Johannes Kepler Universität Linz

Institut für Systemsoftware

KV Testen von Softwaresystemen
Java Test Tools

Leitung: Dr. Christoph Steindl

Jürgen ERHART
Thomas WEBERNDORFER

Linz, 17. November 2004

Überblick

In diesem Paper werden vier Java Test Tools näher betrachtet. Dabei wird für jedes Tool ein kurzer Überblick gegeben, wofür man es einsetzen kann, und warum man es überhaupt einsetzen soll. Es folgen kurze Installationshinweise und eine Übersicht über die wichtigsten Funktionen. Abschließend werden nochmals die wichtigsten Punkte zusammengefasst sowie die Vor- und Nachteile, die dieses Tool mit sich bringt, aufgezeigt.

Aufgrund der Längenbegrenzung für dieses Paper ist es nicht möglich, alle für Java vorhandenen Test Tools vorzustellen. Sollte jedoch das Interesse an Java Test Tools durch dieses Paper nicht abgedeckt werden, bzw. noch mehr gesteigert werden, dann findet man im Internet unter folgendem Link weitere Test Tool, die bestimmt eines Blickes wert sind.

http://www.aptest.com/resources.html#app-jsource

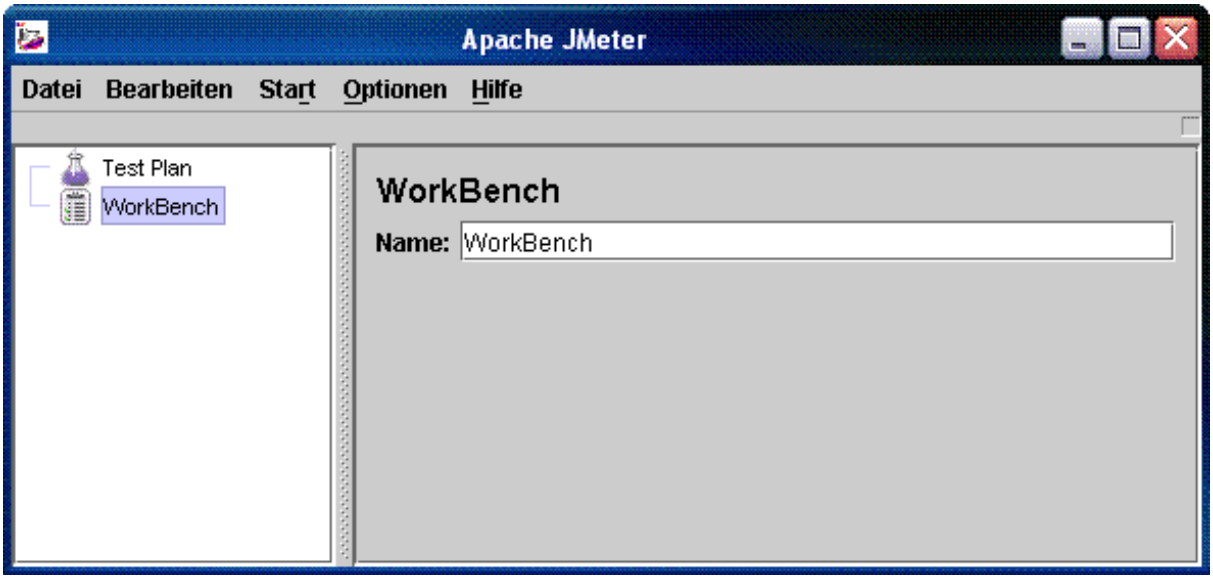
1 JMeter

1.1 Überblick

Apache JMeter ist ein Programm des Apache Jakarta Projekts. Das Programm wurde entwickelt um Belastungstests, Funktionstests und Performancetests durchzuführen. Ursprünglich wurde es zum Testen von Webanwendungen geschaffen. Mittlerweile wurde es mehrfach erweitert und kann nun auch zum Testen von FTP, Java Servlets, Java Objects, SOAP/XML-RPC und JDBC eingesetzt werden. SSL und Remotesteuerung werden ebenfalls unterstützt. Da dieses Programm als Open-Source-Projekt entwickelt wird, besteht natürlich jederzeit die Möglichkeit weitere Funktionen selbst einzubauen.

