzeugen und die Jub Dateien dorthin zu kopieren. Beim Start von Eclipse sollte das Plugin dann erkannt und aktiv sein. Bei unseren Tests war allerdings unter Umständen eine Neuinstallation von Eclipse notwendig damit das Plugin erkannt wurde, was entweder am Plugin-Handling von Eclipse oder an Jub selbst liegt.

Erzeugen von Testfällen:

Hat man die Installationshürde genommen, findet sich fortan beim Anklicken einer Java Klasse im Pop-Menü der Eintrag „*Build JUnit Test Case*", über den ein neues Java-File mit den Testfällen automatisch generiert wird. Leider hat sich beim Testen des Code Generators gezeigt, das man vor allem Glück braucht damit Jub die Testfälle tatsächlich generiert. Manchmal reicht ein Neustart von Eclipse aus, bei manchen Java Klassen haben wir es auch nie geschafft Testcode zu generieren, was aber nicht unbedingt daran liegt das sich aus der Java Klasse kein Testcode generieren lassen würde. Ein Versuch hat gezeigt das zu der gleichen Java Klasse in 2 verschiedenen Eclipse Projekten die Testfälle einmal erzeugt wurden und einmal nicht.
Hier zeigt sich der frühe Entwicklungsstand des Generators noch mehr als deutlich zumal man nicht mal eine Fehlermeldung erhält wenn der Testcode nicht generiert wurde.



Java Bean Tester:

Einleitung:

Der Java Bean Tester kann von der Seite http://javabeantester.sourceforge.net/download.html bezogen werden. Es soll gleich vorrausgeschickt werden, dass es sich beim Java Bean Tester nicht um einen Testcodegenerator handelt. Vielmehr ist dieses Werkzeug als Rahmen für den JUnit gestützten Java Beans - Test anzusehen. Zu beachten ist, dass zur Verwendung des Werkzeuges JUnit vonnöten ist.

Installation:

Nachdem die Datei Jbeantester05.zip auf den Rechner heruntergeladen wurde muss sie in ein eigenes Verzeichnis entpackt werden. Es werden 3 JAR-files entpackt, ein jbeantester1.3.jar, ein jbeantester1.4.jar und ein src.jar. In der src.jar befinden sich die Sourcen der Implementierung, daher kann diese Datei einmal ignoriert werden. Die beiden anderen JAR-Dateien sind die Archive für den compilierten Code der Versionen 1.3 und 1.4. Da 1.3 noch weitere Zusatzinstallationen benötigt wenden wir uns vorzugsweise der Version 1.4 zu.
Zuletzt ist es zu empfehlen sich eine Kopie des JUnit – Codes in Form der Datei JUnit.jar ins Verzeichnis zu holen, alternativ kann aber auch die CLASSPATH Umgebungsvariable in der Systemsteuerung -> System -> Reiter „Erweitert“ mit einem vollständigen Verweis auf die Datei JUnit.jar in ihrem Stammverzeichnis ergänzt werden.

Verwendung:

Der Java Bean Tester kann nicht als alleinstehendes Utility verwendet werden, sondern ist im Grunde in bestehende Projekte „einzuprogrammieren“.
Leider ist zur Funktionalität des Java Bean Testers keine angemessene Dokumentation verfügbar, der Interessierte wird auf der Produkthomepage auf die automatisch generierte JavaDoc verwiesen, die alleine allerdings auch noch nicht viele der anstehenden Fragen beantworten kann. Daher ist man gezwungen einen Blick auf ein Progammierbeispiel zu werfen, welches die Funktionsweise des Java Bean Testers veranschaulichen soll.
Dieses Beispiel ist im Stammverzeichnis unter tests\jbeantester\tests\ die Datei JBeanTest.java.
Um dieses Beispiel durchlaufen zu lassen öffnet man ein Fenster mit Eingabeaufforderung und gibt folgendes ein: java -cp jbeantester1.4.jar;junit.jar jbeantester.tests.JBeanTest.
Die Ausgabe sollte dann in etwa wie folgt aussehen:

...
Time: 0,04

OK (3 tests)

Das Vorgehen bei eigenen Tests ist folgendermaßen: Zu schreibende Testklassen erben von der Klasse jbeantester.junit.BaseTestCase und führen Test an Beans anhand der Klasse jbeantester.framework.JBeanTester durch. Diese besitzt die Methoden testJBeanConstructor und testJBeanProperties welche beide ein JBeanTestResults Objekt liefern. Anhand dieses Objektes können die verschiedensten Sachverhalte(wie Vorhandensein eines Properties, Gesamtzahl der Properties) über das getestete Bean abgefragt werden.
Diese Sachverhalte können wiederum wieder mit gewöhnlichen JUnit-Assertstatements mit den erwarteten Werten verglichen werden.

WTTOOLS UNIT TEST GENERATOR

Einleitung:

Das Tool kann von der Seite http://sourceforge.net/project/showfiles.php?group_id=31579 heruntergeladen werden. Es ist aber unbedingt darauf zu achten, dass man nicht nur den gezippten Sourcecode herunterlädt, sondern gleich den binären Code, weil man sich dabei viele Mühen beim Übersetzten und aber auch beim Ausführen ersparen kann. Die aktuelle Version des Tools ist die 2.0.2 entsprechend sollte man die Datei unittestsgen-2.0.2.zip aus der binär Sektion auf den Rechner laden, die Source Version kann man zusätzlich herunterladen um Einblick in Details der Implementierung zu haben.

Installation:

Zuerst muss der binäre Zip in ein Verzeichnis entpackt werden. Ich gehe davon aus, dass Java bereits auf dem Rechner installiert ist und das Hauptverzeichnis davon in die Umgebungsvariable PATH gespeichert wurde. Es sind dann zwei Verzeichnisse im Stammverzeichnis des Tools anzulegen, die Verzeichnisse „src“ und „test-src“. Alle Sourcedateien für deren Methoden man Testfälle schreiben will sind in diesen Ordner zu kopieren. Weiters sollte man das JUnit Jar-File das man von www.junit.org beziehen kann in den „jar“ Ordner kopieren, dadurch ist das Handling des Tools leichter weil man einfachere Classpaths angeben kann.

