Mutation Testing mit Jester

Patrick Haruksteiner
9955634*

Stefan Hauser
0055964†

Helmut Rohregger
0056851‡

Thomas Schneider
9555098§

Testen von Softwaresystemen
November 2004

*patrick.haruksteiner@students.jku.at
†mail@stefanhauser.net
‡helmut.rohregger@students.jku.at
§thomas.schneider@students.jku.at

Inhaltsverzeichnis

Zusammenfassung

Jester ist ein Tool für Mutationstests von Java Programmen. Bei Mutationstests werden bestimmte Teile im Sourcecode verändert und es wird geprüft ob die Tests mit diesen Änderungen noch bestanden werden. Dadurch soll das aufspüren von Fehlern in den Tests oder im Sourcecode der Programme erleichtert werden.

1 Einleitung

Jester ist ein Java Tool um Codestücke aufzufinden die nicht von Testfällen abgedeckt werden. Es geht dabei einen anderen Weg als übliche Code-Coverage Tools. Jester ändert Teile des Sourcecode, kompiliert ihn neu und überprüft ob die Tests mit den vorgenommenen Änderungen immer noch bestanden werden. Sollte ein Test trotz Änderungen bestanden werden benachrichtigt Jester den Programmierer darüber und gibt an welche Änderungen es vorgenommen hat.
Jester kann nicht nur den zu testenden Code abändern, sonder auch die zum testen verwendeten Tests. Schlägt ein geänderter Test nicht fehl kann so nachgewiesen werden, dass er entweder redundant oder fehlerhaft ist.
[Moo03, Moo04a]

1.1 Verfügbarkeit

Jester ist in Java programmiert und verwendet JUnit zum ausführen der Tests. Daher ist Jester theoretisch für jede Plattform für die es eine aktuelle Javaversion gibt verfügbar. Eine zusätzliche wichtige Voraussetzung ist, dass JUnit auch auf dieser Plattform lauffähig ist. Es sind aber auch Ports für Python (Pester/PyUnit) und C# (Nester) erhältlich. Die Tatsache, dass Jester Open Source ist macht es wegen der daraus resultierenden Kostenersparnis zusätzlich interessant. So kann ohne Kosten uneingeschränkt getestet werden, ob es den persönlichen Anwendungsanforderungen genügt.
[Moo04b]

2 Mutation Testing

2.1 Sinnhaftigkeit

Mutation Testing geht weniger nach der Devise vor, ob das Projekt genug getestet wurde, sondern eher ob die Tests ausreichen um Fehler im Projekt zu finden. Ziel ist also eher die Erstellung möglichst guter Testfälle als das auffinden von Programmierfehlern. Darin liegt aber der Vorteil, dass gerade dadurch das Auffinden von Fehlern erleichtert wird.
Um erzeugtem Code vertrauen zu können muss er getestet werden. Daher vertrauen Programmierer ihrem Code wenn er die Tests fehlerfrei durchläuft und ihren Test wenn diese Fehler aufzeigen. Um die Tests zu testen bauen manche Programmierer absichtlich Fehler in ihren Code ein um festzustellen ob diese auch von ihren Tests erkannt werden. In manchen Projektteams gibt es dazu so genannte *Project Saboteurs* die absichtlich Fehler in das Projekt einbauen und überprüfen ob diese von den Tests erfasst werden.
Diese Aufgaben können durch die Verwendung von Mutation Test Tools vereinfacht werden. Derartige Tools finden Code der nicht getestet wurde oder redundant ist. Denn auch

redundanter Code wirkt sich negativ auf die Projektentwicklung aus, da damit wertvolle Entwicklungszeit beim lesen oder verändern vergeudet wird.
[Ada03, Inc04, Moo04a]

2.2 Grundlagen

Mutation Testing basiert auf zwei Hypothesen:

- die Programmierer sind kompetent
- es existiert ein Kopplungseffekt

Die erste Hypothese besagt, dass Programmierer üblicherweise kompetent sind und wissen was sie tun. Sie schreiben also beinahe korrekte Programme. Das erzeugte Programm unterscheidet sich vom korrekten Programm nur in Kleinigkeiten.
Beim Kopplungseffekt geht man davon aus, dass große Fehler meist mit kleinen Fehlern gekoppelt sind. Große Fehle sind meist semantischer Natur, wogegen kleine Fehler fast ausschließlich syntaktische Fehler sind. Diese syntaktischen Fehler können durch Mutationstests aufgespürt werden. Auf diese Art kann man über kleine Fehler die größeren finden.
Mutation Testing ist ein fehlerbasierendes Whitebox Testverfahren. Selten wird dafür auch der Begriff *Active Nonlinear Testing* verwendet. Die Qualität der Testfälle wird dabei nach ihrer Effektivität und der Fähigkeit Fehler aufzufinden beurteilt.
[Ada03, Inc04, Moo04a]

2.3 Funktionsweise

Mutation Test Tools arbeiten mit so genannten *Mutanten*. Mutanten sind veränderte Versionen des ursprünglichen Programms, die sich nur durch eine Mutation von jenem unterscheiden. Jede Mutation entspricht einem typischen Fehler wie sie oft in Programmen auftreten. Beispiele hierfür wären Tippfehler (z.B. + statt −), falsche Werte (`1` statt `0` in Schleifen → *off-by-one*) und Ähnliches.
Zum testen werden Mutanten des Programms erstellt und überprüft ob diese Mutanten die vorgegebenen Testfälle bestehen. Die Erzeugung der Mutanten geschieht über *Mutationsoperatoren*. Diese werden auch als *Mutagene, Mutationstransformationen* oder *Mutationsregeln* bezeichnet und in verschiedene Klassen unterteilt (Statement, Operator, Variable, Konstante, usw.). Zum Erstellen der Mutanten wird folgende Vorgehensweise verwendet:

- verändern von Ausdrücken durch das ersetzen von Operatoren und das einfügen neuer Operatoren
- ersetzen aller Operanden durch jeden syntaktisch erlaubten anderen Operand
- entfernen einzelner Statements

Folgendes Beispiel soll zeigen wie Mutanten das verhalten von Programmen beeinflussen können und trotzdem bei schlecht entworfenen Tests als bestanden durchgehen:

Originalprogramm:

```
...
  x = y + z;
...
```

Mutant:

```
...
  x = y * z;
...
```

Bei `y = 3` und `z = 2` würde der Test fehlschlagen, da `x` ein falsches Ergebnis liefert.

```
x = 5
```

```
x = 6
```

Bei `y = 2` und `z = 2` würde der Test jedoch kein falsches Ergebnis für `x` anzeigen.

```
x = 4
```

```
x = 4
```

Bei derartigen Mutationstest können *äquivalente Mutanten* auftreten. Diese sind syntaktisch verschieden vom Originalprogramm, besitzen aber das gleiche Verhalten. Ein großes Problem mit äquivalenten Mutanten liegt darin zu entscheiden ob die Äquivalenz auf alle möglichen Testfälle zutrifft. Dieses Problem ist in der Regel unentscheidbar und der versuch es zu lösen würde ein exhaustives Testen erfordern. Folgendes Beispiel zeigt einen derartigen Mutanten:

Originalprogramm:

```
...
if(y==2 && z==2)
  x = y + z;
...
```

```
x = 4
```

äquivalenter Mutant:

```
...
if(y==2 && z==2)
  x = y * z;
...
```

```
x = 4
```

Wenn die Bedingung erfüllt ist liefert der Mutant für jeden möglichen Testfall das gleiche Ergebnis wie das Originalprogramm (`x = 4`). Es handelt sich daher hierbei um einen äquivalenten Mutanten.

Der Verlauf von Mutationstests kann folgenderweise beschrieben werden:

- einschleusen eines Fehlers durch anwenden eines Mutationsoperators auf das Programm
- feststellen ob der Fehler durch das Testen gefunden wurde
  - bei unterschiedlichem Ergebnis wurde der Mutant durch den Test gefunden (*killed mutant*)
  - bei gleichem Ergebnis wurde der Mutant nicht gefunden, es handelt es sich entweder um einen äquivalenten Mutant, oder der Testfall ist nicht ausreichen (*alive mutant*)

[Ada03, Moo04a]