1.2 Installation

JMeter ist Freeware und kann daher gratis von http://jakarta.apache.org/jmeter/index.html heruntergeladen werden. Zur Zeit (12.11.04) sind die Binarys und die Sourcen der Version 2.0.1 erhältlich. Die Zip-Version ist knapp 7 MB groß und setzt voraus, dass JAVA JDK1.4 bereits installiert ist. Bei älteren Versionen müssen noch zusätzliche Teile nachinstalliert werden, wenn SSL benützt werden soll. Um sicherzustellen das JMeter sofort läuft, sollte man noch die Umgebungsvariable JAVA_HOME setzen. Danach muss man nur noch die entsprechende Batchdatei oder das passende Shellscript starten.

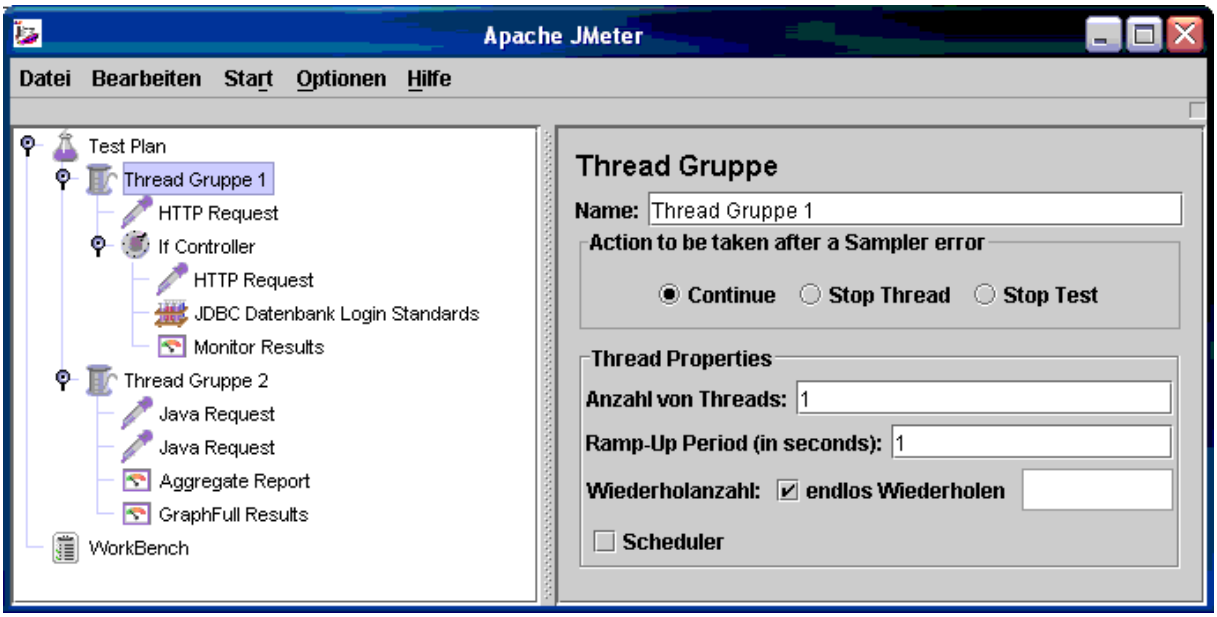


Grafik 1.2, Screenshot nach Start

1.3 Grundlagen

Da JMeter für Belastungstests entwickelt wurde, können mehrere Benutzer simuliert werden, die verschiedene Funktionen testen. Das Symbol „Test Plan“ stellt die Wurzel eines Baumes dar und kann durch „Thread Gruppen“-Knoten erweitert werden. Jede „Thread Gruppe“ kann einen eigenen Test durchführen, der von mehreren Threads aus der „Thread Gruppe“ zeitlich versetzt oder zeitgleich ausgeführt wird. An diese „Thread Gruppen“-Knoten werden die einzelnen Tests wiederum als Knoten angehängt.

Durch auswählen eines Knotens kann man seine Parameter in der rechten Hälfte der Oberfläche einstellen. In Falle einer „Thread Gruppe“ kann man den Namen der Gruppe, die Reaktion auf einen Fehler, die Anzahl der Threads, die Zeit der Startrampe und die Anzahl der Wiederholungen einstellen. Legt man beispielsweise 2 Threads mit 10 Wiederholungen und eine Startrampe von 2 Sekunden an, werden die Tests die an diesem Knoten hängen 20 mal ausgeführt, wobei die 2 Threads jeweils nach einer Sekunde starten, damit alle Threads in zwei Sekunden verfügbar sind.