Benutzung:

Es muss entweder direkt mit der Kommandozeile gearbeitet werden oder man schreibt sich eine kleine Batchdatei, die die Aufgaben für das Erzeugen der Testklassen automatisiert.
Als erstes muss man in das Hauptverzeichnis des Tools wechseln also in ..\unittestsgen2.0.2.
Dort muss man zuerst die Sourcen im „src“ Verzeichnis in Code übersetzen mittels „javac src*.java“. Danach befinden sich die Klassen im Sourceverzeichnis.
Als nächsten Schritt kann man sich die ersten Testsourcen erzeugen lassen. Dazu gibt man das folgende ein: „java –jar jar\unittestsgen.jar src\Beispiel.java -cp src\;jar\junit.jar“.
Falls man hinten vergisst die JUnit mit anzugeben gibt es eine Fehlermeldung, es werden aber trotzdem die TestSourcen erzeugt. Warum JUnit vonnöten ist konnte ich bis jetzt noch nicht herausfinden. JUnit wird auch wenn das JAR-file angegeben wird nicht ausgeführt.
Mittels „javac test -src*.java -classpath jar\junit.jar;src;.” werden jetzt die einzelnen Testfälle kompiliert und anschliessend mittels „java -cp "test-src;jar/junit.jar"junit.swingui.TestRunner BeispielTestCase“ ausgeführt.
Für die Benutzung des Unit Tests Generator gibt es auch eine Anzahl von Kommandozeilenoptionen:

-i: gibt den Pfad zu den EingabeSourcen an, Standard ist „src“
-o: gibt den Ausgabepfad zu den TestSourcen an, Standard ist „test-src“
-cf KonfigDatei: Man gibt eine Datei an in der alle Parameter zum Aufruf gespeichert sind
-n Muster: Das Muster gibt an wie die erzeugten Testklasse benannt werden, gibt man z.B. „*TestCase“ an dann wird der Name der Quellsource genommen und daran TestCase drangehängt um den vollständigen Namen der Testklasse zu erhalten
-cp: Damit kann man Verzeichnisse in den Classpath geben, obwohl man vorne mit dem Java Aufruf keine Möglichkeit dazu hatte bsp: „java –jar unittestsgen.jar src*.java –cp src\“
-f: Man erzwingt das Überschreiben von TestSourcen, Nachteil ist, dass ergänzte TestSourcen wieder mit blanken Testmethoden überschrieben werden
-v: Versionnummer anzeigen

-g: none|code|comments|both
-d: Zeigt zusätzliche Debuginformation
-r: Erzeugt auch Testklassen für Sourcen in Unterverzeichnissen
-q: keine Ausgabe am Bildschirm

Code:

Der vom Unit Tests Generator erzeugte Code:
Im Folgenden wird anhand eine kleinen Beispiels anschaulich gemacht welche Synthesefähigkeiten der Unit Tests Generator besitzt. Dazu wird für ein kleines Beispielsprogramm welches aus lediglich 4 Prozeduren besteht Testcode erzeugt.
Das von mir verwendete Programm sieht folgendermaßen aus:

```
public class Beispiel {

      public static void main(String[] args){
               System.out.println("Die Hauptmethode wurde aufgerufen!");
      }

      public int add(int a, int b){
               return a+b;
      }

      public int subtract(int a, int b){
               return a-b;
      }

      public int multiply(int a, int b){
            return a*b;
      }
}
```

Test Klassen:

Unit Tests Generator erzeugte daraus folgende Test Klassen:
BeispielTestCase.java
TestAll.java

In TestAll.java befindet sich nur ganz wenig Code(Kommentare wurden weggelassen):

```
import junit.framework.*;

public class TestAll {

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  } // end of main(Stringp[ args)
  public static Test suite() {
    TestSuite suite= new TestSuite("All JUnit Tests");
    suite.addTest(BeispielTestCase.suite());
    return suite;
  } // end of suite()
} // end of TestAll
```

Es werden also erwartungsgemäß nur alle Testfälle in eine Suite verpackt.

Interessanter für den Entwickler ist allerdings der Inhalt von BeispielTestCase.java.
Die erzeugten TestMethoden schauen so aus:

```
public void testNoMethods() {
}



public void testAdd104431104431() {
    assertTrue("Warning! This is new test method with no real test code
    inside.", false);
}

public void testMain1888107655() {
    assertTrue("Warning! This is new test method with no real test code
    inside.", false);
}

public void testMultiply104431104431() {
    assertTrue("Warning! This is new test method with no real test code
    inside.", false);
}

public void testSubtract104431104431() {
    assertTrue("Warning! This is new test method with no real test code
    inside.", false);
}
```

Man erkennt, dass erstens nur einfache Methoden ohne Aufruf der getesteten Methode erzeugt werden, weiters dass auch die Methode main und eine allgemeine Testmethode erzeugt werden. Die Methodennamen werden alphabetisch geordnet und bekommen als Anhängsel eine Zahl, die die Eindeutigkeit der Methoden absichert.
Laut JavaDoc Dokumentation soll es angeblich auch möglich sein, Testfälle zu generieren, die in den Sourcen definiert werden, aber diese Funktionalität konnte ich leider noch nicht nachvollziehen.
Der Unit Tests Generator bietet auch die Möglichkeit als Ant Target ausgeführt zu werden.
Dazu wird bei jedem Verarbeitungsvorgang der Sourcefiles auch ein sample-junit-target.xml erstellt, das in das build.xml des aktuellen Softwareprojektes einfließen kann.
Dadurch erreicht man, dass für das aktuelle Projekt bei jedem Ant-Build nach der Compilierung auch neue Testklassen erzeugt werden. Zu beachten ist aber, dass bestehende Testklassen normalerweise nicht überschrieben werden, bzw. nur wenn man die explizite Option –f einsetzt, wobei diese auch vorhandenen Testcode in den Methoden wieder löscht.
Daher ist zum gegenwärtigen Zeitpunkt eine gute Praxistauglichkeit des Werkzeuges aus meiner Sicht noch nicht gegeben. Das Tool ist aber sicherlich hilfreich, wenn man bereits eine Anzahl Sourcen hat und dann im Nachhinein Tests dafür schreiben will.