2.4 Testbewertung

Um ein Maß für die Qualität der Testfälle angeben zu können wird oft der *Mutation Score* benutzt. Der Mutation Score MS eines Programms P und Test T ist definiert als:

$$MS(P,T) = \frac{killed\ mutants}{non\text{-}equivalent\ mutants}$$

$$\text{mit } non\text{-}equivalent\ mutants = \ total\ mutants - \ equivalent\ mutants$$

$$\text{und } 0 \leq MS \leq 1 \text{ oder } 0 \leq MS\% \leq 100$$

Die Testdaten sind mutationsadäquat, wenn der Mutation Score bei 100% liegt. In diesem Fall wurden alle nicht äquivalenten Mutanten gefunden.
[Ada03]

2.5 Probleme

Bei der Anwendung von Mutationstests stößt man bald auf einige Probleme:

- es ist schwer äquivalente Mutanten herauszufiltern
- manche nicht-äquivalente Mutanten sind schwer zu beseitigen
- es wird ein enormer Rechenaufwand benötigt um alle möglichen Mutanten gegen alle Testfälle zu testen
- hoher Arbeitsaufwand um mutationsadäquate Testfälle zu entwickeln
- hoher Zeitaufwand um die Ergebnisse der Mutationstests zu interpretieren

[Ada03]

3 Arbeitsweise von Jester

Jester modifiziert den Java Quellcode mit sehr einfachen Mitteln, welche kein Parsen benötigen. Dazu wird ein textbasierender *(*search-and-replace) Algorithmus verwendet. Es werden immer nur Änderungen an jeweils einer Quelledatei vorgenommen. Von Jester vorgenommene Änderungen sind:

- modifizieren von Literalen: z.B. aus einer `0` wird eine `1`
- ändern von boolean Variablen: `true` ↔ `false`
- ändern von `if(` nach `if(true ||`
- ändern von `if(` nach `if(false &&`

Die letzten beiden Änderungen haben den Effekt, dass die `if` Bedingung entweder immer wahr oder immer falsch ist. Auf den ersten Blick scheint es einfacher gleich die ganze Bedingung durch `if(true)` oder `if(false)` zu ersetzen. Auf diese Weise wäre es aber notwendig die Bedingung zu parsen um ihr Ende zu finden, was durch die von Jester verwendete Art

der Abänderung vermieden wird. Da zeitgleich immer nur eine Änderung am Quellcode vorgenommen wird und diese vor der nächsten Änderung wieder rückgängig gemacht wird, kann es nie vorkommen, dass sich Modifikationen gegenseitig aufheben.
Diese einfachen Modifikationen haben sich bisher als sehr effektiv erwiesen. In zukünftige Version von Jester soll versuchet werden höher entwickelte Änderungen zu implementieren welche zu noch besseren Ergebnisse führen.
Jester verwendet eine modifizierte Version der Klasser `textui.TestRunner` von JUnit die nur `PASSED` oder `FAILED` zurückgibt. Mehr Information wird von Jester nicht benötigt. Um den Java Compiler zum erneuten kompilieren des modifizierten Quellcode aufzurufen benutzt Jester `Runtime.getRuntime().exec("javac ...")`. Der Java Compiler läuft nicht direkt in Java sondern im darunter liegenden Betriebssystem. Damit soll es ermöglicht werden Jester so modifizieren zu können auch Compiler für anderer Sprachen aufzurufen.
Über `Runtime.getRuntime().exec("java jester.TestRunnerImpl ...")` werden die Tests in einer neuen Instanz der Java Virtual Machine gestartet. Die einzelnen Tests müssen in unabhängigen Java VMs laufen, da eine geladen Klasse nicht einfach durch ein erneutes kompilieren des veränderten Source neu geladen wird. Jester benötigt aber immer die aktuelle, veränderte Version der Klasse.
Folgendes Beispiel stellt mögliche von Jester generierte Mutationen dar:

Mutation 1:

```
boolean someValue = somethingThatVaries;
if(true || someValue){
   System.out.println("Hello World");
}
```

Mutation2:

```
boolean someValue = somethingThatVaries;
if(false && someValue){
   System.out.println("Hello World");
}
```