Grafik 1.3, Der Screenshot zeigt einen Testplan und die Einstellungen für eine „Thread Gruppe“

Nun müssen auch noch Tests an die Knoten angehängt werden. Dazu gibt es eine Reihe von vordefinierten Elementen mit denen man unterschiedliche Funktionen durchführen kann. Zusätzlich gibt es noch Elemente die Daten erfassen und so Auswertungen möglich machen. Auch diese werden als Knoten an „Thread Gruppen“ angehängt.
Die Anzahl der möglichen Knotenelemente ist mittlerweile sehr groß, da immer wieder neue hinzugefügt werden und, dank Open-Source, die Möglichkeit besteht, auch selbst solche Knotenelemente zu schreiben.

Grundsätzlich kann man die Knotenelemente in sechs Gruppen einteilen:

Sampler	Diese Elemente führen Requests zu Servern aus. Es gibt Sampler für FTP, HTTP, Soap/XML, BSF, BeanShell, JDBC, LDAP, TCP, usw.
Listener	Damit kann man die Ergebnisse der Testläufe ermitteln. Auch hier gibt es wieder eine große Auswahl mit verschiedenen Darstellungen der Daten.
Timer	Mit Timern kann man Wartezeiten zwischen Requests definieren. Es gibt Timer für konstanten Durchsatz, konstante Zeit, Gaus- und Gleichverteilung. Durch Timer kann man die Testläufe „realistischer“ gestalten.
Logic Controller	Damit kann man unterschiedliche Abläufe in Testläufe realisieren. Es gibt Verzweigungen, Schleifen, zufällige Verzweigungen, usw.
Configuration Element	Mit diesen Elementen kann man zusätzlich Informationen zu den Elementen vom Typ Sampler hinzufügen, um kompliziertere Requests zu formulieren.
Assertion	Durch Assertions kann man Ergebnisse mit Sollwerten vergleichen.

Tabelle 1.3, Übersicht der Knotenelemente

Aus diesen Elementen lassen sich beliebig komplexe Bäume aufbauen. Ausgeführt wird dieser Testplan durch Aufrufen von „Start“ im Menü „Start“. Danach kann man beobachten wie sich die Listeners mit Daten füllen.

Wenn man sehr große Testfälle realisieren will, ist es ratsam den Testfall automatisch generieren zu lassen. Dazu muss JMeter als Proxyserver verwendet werden. Es muss der Browser so umgestellt werden, dass er nur über den Proxyserver auf den zu testenden Server zugreifen kann. JMeter kann so alle Requests aufzeichnen. Meist ist noch eine Nachbearbeitung des aufgezeichneten Anwendungsfalls notwendig.
Durch sehr große Tests ergibt sich leider auch ein großer Verbrauch an Speicher und Rechenleistung. Will man nun auch noch mehrere Zugriffe simulieren, besteht die Gefahr, dass der Computer der JMeter ausführt, der Belastung nicht mehr standhalten kann und Requests verzögert versendet werden, bevor die gewünschte Belastung am Server auftritt. Die so gewonnenen Ergebnisse sind daher leider wertlos, da man ja den Server belasten will und nicht den eigenen Computer. Um dieses Problem in den Griff zu bekommen, kann man JMeter auf mehrere Computer „aufteilen“. Dabei wird die GUI und der Teil, welcher Requests sendet (JMeter-Server), getrennt. Der große Vorteil besteht nun darin, dass man mit einer GUI mehrere JMeter-Server gleichzeitig steuern kann.

Durch Einrichten von zusätzlichen JMeter-Servern, kann man somit genügend Rechenleistung aufbringen, um jeden Server beliebig belasten zu können. Die Kommunikation zwischen den Teilen erfolgt über den JVA RMI-Mechanismus.

JMeter bietet auch noch einen Textmodus, um etwas Speicher und Rechenleistung zu sparen. Diese Einsparungen werden jedoch nur in Grenzfällen helfen. Man kann diesen Modus aber benutzten, um JMeter durch andere Tools zu automatisieren. Die Ausgabe wird dann in Dateien gespeichert.