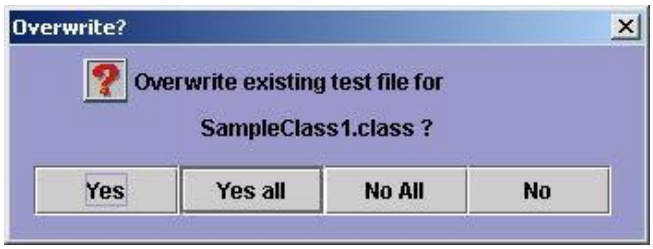
JUnit Test Generator:

Einleitung:

Der TestGenerator1a konnte von Source Forge (http://prdownloads.sourceforge.net/junittestmaker/TestGenerator1a.zip?download) heruntergeladen werden. Der Generator liegt sowohl als kompilierte Version, als auch als Quelltext vor. Eine mitgelieferte Batch Datei startet unverzüglich das GUI.

Features:

Dieses Tool kann neben den üblichen Testfälle für alle public Methoden auch Testfälle für protected und default Methoden generieren.
Der Testgenerator greift nicht wie die restlichen TestCodeErzeuger auf die java Datei, den Quellcode, zu, sondern benötigt die kompilierten Klassen.
Bei Generierung der TestCode java Dateien wird vor Erstellung dieser überprüft ob eine gleichnamig Testklasse existiert und mittels Dialogfeld darauf aufmerksam gemacht. Diese Vorgehensweise verhindert unabsichtliches Überschreiben von modifizierten Testklassen und somit den Verlust der händisch erstellten Testfälle.



Installation:

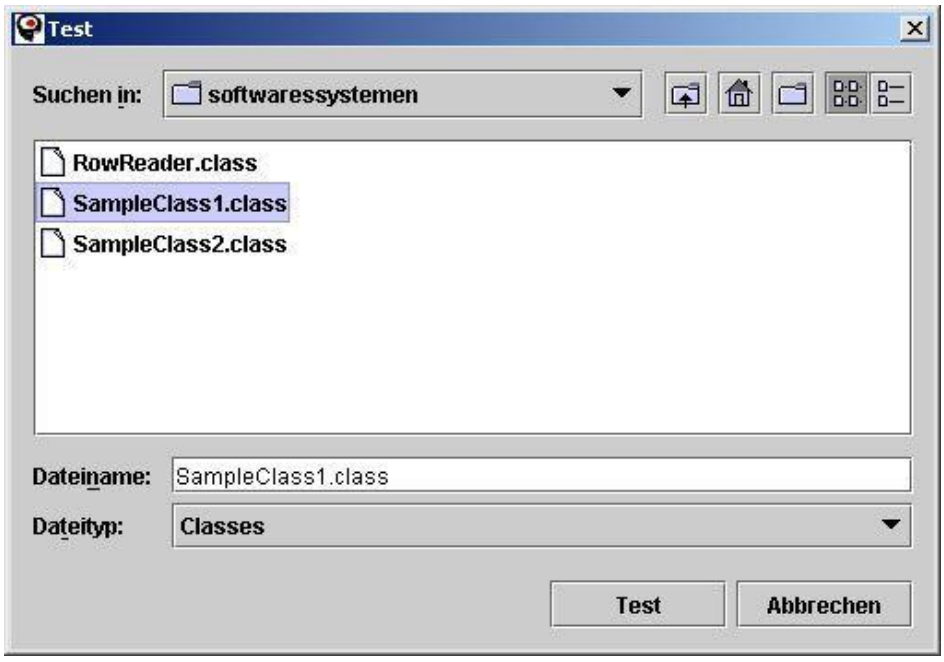
Nach dem Entpacken der zip Datei befindet sich der Quellcode im Verzeichnis „TestGenerator1a“. Im Ordner „testing“ liegen die kompilierten Class Dateien. Mittels der Batch Datei „StartGUI.bat“ wird der TestGenerator1a gestartet und man sollte nach dem Startscreen folgende Oberfläche erhalten.

Alternativ kann man das Tool auch mittels „java testing.TestGenerator“ von der Kommandozeile aus starten.
Unter „Options/Setting“ kann man den Ausgabepfad für den generierten Testcode definieren, genauso wie Angaben über den Testcode Ersteller (Autor, Datum, Revision) .

Erzeugen von Testfällen:

Die Auswahl zur Erzeugung der Testfälle benötigt die kompilierten Klassen und ist auf eine Datei pro Generierung beschränkt. Auf Grund dieser fehlenden Mehrfachselektion muss jede Klasse einzeln selektiert und mittels „Test“ wird für diese die dazugehörige Datei mit dem erzeugten TestCode erstellt. Der erzeugte Testcode beinhaltet nur das Gerüst ohne Testfälle und Sicherungsmaßnahmen. Als Sicherungsmaßnahmen ist eine defaultmäßige „fail“ Anweisung für jede Funktion zu verstehen, die positive JUnitTestergebnissen aufgrund fehlender Testfälle verhindert.
Die Erzeugung von TestCode basierend auf Interfaces liefert keine Fehlermeldung aber auch keine Testfälle. Man liest lediglich „Total test generated: 0.“

Auswahl der Datei für die der TestGenerator1a den TestCode automatisch erzeugen soll.



Nachdem eine Datei ausgewählt wurde erhält man folgende Informationen. Als erstes die absolute Pfadangabe der class Datei, danach den Dateinamen, der den class-Namen mit einem angehängtem „Test“ erweitert, des erzeugten TestCodes und zu Letzt die Anzahl der erfolgreich erstellten Tests.