Ein schlecht gestaltete Test kann z.B. immer ohne Fehlermeldung durchlaufen werden, egal ob `println()` immer oder niemals ausgeführt wird. In diesem Fall würde der Test trotz Mutationen gelingen, daher wird der Programmierer von Jester über die Änderungen benachrichtigt, mit denen der Test immer noch ohne Fehler durchläuft.
[MN04, Moo04a]

4 Vergleich mit Code-Coverage Tools

Code Coverage Tools versuchen zu zeigen welche Teile des Codes nicht von der Test Suite behandelt worden sind. Das kann sehr nützlich sein um redundanten Code beziehungsweise vermissende Tests zu finden und in manchen Fällen würde dies sogar einfacher zu verstehen sein als die Ergebnisse die Jester liefert.
Jedoch können Tests auch Code ausführen wobei das Ergebnis nicht überprüft wird, dies heißt aber das Code Coverage Tools zwar Code ausführt, diesen aber nicht auf Richtigkeit des Ergebnisses prüft. Jester dagegen kann auch nicht getesteten Code aufspüren, selbst wenn er bei den Tests ausgeführt wird.

Des Weiteren kann Jester Hinweise auf die Art der Tests, die noch implementiert werden müssen, geben indem Jester anzeigt wie es den Code manipuliert hat, sodass trotzdem die Tests noch einwandfrei durchlaufen werden.
Es folgen Beispiele für zwei Arten von Code Coverage Tools, und zwar für *Line Coverage* und für *Path Coverage*:

JProbe Coverage (Line Coverage Tool):
http://www.sitraka.com/software/jprobe/jprobecoverage.html

```
boolean someValue = false;
if(someValue){
   System.out.println("Hello World");
}
```

Man sieht welche Zeilen nicht von diesem Tool überprüft worden sind. Es ist sehr einfach zu benützen und die Ergebnisse sind leicht verständlich. Jedoch ist das Maß des Ergebnisses nicht wirklich aussagekräftig. Zudem ist dieses Tool kommerziell und daher kostenaufwendiger als Jester.

Hansel (Path Coverage Tool):
http://hansel.sf.net

```
boolean someValue = true;
if(someValue){
   System.out.println("Hello World");
}
```

Vorteile dieses Tools sind seine einfache Benutzbarkeit und die Integrierbarkeit in JUnit. Es ist gleich wie Jester als Open Source verfügbar. Ein Nachteil ist, dass nur geprüft wird welche Teile von einem Test abgedeckten werden und nicht welches Ergebnis sie liefern. Deshalb wird es hier wichtig gute Test Fälle zu schreiben. Obiges Beispiel zeigt einen Fall in dem zwar alle Zeile durchlaufen werden, da `someValue` in der `if`-Anweisung nie `false` ist, aber die daraus resultierende Redundanz wird nicht erkannt.
[MN04, Moo04a]

5 Vor- & Nachteile von Jester

Jester ist ein sehr aussagekräftigstes Tool, wenn man ein Maß angeben möchte wie gut getestet worden ist. Jester ist außerdem als Open Source verfügbar. Weiter Vorteile von Jester sind das auffinden von fehlenden Tests und redundantem Code. Besonders das auffinden fehlender Test kann viel Kosten ersparen, weil dadurch Fehler aufgedeckt werden könne die sonnst unentdeckt geblieben wären.
Die Nachteile liegen jedoch in dem hohen Zeitaufwand. Einerseits um die Tests auszuführen und diese andererseits die von den Entwicklern zu Interpretation der Tests aufzuwendende Zeit. Diese Interpretation kann sich als schwierig erweisen, da die Ergebnisse nicht immer einfach zu verstehen sind.

Dazu kommen noch Probleme welche mit der derzeit aktuellen Version (Jester 1.35) auftreten:

- Falls der Code aus irgendwelchen Gründen nicht kompiliert, kann Jester in einer Endlosschleife hängen bleiben.
- Wird Jester von ANT gestartet, so wird die Konfigurationsdatei nicht gefunden. Stattdessen werden die Standardwerte verwendet.