1.4 Folgerungen

JMeter eignet sich sehr gut um Belastungstests durchzuführen. Es ist sehr einfach zu verwenden. Man sollte sich aber genügend Zeit nehmen, um die Testfälle und gegebenenfalls mehrere JMeter-Server einzurichten, um möglichst gute und aussagekräftige Ergebnisse zu erhalten. Da es sich um ein Open-Source-Projekt handelt, besteht natürlich immer die Möglichkeit von Fehlfunktionen für die man das Risiko selbst übernehmen muss. Dafür kann man aber eigene Teile programmieren und zu JMeter hinzufügen. Durch dieses modulare Konzept ist JMeter sehr gut erweiterbar und flexibel. All diese Eigenschaften machen JMeter zu einen guten Konkurrenten gegenüber den oft sehr teuren kommerziellen Produkten. Daher sollte man sich überlegen, ob man nicht mit JMeter das Auslagen findet, bevor man viel Geld investiert.

2 JStyle

2.1 Überblick

JStyle ist ein Tool für automatischen Codereview. Es wurde von Man Machine Systems entwickelt. Mit diesem Tool kann man Sourcecode analysieren und so Fehler und Schwachstellen vor der Kompilierung finden. Diese Fehler sind aber nicht nur Fehler im üblichen Sinn, sondern es werden auch noch eine Reihe anderer Schwachstellen erkannt:

- Zuwenig Kommentare
- Falsche Namen, daher Verstoß gegen Konventionen
- Kodestücke, die Wiederverwendung unmöglich machen
- Codestücke, die nicht effizient sind
- Codestücke, die nie zur Ausführung kommen können

Die Schwachstellen werden mit Hilfe von Metriken berechnet. Es wird dazu der Code analysiert und sobald die berechneten Werte definierte Grenzwerte überschreiten, der eingegebene Code als möglicherweise fehlerhaft markiert. Eine andere Variante vergleicht den Quelltext mit Strukturmustern, welche ungeschickte, fehlerhafte oder sehr fehleranfällige Codestücke beschreiben. Diese Codestücke sollten dann von Entwicklern noch einmal begutachtet werden.

JStyle erfüllt natürlich diese Fähigkeiten und stellt noch einiges mehr zur Verfügung:

- Hohe Performanz für große Projekte
- Code Reviews und OO Metriken
- Unterstützt über 100 Stilrichtlinien
- Hinzufügen von Regeln und Richtlinien mit JMScript und VBScript
- Filtern von Kommentaren und Berichten
- Kommandozeilensteuerung zum Automatisieren durch zusätzliche Tools
- Einen guten Berichtsgenerator
- Kontrollflussgrafen

2.2 Installation

Das Programm ist auf der Homepage von Man Machine Systems http://www.mmsindia.com/default.htm verfügbar. Leider ist dieses Programm nicht Freeware. Es gibt zwei Lizenzen: (Stand: 12.11.04)

- "Single User Node-Locked License" um 595 $
- "Single User Floating License" um 895 $

Zusätzlich steht noch eine Trialversion zum Downloaden in der Große von 20MB zur Verfügung. Damit hat man 16 Tage Zeit, sich für einen Kauf zu entscheiden. Damit das Produkt überhaupt startet, benötigt man Windows 95/98/NT4.0/2000/XP, 128MB Ram-Speicher, 50MB freie Festplatte und Java ab der Version 1.1

2.3 Die wichtigsten Funktionen

JStyle bietet eine große Menge von Funktionen an. Daher werden hier nur wenige besonders praktische und wichtige besprochen. Die im Folgenden beschriebenen, von JStyle generierten, Berichte und Statistiken können auch als HTML- oder Textdokument ausgegeben werden.