Vergleich der generierten Testklassen

Im folgenden Abschnitt werden die erzeugten Testklassen von einigen Code Generatoren miteinander verglichen
Zum Vergleich herangezogen werden die Ergebnisse des Junit Doclets, Webmind Junit Test Generators, Junit Test Case Builders und des WTTOOLS Unit Tests Generators.
Zu diesem Zweck lassen wir von allen Generatoren die Tests zu einem selbsterstellten Java-Package erzeugen.
(Anm.: bei den Quellcodes der automatisch generierten Java Files wurden manche Kommentare entfernt oder gekürzt)

Beschreibung des Testpackages

Im Package testen.von.softwaressystemen befinden sich drei Java Dateien von denen die Code Generatoren die Testklassen erzeugen sollen:
RowReader.java, SampleClass1.java, SampleClass2.java

SampleClass1.java

```
package testen.von.softwaresystemen;

/**
 * Die erste SampleClass, von deren Methoden Testfälle generiert werden sollen.
 */
public class SampleClass1 {
  /**
   * Eine erste zu testende Methode:
   * Wieviel Stellen braucht eine Dezimalzahl?
   *
   * @param a die Zahl, deren Stellenzahl zu ermitteln ist
   * @return  die Anzahl der Dezimalstellen
   */
  public int numOfDecimalDigits(int a) {
    int digits = 0;
    a = Math.abs(a);

    do {
      digits++;
      a = a/10;
    } while (a != 0);

    return digits;
  }

  /**
   * Bringt eine Dezimalzahl rechtsbuendig in einer
   * Zeichenkette konstanter Laenge unter.
   *
   * @param a  die darzustellende Zahl
   * @return der rechtsbuendig formatierte String
   */
  public String getRightAlignedIntString(int a) {
    StringBuffer sb = new StringBuffer();
    int numDigits   = numOfDecimalDigits(a);
    int numSign     = (a < 0) ? 1 : 0;
    int numSpaces   = 11 - numDigits - numSign;

    for (int i = 0; i < numSpaces; i++) {
       sb.append(" ");
         }

         sb.append(a);
         return sb.toString();
       }
}
```

SampleClass1.java enthält zwei Methoden zu denen die Code Generatoren Tests erzeugen sollen. Beide Methoden erwarten einen Integer als Eingabewert und als Rückgabewert einmal Integer und einmal String. Abgesehen davon, dass sich diese Klasse im selben Package wie die anderen Klassen befindet gibt es keine Abhängigkeiten.

SampleClass2.java

```
package testen.von.softwaresystemen;

/**
 * Die zweite SampleClass, von deren Methoden Testfälle generiert werden sollen.
 */
public class SampleClass2 {
  /**
   * Bringt eine Dezimalzahl rechtsbuendig in einer
   * Zeichenkette vorgegebener Laenge unter. Reicht
   * der Platz nicht, wird eine Ausnahme ausgeloest.
   *
   * @param len die Laenge des String
   * @param a   die darzustellende Zahl
   * @return    der rechtsbuendig formatierte String
   */
  public String getRightAlignedIntString(int len, int a) {
    StringBuffer sb = new StringBuffer();
    String content  = Integer.toString(a);
    int numDigits   = content.length();

    if (len<numDigits) {
      throw new IllegalArgumentException("Zahl ist zu lang");
    }

    int numSpaces = len - numDigits;

    for (int i = 0; i < numSpaces; i++) {
      sb.append(" ");
    }
    sb.append(content);
    return sb.toString();
  }

  /**
   * Berechnung einer Tabellenzeile im Textformat (Leerzeichen als
   * Fuellzeichen). Die Methode baut auf der Formatierung einer Zahl
   * in einen rechtsbuendigen String (getRightAlignedIntString) auf.
   * Als Quelle der Daten wird ein anderes Objekt uebergeben, das
   * aber bis zu einem Interface mit nur zwei Methoden abstrahiert ist.
   * So wird es einfacher, fuer Testzwecke einmal ein ein Mock-Objekt
   * zu uebergeben.
   *
   * @param rowReader   eine Referenz auf eine Quelle fuer die Daten einer Zeile
   * @param columnWidth Spaltenbreite
   * @return String mit den rechtsbuendig formatierten Zahlen einer Zeile
   */
  public String getRowString(RowReader rowReader, int columnWidth) {
    StringBuffer sb = new StringBuffer();
    int columns;
    int value;

    if (rowReader != null) {
      columns = rowReader.getNumColumns();
      for (int i=0; i<columns; i++) {
        value = rowReader.getRowValue(i);
        sb.append(getRightAlignedIntString(columnWidth, value));
      }
    }
    return sb.toString();
  }
}
```

SampleClass2.java enthält ebenfalls zwei Methoden mit Eingangs –und Ausgangsparameter. Die Methode `getRightAlignedIntString(…)` benötigt im Gegensatz zur gleichnamigen Methoden in SampleClass1 allerdings zwei (Integer) Eingangsparameter.
Die zweite Methode `getRowString(…)` benötigt die Methoden `getRowValue(…)` und `getNumColumns(…)` die im RowReader Interface (Datei **RowReader.java**) deklariert sind. Auf einen Implementierung dieser beiden Dateien wurde der Einfachheit halber verzichtet.
Um die Methode aber trotzdem testen zu können muss dann aber zumindest in der Testklasse eine Implementierung als Mock-Funktion erfolgen (siehe Kapitel „händisch erzeugte Testfälle“)

RowReader.java

```
package testen.von.softwaresystemen;

/**
 * Interface deren Methoden von SampleClass2 benötigt werden
 * Dieses Interface abstrahiert von einer zeilenorientierten
 * Datenquelle (z.B. einem Datenbank-Record, einer Zeile einer
 * CSV-Datei, einem XML-Element mit seinen Attributen, ...).
 * Diese Schnittstelle wird von einer anderen Klasse implementiert,
 * die auch mehrere Zeilen adressieren kann.
 * Fuer unsere Testfallgenerierungen ist eine solche Klasse aber
 * nicht notwendig deswegen habe wir darauf verzichtet
 */
public interface RowReader {
  /**
   * Ermittelt die Spaltenzahl der Zeile.
   *
   * @return Anzahl der Spalten
   */
  public int getNumColumns();

  /**
   * Ermittelt den Wert in einer bestimmten Spalte der Zeile.
   *
   * @param column Spalte (0 bis numColumns-1)
   * @return Zahlenwert der gegebenen Spalte als ganze Zahl.
   */
  public int getRowValue(int column);
}
```

Händisch erzeugte Junit Testklassen:

Zum besseren Vergleich mit den Testklassen der Code Generatoren zeigen wir auch wir eine händisch erzeugte Testsuite aussehen könnte. Man sollte an dieser Stelle allerdings nochmals darauf hinweisen, dass die Generatoren immer nur die Struktur der Testklassen erzeugen und niemals die Tests selber, die in diesem händischen Fall zum Teil schon enthalten sind.
Es wurde eine Testsuite erzeugt die die Tests für beide Java Klassen enthält.
Die Methoden aus SampleClass1 und SampleClass2 werden einigen Assert-Prüfungen unterzogen.