[Moo04a, Moo04b]

6 Verwendung von Jester

6.1 Installation

- Entpacken der gezippten Datei
- JDK (Version $\geq$ 1.4) muss installiert und der Pfad zu `java` und `javac` gesetzt sein
- JUnit (Version $\geq$ 3.7) muss installiert und der Classpath gesetzt sein
- Jester muss zum Classpath hinzugefügt werden:
  `set classpath=%classpath%;INSTALL_DIR\jester.jar`
  Für Windows gibt es für diesen Zweck eine Batchdatei (`setcp.bat`)
- Um testen zu können, ob die Installation erfolgreich war, kann ein einfaches Beispiel durch Starten von `test.bat` oder `acceptancetest.bat` ausgeführt werden
- Die Tools die im Packet enthalten sind benötigen Python (Version $\geq$ 2.0)

[Moo04b]

6.2 Hinweis vor der Verwendung

Vor der eigentlichen Erläuterung, wie Jester verwendet wird, muss folgendes abgeklärt werden: Da Jester direkte Änderungen im Sourcecode unternimmt, sollte Jester nur auf eine Kopie der Sourcen ausgeführt werden. Die bessere Variante ist die Verwendung eines Source Control Management Systems. Bei Verwendung eines solchen muss darauf geachtet werden, den aktuelle Source in das Repository zu übertragen bevor der Jester Test ausgeführt wird. Hier nur eine kleine Auswahl an SCM Systemen: CVS, Subversion, SourceSafe, ...
[Moo04b]

6.3 Befehlsausführung

Um Jester ausführen zu können braucht es die JUnit Test Klasse und das Verzeichnis in dem der Source Code enthalten ist um herauszufinden, ob die Testfälle die Veränderungen abdecken. Die Ausführung von Jester kann eine ganze Weile dauern, deshalb ist es ratsam Jester nur einen kleinen Teil des Codes testen zu lassen, bevor alles getestet wird.
Um Jester auszuführen muss `java jester.TestTester TEST_CLASS SOURCE_DIRECTORY` in die Konsole eingegeben werden. `TEST_CLASS` ist der Name der JUnit Test Klasse, die für den Source geschrieben wurde (die Klasse für den JUnit TestRunner). Normalerweise ist dies die

TestAll Klasse des Java Packages. `SOURCE_DIRECTORY` ist das Verzeichnis oder ein einzelnes File welches den Source Code des eigentlichen Programms enthält. Jester versucht in diesem Source Code Änderungen durchzuführen um die JUnit Tests zu testen.
Wird Jester unter der Ausführung mit CTRL-C unterbrochen, so besteht das Risiko, dass der Source Code nicht mehr zum Ausgangszustand zurückgeführt wird. Dadurch kann es passieren, dass Jester nicht mehr auf den Code ausgeführt werden kann, solange die Änderungen nicht rückgängig gemacht werden.
[Moo04b]

6.4 Ausgabe

Für alle Änderungen die Jester macht, sodass die Testfälle nicht fehlschlagen, gibt es den Namen der veränderten Datei, die Position der Veränderung in der Datei und einen Ausschnitt des Codes, um diesen in der Datei leicht finden zu können, am Bildschirm aus.
Jester produziert zusätzlich eine Datei namens `jesterReport.xml`, welche jene Änderungen enthält die Jester durchführte, bei denen die Tests nicht fehlgeschlagen sind.
Weiters wird die Datei `jesterTimeout.txt` erzeugt, welche die Dauer in Millisekunden enthält, die für den ersten Testlauf benötigt wurde. Dies ist notwendig, um den Testlauf zu stoppen falls dieser zu lange dauert (Endlosschleife usw.).
[Moo04b]

6.5 Tools

Die Tools sind alle in Python geschrieben, deshalb ist es notwendig Python ≥ 2.0 zu installieren, um diese zu verwenden.
Die Ausführung von `python makeAllChangesFiles.py jesterReport.xml` generiert Kopien des originalen Source Codes, welche die Veränderungen von Jester enthalten. Diese Dateien befinden sich im selben Verzeichnis wie der Original-Code mit dem selben Namen, aber `.jester` als Dateierweiterung (konfigurierbar). Dies ist nützlich um einen Vergleich der Orginal-Dateien mit den veränderten Dateien durchführen zu können.
Die Ausführung von `python makeWebView.py` generiert Kopien des originalen Source Codes als `.html` Seiten. Diese enthalten die Veränderungen die Jester machte und die Testfälle, die nicht fehlschlagen sind, dargestellt in roter Farbe. Der geänderte Code wird durchgestrichen angezeigt. Weiters wird eine Datei `jester.html` generiert, die zu allen Dateien linkt.
[Moo04b]