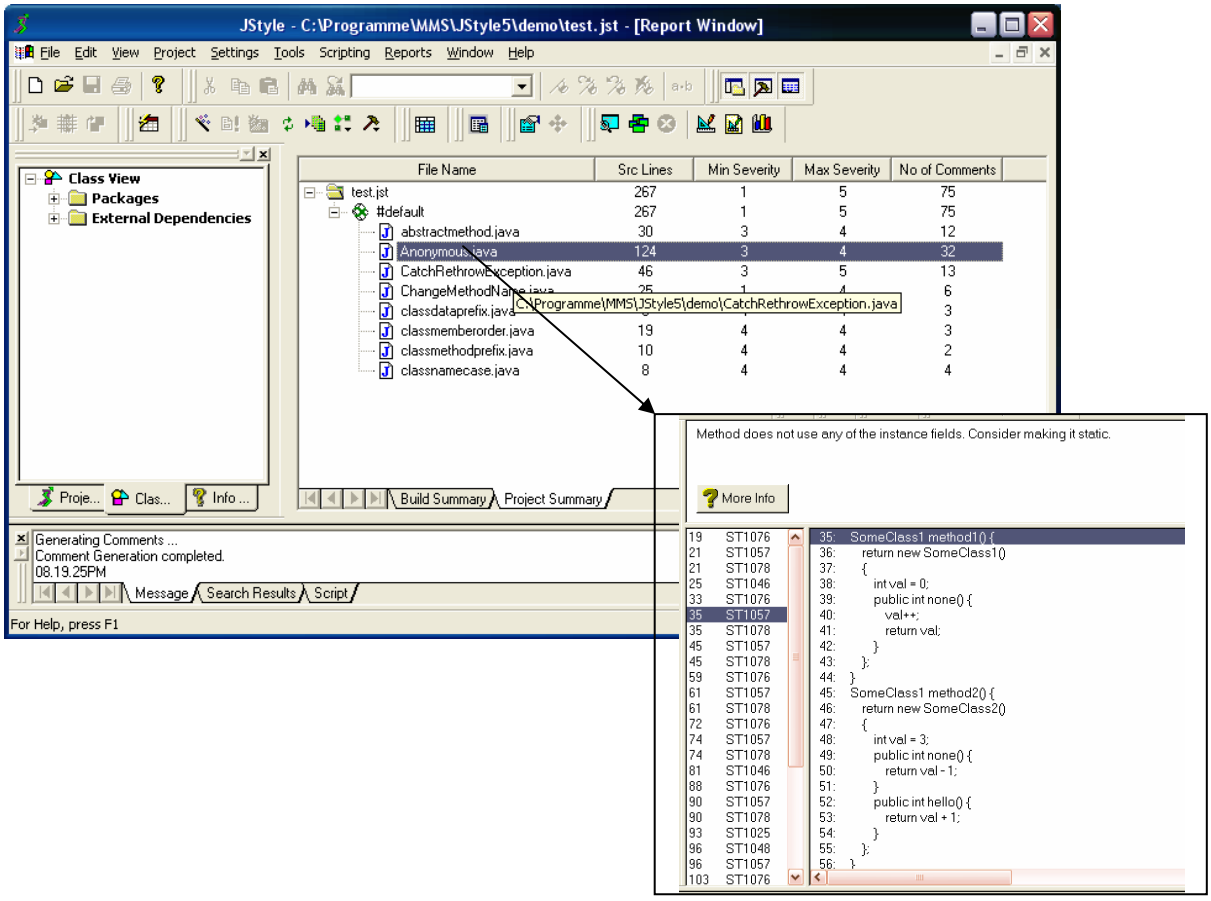
2.3.1 Anlegen eines neuen Projektes:

Um überhaupt Java-Dateien importieren zu können, muss zuerst ein Projekt angelegt werden. Dazu muss man unter „Project" den Eintrag „new" auswählen. Nachdem die Datei festgelegt wurde, in der das Projekt gespeichert werden soll, erscheint ein Dialogfeld zum Einfügen der Java-Dateien. Das Projekt ist nun erstellt und kann gespeichert werden. Änderungen, welche die Java-Dateien betreffen, werden immer sofort in die betroffenen Dateien übernommen.

2.3.2 Generate Comment:

Als nächstes muss die Funktion „Generate Comment“ ausgeführt werden. Die Quelldateien werden dabei übersetzt und zwei Zusammenfassungen gebildet:

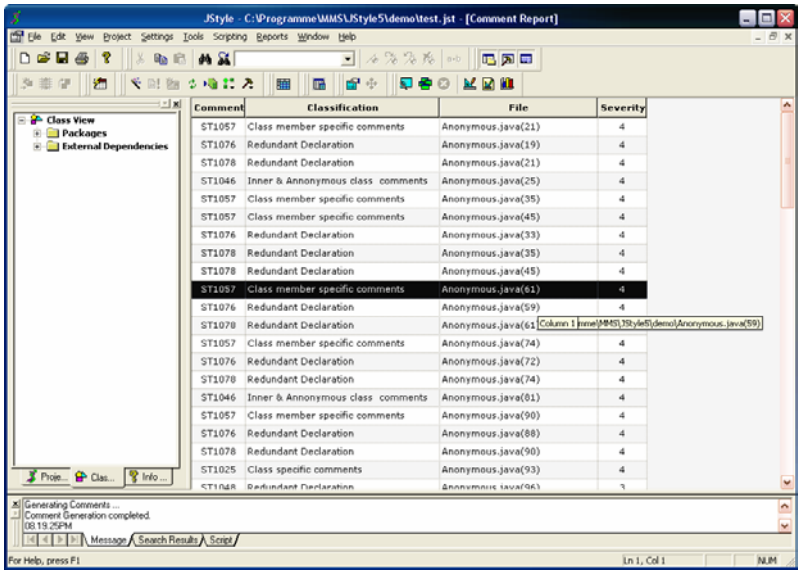
- Build Summary: Darin werden alle Quelldateien angezeigt und dazugehörend die Anzahl der Klassen, Anzahl der Zeilen und die Anzahl der Fehler. Durch klicken auf eine Klasse wird die Klasse selbst angezeigt.
- Project Summary: Hier werden ebenfalls alle Quelldateien angezeigt mit den zugehörigen Informationen wie Anzahl der Zeilen, der minimalen und maximalen Schwierigkeit und die Anzahl der Kommentare. Durch klicken auf eine Klasse werden die gefundenen Fehler angezeigt. Der einzuhaltende Stiel wird auch bewertet.



Grafik 2.3.2, Screenshot nach Auswahl einer Quelldatei

2.3.3 View Comment Sheet

Um eine Übersicht über alle gefundenen Kritiken zu haben, kann die Funktion „View Comment Sheet“ aufgerufen werden. Dabei werden alle aufgetretenen Meldungen aufgelistet und dazu die Anzahl angezeigt. Damit kann man sich sehr schnell einen Überblick über das gesamte Projekt verschaffen.



Grafik 2.3.3, Screenshot zeigt Comment Sheet

2.3.4 View Metrix Sheet

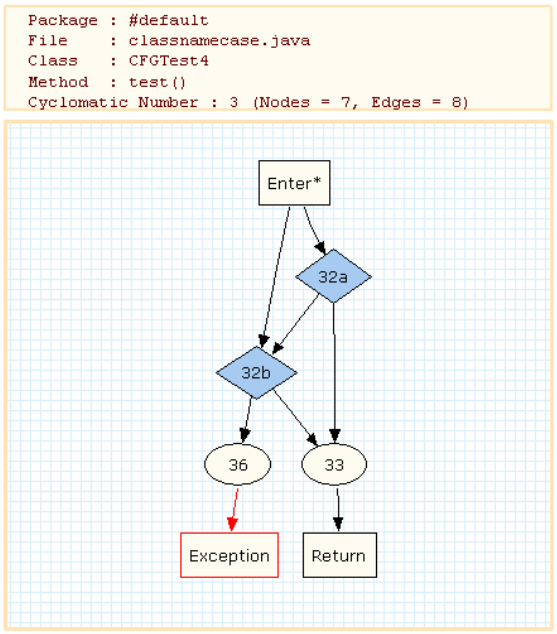
Diese Funktion dient dazu, viele Metriken die das gesamte Projekt betreffen auf einen Blick zu erhalten. Es werden die Anzahl der verschiedenen Deklarationen und Aufrufe, durchschnittliche Vererbungstiefe, die Anzahl der verschiedenen Methodentypen, Anzahl der Zeilen, Anzahl der Kommentare, Kommentarhäufigkeit, Schwierigkeit, Aufwand, Fehlerwahrscheinlichkeit und vieles mehr angezeigt. Mit Hilfe dieser Werte kann man Klassen identifizieren, die sich von den anderen abheben und deswegen noch einmal durchgesehen werden sollten.