Um `getRowString(…)` aus SampleClass2 zu testen mußten in SampleClass2Test die Interface Methoden in irgendeiner Form (als Mock-Funktionen) implementiert werden.

manuelleTestSuite.java

```
package testen.von.softwaressystemen;

import junit.framework.TestSuite;

public class manuelleTestSuite
{

  public static TestSuite suite() {
    TestSuite suite;
    suite = new TestSuite("testen.von.softwaresystemen");
    suite.addTestSuite(testen.von.softwaressystemen.SampleClass2Test.class);
    suite.addTestSuite(testen.von.softwaressystemen.SampleClass1Test.class);

    return suite;
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

SampleClass1Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;

public class SampleClass1Test extends TestCase {

  public SampleClass1Test(String name) {
    super(name);
  }

  public void testNumOfDecimalDigits() {
    SampleClass1 sample = new SampleClass1();

    // 2 Beispiele für mögliche Testfälle
    assertEquals(1, sample.numOfDecimalDigits(0));
    assertEquals(1, sample.numOfDecimalDigits(1));

    sample = null;
  }

  public void testGetRightAlignedIntString() {
    SampleClass1 sample = new SampleClass1();

    // 2 Beispiele für mögliche Testfälle
    assertEquals("          0", sample.getRightAlignedIntString(0));
    assertEquals("          1", sample.getRightAlignedIntString(1));

    sample = null;
  }
}
```

SampleClass2Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;
import junit.textui.TestRunner;

/**
 * Die Testklasse selbst implementiert das Interface, was fuer einen
 * der Tests gebraucht wird. Die Methoden werden zum Testen jetzt
 * hier als Mock-Funktionen implementiert
 */
public class SampleClass2Test extends TestCase implements RowReader
```

```
{
  testen.von.softwaressystemen.SampleClass2 sample = null;

  int mockNumColumns;

  public int getRowValue(int column) {
    /* Fuer Testzwecke wird einfach die Spaltennummer zurueckgegeben.    */
    return column;
  }

  public int getNumColumns() {
    /* Fuer die Tests muss die Spaltenzahl natuerlich erst einmal gesetzt
       sein. In der setUp-Methode wird der Wert erst einmal auf drei
       festgelegt.  */
    return mockNumColumns;
  }

  public SampleClass2Test(String name) {
    super(name);
  }
  /**
   * Erzeuge die Instanz der zu testenden Klasse in einer gesonderten Methode
   *
   * @return eine Instanz der zu testenden Klasse.
   * @throws Exception Wenn etwas schiefgeht wird das im JUnit-Framework
   * aufgefangen und als Fehler gewertet.
   */
  public testen.von.softwaressystemen.SampleClass2 createInstance() throws
Exception {
    return new testen.von.softwaressystemen.SampleClass2();
  }

  /**
   *
   * @throws Exception Wenn etwas schiefgeht wird das im JUnit-Framework
   * aufgefangen und als Fehler gewertet.
   */
  protected void setUp() throws Exception {
    super.setUp(); // Eventuell hat die ererbte Methode noch etwas zu tun?

    sample  = createInstance(); // Der Testaufbau besteht in diesem
                                // Beispiel nur aus einer Instanz der
                                // zu testenden Klasse.

    mockNumColumns = 3;         // allgemeines Verhalten der Mock-
                                // Funktion getNumColumns festlegen.
  }

  /**
   *
   * @throws Exception Wenn irgendwas schiefgeht wird das im JUnit-Framework
   * aufgefangen und als Fehler gewertet.
   */
  protected void tearDown() throws Exception {
    sample = null;    // Referenzen auf nicht mehr benutzte Instanzen
                      // werden bewusst auf null gesetzt, um es dem
                      // Garbage Collector einfacher zu machen.
    super.tearDown(); // Eventuell hat die ererbte Methode noch etwas zu tun?
  }

  public void testGetRightAlignedIntString() throws Exception {
    // 2 Beispiele für mögliche Testfälle
    assertEquals("          0", sample.getRightAlignedIntString(11, 0));
    assertEquals("          1", sample.getRightAlignedIntString(11, 1));

    try {
      String buffer=sample.getRightAlignedIntString(10, Integer.MIN_VALUE );
      fail("Es sollte eine Exception geworfen werden (buffer="+buffer+")");
    } catch (IllegalArgumentException e) {
```

```
      // Alles in Ordnung
    }
  }

  /**
   *
   * @throws Exception Wenn etwas schiefgeht wird das im JUnit-Framework
   * aufgefangen und als Fehler gewertet.
   */
  public void testGetRowString() throws Exception {
    // 2 Beispiele für mögliche Testfälle
    assertEquals("012",       sample.getRowString(this, 1));
    assertEquals(" 0 1 2",    sample.getRowString(this, 2));
  }
}
```

1. Vergleich: erzeugte Testsuiten

Von den für den Vergleich herangezogenen Tools waren alle bis auf JUB in der Lage komplette Testsuiten zum Ausführen von mehreren Testklassen zu erzeugen.

Junit Doclet: SoftwaresystemenSuite.java

```
package testen.von.softwaressystemen;

import junit.framework.TestSuite;

// JUnitDoclet begin import
// JUnitDoclet end import

public class SoftwaressystemenSuite
// JUnitDoclet begin extends_implements
// JUnitDoclet end extends_implements
{
  // JUnitDoclet begin class
  // JUnitDoclet end class

  public static TestSuite suite() {

    TestSuite suite;

    suite = new
TestSuite("testen.von.softwaressystemen");

suite.addTestSuite(testen.von.softwaressystem
en.SampleClass1Test.class);

suite.addTestSuite(testen.von.softwaressystem
en.SampleClass2Test.class);