6.6 Resultate

In [Moo04a] wird gezeigt welche Resultate Jester bei einem kleinen Programm liefert. Hierzu wird das `Money`-Beispiel, welches bei JUnit 3.2 mitgeliefert wird, verwendet. Dieser Source Code ist ungefähr 400 Zeilen lang.
Jester machte 47 Modifikationen, von denen 10 die Testfälle nicht fehlschlagen ließen. Diese Änderungen wiesen auf fehlende Tests oder redundanten Code hin. Allein drei der Modifikationen wurden in der `equals` Methode der `Money` Klasse gemacht:

```
public boolean equals(Object anObject)
{
  if (isNull())
    if (anObject instanceof IMoney)
      return ((IMoney)anObject).isNull();
...
```

Eine der Modifikation war folgende: `if (isNull())` wurde durch `if (false && isNull())` ersetzt. Dies verrät, dass `isNull()` immer `false` bei den Testfällen ist. Das heißt, es fehlt ein Testfall für den Fall wo `isNull()` gleich `true` ist. In [Moo04a] sind noch weitere Fälle dargestellt.

7 Konfiguration von Jester

7.1 Konfigurationsoptionen

Die Konfigurationsdatei von Jester ist `jester.cfg`. Diese befindet sich im Installationsverzeichnis. Wichtig ist es, dass die Konfigurationsdatei im Classpath liegt.

- Jester kann auch einen anderen Compiler als den Standard Java Compiler verwenden. Um dies zu tun muss `compilationCommand` auf den gewollten Compiler gesetzt werden. Als Standard ist `compilationCommand=javac` gesetzt.
- Falls der Compiler eine andere Dateierweiterung als `.java` verwendet, muss `sourceFileExtension` auf diese gesetzt werden.
- Standardmäßig wird der Testdialog sofort nach der Ausführung geschlossen. Falls dieser Dialog geöffnet bleiben soll, muss `closeUIOnFinish=false` gesetzt sein.
- Man kann Jester so konfigurieren, dass die Ergebnisse unter der Ausführung anstatt erst am Ende ausgegeben werden. Außerdem wird jede Änderung, die Jester durchgeführt hat, für jede Datei ausgegeben. Dies kann hilfreich sein, falls die Ausführung von Jester nicht beendet wird. Um diese Option zu nutzen, muss `shouldReportEagerly=true` gesetzt werden.

[Moo04b]

7.2 Konfigurierbare Mutationen

Die Mutationen die Jester auf den Source Code ausführt sind konfigurierbar, durch das Editieren der Datei `mutations.cfg`, welche sich im Installationsverzeichnis befindet, wobei sich die Konfigurationsdatei im Classpath befinden muss. Die Zeilen müssen folgendes Format aufweisen:

- `%if(%if(true ||%if (%if (true ||`
  Hier werden die Zeichenfolgen "`if(`" mit "`if(true ||`" und "`if (`" mit "`if (true ||`" getauscht
- `%true%false%false%true`
  Hier werden "true" mit "false" und "false" mit "true" getauscht

Das Zeichen % wird als Trennsymbol verwendet.
[Moo04b]

7.3 Ignore Liste

Jester kann so konfiguriert werden, um bestimmte Teile des Source Codes zu ignorieren, durch Verwenden der Datei `ignorelist.cfg`. Die Datei enthält Zeilen, die festlegen welche Regionen im Source Code ignoriert werden sollen, d.h. Jester versucht hier nicht Mutationen durchzuführen. Der erste String ist der Beginn und der zweite das Ende des Ignore Blocks. Getrennt werden diese Strings mit dem Trennsymbol %. Um zum Beispiel Kommentare in Java zu ignorieren, enthält die Datei `ignorelist.cfg`:

```
%/*%*/
%//%\n
```