    // JUnitDoclet begin method suite
    // JUnitDoclet end method suite

    return suite;
  }

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

Unit Tests Generator: TestAll.java

```
import junit.framework.*;

/**
 * TestSuite that runs all the tests.
 * You can run unit tests in many ways,
however this file is designed
 * performing tests in the following way:<br/>
 * <pre>
 *   java -cp
"jar/thisjarfile.jar;lib/junit.jar" TestAll
 * </pre>
 *
 */
public class TestAll {

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  } // end of main(Stringp[ args)
  public static Test suite() {
    TestSuite suite= new TestSuite("All JUnit
Tests");

suite.addTest(testen.von.softwaresystemen.Samp
leClass1TestCase.suite());

suite.addTest(testen.von.softwaresystemen.Samp
leClass2TestCase.suite());
    return suite;
  } // end of suite()
} // end of TestAll
```

2. Vergleich: Testklassen zu SampleClass1.java

Junit Doclet:
SampleClass1Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;
// JUnitDoclet begin import
import
testen.von.softwaressystemen.SampleClass1;
// JUnitDoclet end import


public class SampleClass1Test
// JUnitDoclet begin extends_implements
extends TestCase
// JUnitDoclet end extends_implements
{
  // JUnitDoclet begin class
  testen.von.softwaressystemen.SampleClass1
sampleclass1 = null;
  // JUnitDoclet end class

  public SampleClass1Test(String name) {
    // JUnitDoclet begin method
    super(name);
    // JUnitDoclet end method SampleClass1Test
  }

  public
testen.von.softwaressystemen.SampleClass1
createInstance() throws Exception {
    // JUnitDoclet begin method
    return new
testen.von.softwaressystemen.SampleClass1();
    // JUnitDoclet end method
  }

  protected void setUp() throws Exception {
    // JUnitDoclet begin method testcase.setUp
    super.setUp();
    sampleclass1 = createInstance();
    // JUnitDoclet end method testcase.setUp
  }

  protected void tearDown() throws Exception {
    // JUnitDoclet begin method
    sampleclass1 = null;
    super.tearDown();
    // JUnitDoclet end method
  }

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

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

  /** JUnitDoclet moves marker to this method,
if there is not match
  * for them in the regenerated code and if
the marker is not empty.
  * This way, no test gets lost when
regenerating after renaming.
  * Method testVault is supposed to be empty.
  */
  public void testVault() throws Exception {
    // JUnitDoclet begin method
    // JUnitDoclet end method
  }
```

Unit Tests Generator:
SampleClass1TestCase.java

```
package testen.von.softwaresystemen;

import junit.framework.*;
import junit.extensions.*;
import java.awt.Toolkit;
import java.awt.AWTEvent;
import java.awt.event.AWTEventListener;
import java.awt.event.WindowEvent;

public class SampleClass1TestCase extends
TestCase
{
  /** Instance of tested class. */
  protected SampleClass1 varSampleClass1;

  /** Public constructor for creating testing
class. */
  public SampleClass1TestCase(String name) {
    super(name);
  } // end of SampleClass1TestCase(String
name)

 /** This main method is used for run tests
for this class only
   * from command line. */
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  } // end of main(Stringp[] args)

 /** This method is called every time before
particular test execution.
   * It creates new instance of tested class
and it can perform some more
   * actions which are necessary for performs
tests. */
  protected void setUp() {
Toolkit.getDefaultToolkit().addAWTEventListene
r(new AWTEventListener() {
        public void eventDispatched(AWTEvent
event) {
          WindowEvent we = ((WindowEvent)
event);
          if (we.getID() ==
WindowEvent.WINDOW_OPENED)
            we.getWindow().dispose();
        }
      }, AWTEvent.WINDOW_EVENT_MASK);
    varSampleClass1 = new
testen.von.softwaresystemen.SampleClass1();
  } // end of setUp()

  /** Returns all tests which should be
performed for testing class.
   * By default it returns only name of
testing class. Instance of this
   * is then created with its constructor. */
  public static Test suite() {
    return new
TestSuite(SampleClass1TestCase.class);
  } // end of suite()

 /** for classes which doesn't contain any
methods here is one additional
   * method for performing test on such
classes. */
  public void testNoMethods() {
  }

  /** Method for testing original source
method:
   * java.lang.String
getRightAlignedIntString(int)
```

```
  public static void main(String[] args) {
    // JUnitDoclet begin method testcase.main

junit.textui.TestRunner.run(SampleClass1Test.c
lass);
    // JUnitDoclet end method testcase.main
  }
}
```

```
   * from tested class */
  public void
testGetRightAlignedIntString104431() {
    assertTrue("Warning! This is new test
method with no real test code inside.",
false);
  } // end of testGetRight…

  /** Method for testing original source
method:
   * int numOfDecimalDigits(int)
   * from tested class */
  public void testNumOfDecimalDigits104431() {
    assertTrue("Warning! This is new test
method with no real test code inside.",
false);
  } // end of testNumOf...

} // end of SampleClass1TestCase
```

Junit Test Case Builder:
SampleClass1Test.java

```
package testen.von.softwaressystemen;

import junit.framework.*;

public class SampleClass1Test extends TestCase
{
       //declare reusable objects to be used
across multiple tests
  public SampleClass1Test(String name) {
    super(name);
  }

  public static void main(String[] args) {
junit.textui.TestRunner.run(SampleClass1Test.c
lass);
  }

  public static Test suite() {
    return new
TestSuite(SampleClass1Test.class);
  }

  protected void setUp() {
    //define reusable objects to be used
across multiple tests
  }

  protected void tearDown() {
    //clean up after testing (if necessary)
  }

  public void testNumOfDecimalDigits() {
    fail("Newly generated method - fix or
disable");

    //insert code testing basic functionality
    SampleClass1 var0 = null;
    int var1 = 0;
    int var2 = 0;
    var2 = var0.numOfDecimalDigits(var1);
  }

  public void testGetRightAlignedIntString() {
    fail("Newly generated method - fix or
disable");

    //insert code testing basic functionality
    SampleClass1 var0 = null;
    int var1 = 0;
    String var2 = null;
    var2 =
var0.getRightAlignedIntString(var1);
  }
}
```

Webmind Junit Test Generator:
// SampleClass1Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * SampleClass1Test    (Copyright 2001 Your
Company)
 *
 * <p> This class performs unit tests on
testen.von.softwaressystemen.SampleClass1 </p>
 *
 * <p> Explanation about the tested class and
its responsibilities </p>
 *
 * <p> Relations:
 *     SampleClass1 extends java.lang.Object
<br>
 *
 * @author Your Name Your email - Your Company
 * @date $Date: $
 * @version $Revision: $
 *
 * @see
testen.von.softwaressystemen.SampleClass1
 * @see some.other.package
 */

public class SampleClass1Test extends TestCase
{

  /**
   * Constructor (needed for JTest)
   * @param name    Name of Object
   */
  public SampleClass1Test(String name) {
    super(name);
  }

  /**
   * Used by JUnit (called before each test
method)
   */
  protected void setUp() {
    //sampleclass1 = new SampleClass1();
  }

  /**
   * Used by JUnit (called after each test
method)
   */
  protected void tearDown() {
    sampleclass1 = null;
  }
```

```
  /**
   * Test the constructor: SampleClass1()
   */
  public void testSampleClass1() {

  }

  /**
   * Test method: int numOfDecimalDigits(int)
   */
  public void testNumOfDecimalDigits() {
    //Must test for the following parameters!
    int intValues [] = {-1, 0, 1,
Integer.MAX_VALUE, Integer.MIN_VALUE};

  }

  /**
   * Test method: String
getRightAlignedIntString(int)
   */
  public void testGetRightAlignedIntString() {
    //Must test for the following parameters!
    int intValues [] = {-1, 0, 1,
Integer.MAX_VALUE, Integer.MIN_VALUE};

  }

  /**
   * Main method needed to make a self
runnable class
   *
   * @param args This is required for main
method
   */
  public static void main(String[] args) {
    junit.textui.TestRunner.run( new
TestSuite(SampleClass1Test.class) );
  }
  private SampleClass1 sampleclass1;
}
```

3. Testklassen zu SampleClass2.java

Junit Doclet:
SampleClass2Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;
// JUnitDoclet begin import
import
testen.von.softwaressystemen.SampleClass2;
// JUnitDoclet end import

public class SampleClass2Test
// JUnitDoclet begin extends_implements
extends TestCase
// JUnitDoclet end extends_implements
{
  // JUnitDoclet begin class
  testen.von.softwaressystemen.SampleClass2
sampleclass2 = null;
  // JUnitDoclet end class

  public SampleClass2Test(String name) {
    // JUnitDoclet begin method
SampleClass2Test
    super(name);
    // JUnitDoclet end method SampleClass2Test
  }

  public
testen.von.softwaressystemen.SampleClass2
```

Unit Tests Generator:
SampleClass2TestCase.java

```
package testen.von.softwaresystemen;

import junit.framework.*;
import junit.extensions.*;
import java.awt.Toolkit;
import java.awt.AWTEvent;
import java.awt.event.AWTEventListener;
import java.awt.event.WindowEvent;

public class SampleClass2TestCase extends
TestCase {
  /** Instance of tested class. */
  protected SampleClass2 varSampleClass2;

  /** Public constructor for creating testing
class. */
  public SampleClass2TestCase(String name) {
    super(name);
  } // end of SampleClass2TestCase(String
name)

 /**
   * This main method is used for run tests
for this class only
   * from command line.
   */
  public static void main(String[] args) {
```

```
createInstance() throws Exception {
    // JUnitDoclet begin method
testcase.createInstance
    return new
testen.von.softwaressystemen.SampleClass2();
    // JUnitDoclet end method
testcase.createInstance
  }

  protected void setUp() throws Exception {
    // JUnitDoclet begin method testcase.setUp
    super.setUp();
    sampleclass2 = createInstance();
    // JUnitDoclet end method testcase.setUp
  }

  protected void tearDown() throws Exception {
    // JUnitDoclet begin method
    sampleclass2 = null;
    super.tearDown();
    // JUnitDoclet end method
  }

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

  public void testGetRowString() throws
Exception {
    // JUnitDoclet begin method getRowString
    // JUnitDoclet end method getRowString
  }

  /** JUnitDoclet moves marker to this method,
if there is not match
  * for them in the regenerated code and if
the marker is not empty.
  * This way, no test gets lost when
regenerating after renaming.
  * Method testVault is supposed to be empty.
  */
  public void testVault() throws Exception {
    // JUnitDoclet begin method
    // JUnitDoclet end method
  }

  public static void main(String[] args) {
    // JUnitDoclet begin method testcase.main
junit.textui.TestRunner.run(SampleClass2Test.c
lass);
    // JUnitDoclet end method testcase.main
  }
}
```

```
    junit.textui.TestRunner.run(suite());
  } // end of main(Stringp[] args)

 /** This method is called every time before
particular test execution.
   * It creates new instance of tested class
and it can perform some more
   * actions which are necessary for performs
tests.
   */
  protected void setUp() {
Toolkit.getDefaultToolkit().addAWTEventListene
r(new AWTEventListener() {
        public void eventDispatched(AWTEvent
event) {
          WindowEvent we = ((WindowEvent)
event);
          if (we.getID() ==
WindowEvent.WINDOW_OPENED)
            we.getWindow().dispose();
        }
      }, AWTEvent.WINDOW_EVENT_MASK);
    varSampleClass2 = new
testen.von.softwaresystemen.SampleClass2();
  } // end of setUp()

 /** Returns all tests which should be
performed for testing class.
   * By default it returns only name of
testing class. Instance of this
   * is then created with its constructor. */
  public static Test suite() {
    return new
TestSuite(SampleClass2TestCase.class);
  } // end of suite()

/** for classes which doesn't contain any
methods here is one additional
   * method for performing test on such
classes. */
  public void testNoMethods() {
  }

  /** Method for testing original source
method:
   * java.lang.String
getRightAlignedIntString(int, int)
   * from tested class */
  public void
testGetRightAlignedIntString104431104431() {
    assertTrue("Warning! This is new test
method with no real test code inside.",
false);
  } // end of testGetRight...