Standardmässig enthält die Konfigurationsdatei die Zeile `%//stopJesting%//resumeJesting` damit man im Source Code explizit bestimmte Blöcke vorm Testen ausschließt. Dies kann hilfreich sein, falls bestimmte Stellen im Source Code Jester Probleme bereiten. Um einen Teil des Codes auszuschließen muss nur vor dem Block `//stopJesting` und am Ende des Blockes `//resumeJesting` als Kommentar eingefügt werden.
[Moo04b]

7.4 Jester von ANT aus

Folgendes Beispiel um Jester von ANT aus zu starten. Das Skript kopiert die Sourcen in ein Verzeichnis, wo Jester arbeiten kann.

```
<target name="jester" depends="">
  <delete dir="${modifiedsrc}"/>
  <copy todir="${modifiedsrc}">
    <fileset dir="${src}">
      <exclude name="**/*Test.java"/>
      <exclude name="**/AllTests.java"/>
    </fileset>
  </copy>
  <java classpath="${classpath}" classname="jester.TestTester">
    <arg value="-cp=${classpath}"/>
    <arg value="myproject.AllTests"/>
    <arg value="${modifiedsrc}/myproject"/>
  </java>
</target>
```

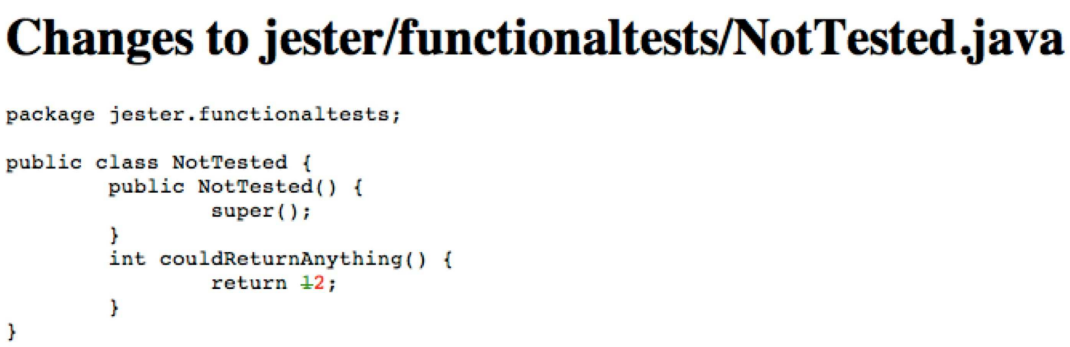
[Moo04b]

8 Anwendungsbeispiel

Wir haben die bei Jester als Beispiel mitgelieferten Tests durchgespielt um einen Einblick in den Arbeitsablauf mit Jester zu erhalten. Die zum durchfüren des ersten Tests auf einem UNIX Betriebssystem notwendigen Schritte sind:

```
sh-2.05b$ export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/
Versions/1.4.2/Home
sh-2.05b$ export PATH=$PATH:$JAVA_HOME/bin
sh-2.05b$ export CLASSPATH=$CLASSPATH:jester.jar:lib/junit.jar:lib/
mmmockobjects.jar:.
sh-2.05b$ javac jester/functionaltests/*.java
sh-2.05b$ java jester.TestTester jester.functionaltests.TestAll jester/
functionaltests
Use classpath: :jester.jar:lib/junit.jar:lib/mmmockobjects.jar:.
For File jester/functionaltests/NotTested.java: 1 mutations survived out
 of 1 changes. Score = 0
jester/functionaltests/NotTested.java - changed source on line 8 (char
 index=139) from 1 to 2
dReturnAnything() {
                return ?1;
        }
}

For File jester/functionaltests/TestAll.java: 0 mutations survived out
 of 0 changes. Score = -1

For File jester/functionaltests/VeryTested.java: 0 mutations survived
 out of 1 changes. Score = 100

For File jester/functionaltests/VeryTestedTest.java: 0 mutations survived
 out of 1 changes. Score = 100

1 mutations survived out of 3 changes. Score = 67
took 0 minutes
sh-2.05b$ python makeAllChangesFiles.py jesterReport.xml
sh-2.05b$ python makeWebView.py -s -z -p
```

Mit diesen Befehlen wurde das Beispiel einem Mutationsttest unterzogen und ein HTML-Report des geänderten Source erstellt. Aus der Ausgabe ist erkennbar, dass in `TestAll.java` keine Änderungen vorgenommen wurden. In `NotTested.java` wurde eine Mutation von den Testfällen nicht erkannt, was auf einen schlecht entworfenen Test hinweist (im Falle dieses Beispiels gar wurde gar nichts getestet).In `VeryTested.java` und dem dazugehörigen Testfall `VeryTestedTest.java` wurde jeweils eine Änderung vorgenommen, welche aber durch Testfälle aufgedeckt wurde.