  /** Method for testing original source
method:
   * java.lang.String getRowString( ... )
   * from tested class */
  public void
testGetRowString435669498104431() {
    assertTrue("Warning! This is new test
method with no real test code inside.",
false);
  } // end of testGetRow...

} // end of SampleClass2TestCase
```

Junit Test Case Builder:
SampleClass2Test.java

```
package testen.von.softwaressystemen;

import junit.framework.*;
import .RowReader;

public class SampleClass2Test extends TestCase
```

Webmind Junit Test Generator:
// SampleClass2Test.java

```
package testen.von.softwaressystemen;

import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
```

```
{
  //declare reusable objects to be used across
multiple tests
  public SampleClass2Test(String name) {
    super(name);
  }

  public static void main(String[] args) {
        junit.textui.TestRunner.run(SampleClas
s2Test.class);
  }

  public static Test suite() {
              return new
TestSuite(SampleClass2Test.class);
  }

  protected void setUp() {
    //define reusable objects to be used
across multiple tests
  }

  protected void tearDown() {
    //clean up after testing (if necessary)
  }

  public void testGetRightAlignedIntString() {
    fail("Newly generated method - fix or
disable");

    //insert code testing basic functionality
    SampleClass2 var0 = null;
    int var1 = 0;
    int var2 = 0;
    String var3 = null;
    var3 = var0.getRightAlignedIntString(var1,
var2);
  }

  public void testGetRowString() {
    fail("Newly generated method - fix or
disable");

    //insert code testing basic functionality
    SampleClass2 var0 = null;
    RowReader var1 = null;
    int var2 = 0;
    String var3 = null;
    var3 = var0.getRowString(var1, var2);
  }
}
```

```
 * SampleClass2Test    (Copyright 2001 Your
Company)
 *
 * <p> This class performs unit tests on
testen.von.softwaressystemen.SampleClass2 </p>
 *
 * <p> Explanation about the tested class and
its responsibilities </p>
 *
 * <p> Relations:
 *     SampleClass2 extends java.lang.Object
<br>
 *
 * @author Your Name Your email - Your Company
 * @date $Date: $
 * @version $Revision: $
 *
 * @see
testen.von.softwaressystemen.SampleClass2
 * @see some.other.package
 */

public class SampleClass2Test extends TestCase
{

  /**
   * Constructor (needed for JTest)
   * @param name    Name of Object
   */
  public SampleClass2Test(String name) {
    super(name);
  }

  /**
   * Used by JUnit (called before each test
method)
   */
  protected void setUp() {
    //sampleclass2 = new SampleClass2();
  }

  /**
   * Used by JUnit (called after each test
method)
   */
  protected void tearDown() {
    sampleclass2 = null;
  }

  /**
   * Test the constructor: SampleClass2()
   */
  public void testSampleClass2() {

  }

  /**
   * Main method needed to make a self
runnable class
   *
   * @param args This is required for main
method
   */
  public static void main(String[] args) {
    junit.textui.TestRunner.run( new
TestSuite(SampleClass2Test.class) );
  }
  private SampleClass2 sampleclass2;
}
```

Ein Vergleich der erzeugten Testklassen zeigt sofort das die Unterschiede im Detail liegen. Alle Generatoren implementieren die Standardmethoden (setUp(), tearDown(), Konstruktur) auf ähnliche bis idente Art und Weise und die Testklassen verfügen auch standardmäßig über main-Methoden um die Testfälle auch außerhalb der Testsuite ausführbar zu machen.

Spezielle Eigenheiten sind beispielsweise, dass nur JunitDoclet dem User die Möglichkeit bietet, ohne weiteres alle generierten Methoden selbständig zu erweitern. (Usercode der zwischen //JunitDoclet begin method und //JunitDoclet end method steht wird beim nächsten generieren nicht gelöscht). Weiters enthält JunitDoclet Code eine testVault() Methode in diese der Testcode des Benutzers automatisch gesichert wird bevor er beispielsweise durch Refaktoring verloren geht.
Der Unit Tests Generator hat in seinem Code wiederum eine eigene testNoMethods() Methode vorgesehen in der Klassen getestet werden können die keine Methoden besitzen. Jub zeichnet sich durch besonders minimalistischen Code aus und versucht als einziger Generator bereits ansatzweise Testcode in den zwei Testmethoden (test…()) einzubauen.
Unit Tests Generator liefert im Unterschied zum Jub keinen generierten Testcode von Methoden sondern Vorschläge.

```
„
//Must test for the following parameters!
int intValues [] = {-1, 0, 1, Integer.MAX_VALUE, Integer.MIN_VALUE};
“
```

bzw

```
„
//sampleclass2 = new SampleClass2();
“
```

Die Testfälle müssen zwar selbst erstellt werden, jedoch bekommt man ein nützliches Variablen Konstrukt, auf welches man zurückgreifen kann.
Der Unit Test Generator, genauso wie manch andere Test Code Generatoren, liefert Kommentare, welche zum automatischen Erzeugen der Test Code Dokumentation dienen. Im Falle vom TestGenerator1a wird beim Import ausschließlich auf die junit Bilibliothek „framework“ (Testcase und Testsuite) zurückgegriffen.

Literaturverzeichnis:

- **„Testing without excuses“**,
  Andreas Braig and Steffen Gemkow
- **„Junit Doclet – Getting Started”**,
  http://www.junitdoclet.org/getting_started.txt
- **„JUB (Junit Test Case Builder)”** ,
  http://jub.sourceforge.net/
- **„Testfieber: Unit-Tests als Sicherheitsnetz für Programmierer“,**
  Karsten Violka, Steffen Gemkow, Martin Uhlig, c't 13/03, Seite 226
- **„Offener Werkzeugkasten: Java-Software entwickeln mit Eclipse“,**
  Stefan Matthias Aust, Michael Rusitska (kav) , c't 03/03, Seite 196
- **„Soft-Werkzeugkasten: Eclipse 3.0“**
  Stefan Matthias Aust, Frank Tonn (kav), c't 15/04, Seite 213