Changes to jester/functionaltests/NotTested.java

```
package jester.functionaltests;

public class NotTested {
        public NotTested() {
                super();
        }
        int couldReturnAnything() {
                return 12;
        }
}
```

HTML-Report der vorgenommenen Änderung zu der nicht entdeckten Mutation

Zum Ausführen des zweiten Testes kann man analog vorgehen. Man muss nur überall `functionaltests` durch `acceptancetests` ersetzen. Dieser Durchlauf von Jester benötigt schon markant mehr Zeit. Man kann dadurch schon erahnen welches Ausmaß die benötigte Rechenzeit zum kompletten Testen größerer Projekte in annimmt. Beim zweiten Testbeispiel sind schon viel mehr Mutanten erfolgreich durch die Testfälle gelangt. Die ausgegeben Auflistung zeigt auch wie Aufwendig die Analyse schon bei so kleinen Beispielen ist und lässt schlimmers fr umfangreichere Projekte erahnen.

Changes to jester\acceptancetests\Untested.java

```
package jester.acceptancetests;

public class Untested {
public Untested() {
        super();
}
public int canDecrementOrWhatever(int aNum){
        return --aNum;
}
public int canIncrementOrWhatever(int aNum){
        return ++aNum;
}
public boolean canReturnAnyBooleanFalse(){
        return falsetrue;
}
public boolean canReturnAnyBooleanTrue(){
        return truefalse;
}
public int canReturnAnyInt(){
        return 12;
}
public int canTakeAnyBranchFor(boolean condition){
        if(if(true ||if(false &&condition){
                return 12;
        }else{
                return 01;
        }
}
public boolean couldBeEqual(){
        return "hi"!=="hello";
}
public boolean couldBeNotEqual(){
        String a = "hi";
        return a==!=a;
}
}
```

Beispiel für einen HTML-Report mit mehreren vorgenommenen Änderung zu nicht entdeckten Mutationen

Am Rande sei noch vermerkt, dass auf dem von uns zu Testzwecken verwendeten UNIX System die aktuelle Jester Version (1.35) nicht tat was sie sollte und völlig irreführende Fehlermeldungen von sich gab. Da es mit gleicher Version unter Windows Problemlos funktionierte haben wir sehr viel Zeit aufgewendet um auch unter UNIX zu einem Ergebnis zu kommen. Nach vielen erfolglosen Versuchen probierten wir es schließlich mit einer älteren Jester Version (1.32). Erstaunlicherweise funktionierte es mit dieser ohne Änderungen auf Anhieb und ohne Probleme.

Literatur

[Ada03] Konstantinos Adamopoulos. *An Overview of MutationTesting*, 2003. http://www.brunel.ac.uk/ cspgkva/MutationTestingWeb/%20papers/K_Adamopoulos_MTOverview_Nov2003.ppt [November 2004].

[Inc04] Cunningham & Cunningham Inc. *Mutation Testing*, 2004. http://c2.com/cgi/wiki?MutationTesting [November 2004].

[MN04] Ivan Moore and John Nolan. *Test Testing*, 2004. http://xpday2.xpday.org/testtesting.pptt [November 2004].

[Moo03] Ivan Moore. *Jester - the JUnit test tester.*, 2003. http://jester.sourceforge.net/ [November 2004].

[Moo04a] Ivan Moore. Jester - a JUnit test tester. Technical report, Connextra Ltd., 2004. http://www.agilealliance.org/articles/articles/Jester.pdf [November 2004].

[Moo04b] Ivan Moore. *README.html, Jester - the JUnit test tester.*, 2004. http://prdownloads.sourceforge.net/jester/jester135.zip?download [November 2004].