2.3.5 Beautify

Wenn gerade eine Klasse angezeigt wird, steht die Funktion „Beautify“ zur Verfügung. Damit kann ein Quellcode in eine zuvor definierte Form gebracht werden. Dabei werden Abstände, Tabulatoren, Zeilenumbrüche, Reihungen von Deklarationen und Methoden korrigiert, sodass sie der eingestellten Vorgabe entsprechen. Die Änderungen werden dabei sofort in der Quelldatei gespeichert. Die von uns getesteten Standardeinstellungen erzeugen bereits eine „schöne“ Quelldatei, nur Zeilenumbrüche werden nicht berücksichtigt. Dadurch werden mit „Beautify“ bearbeitete Java-Dateien beim Ausdrucken etwas verunstaltet.

2.3.6 Control Flow Graph

Wenn besonders komplexe Methoden vorliegen hilft die Funktion „Control Flow Graph“. Diese Funktion steht im Kontextmenü zur Verfügung wenn sich der Cursor innerhalb einer Funktion befindet. Mit diesem Befehl wird der Kontrollfluss in einer übersichtlichen Grafik dargestellt.



Grafik 2.3.6, Der Screenshot zeigt ein Kontrollflussdiagramm

2.3.7 Generate Chart

Falls man die zuvor erhobenen Werte auch grafisch darstellen möchte, kann man mit dieser Funktion bequem verschiedene bunte Diagramme erzeugen. Neue Erkenntnisse werden hier zwar nicht generiert, aber für Präsentationen über den aktuellen Projektstatus könnte man diese Diagramme gut gebrauchen.

2.4 Folgerungen

Viele dieser Funktionen wie Quellcode formatieren und Kontrollgrafen zeichnen sind sehr nützlich. Die generierten Statistiken dagegen sind eher Standard und könnten mit vielen anderen Programmen genauso generiert werden. Damit diese errechneten Ergebnisse einem gesamten Entwicklerteam zur Verfügung stehen, müsste man die Ergebnisse in einem HTML-Dokument erstellen. Diese Tätigkeit könnte aber auch von anderen Programmen wie z.B. Maven genauso gut, aber wesentlich billiger, realisiert werden. Die große Stärke dieses Programms liegt aber in der Fähigkeit, Fehler, schlechten Programmierstiel und sinnlose Codestücke zu entdecken und kann so zu einem wesentlich verbesserten Projekt beitragen.

3 Jemmy

3.1 Überblick

Jemmy ist eine Javabibliothek mit deren Hilfe man automatisierte Tests für grafische Oberflächen schreiben kann. Die Bibliothek beinhaltet viele Methoden, um alle Interaktionen, die mit einer grafischen Oberflache möglich sind, auszulösen. Die Methoden wurden für Swing und AWT entworfen. Man kann entweder JemmyTest verwenden oder selbst Testfälle schreiben und diese in verschiedene Testprogramme einbauen. So kann das API von Jemmy verwendet werden, um JUnit Testfälle zu programmieren.

Eine weitere Einsatzmöglichkeit besteht darin, Vorgänge als Demo ablaufen zu lassen und zusätzlich mit Kommentaren zu versehen. Jemmy wurde als NetBean entwickelt und kann daher auch mit NetBeans IDE verwendet werden.

3.2 Installation

Jemmy ist ein Freewareprodukt und kann von http://jemmy.netbeans.org/ gratis heruntergeladen werden. Es werden zwei Versionen angeboten:

- Jemmy.jar: mit der Größe von 1,6MB
- Jemmy.zip: Darin sind alle Quelldateien, eine Dokumentation und Beispiele enthalten. Die Größe beträgt 3,5MB

3.3 Funktionsweise

Zuerst muss sichergestellt werden, dass der Testfall und das zu testende Programm in der selben JVM laufen. Nur so kann Jemmy als ersten Schritt das zu testende Fenster finden. Dazu wird die Methode „java.awt.Frame.getFrames()“ aufgerufen, um alle Fenster zu erhalten. Danach werden die Fenster, die von „Window.getOwnedWindows()“ zurückgegeben werden, nach dem gewünschten Fenster durchsucht.

Wurde das passende Fenster gefunden, müssen die mit „java.awt.Container.getComponents()“ erhaltenen Komponenten durchsucht werden, bis das zu testende Objekt gefunden wurde.

Normalerweise werden GUI-Änderungen über eine Liste gesteuert. Diese Liste ist auch der Ansatzpunkt von Jemmy. In einer eigenen Klasse „org.netbeans.jemmy.QueueTool“ sind alle notwendigen Operationen für diese Liste implementiert.

Bei Testfällen wird meist zuerst gewartet bis diese Liste leer ist, da dann alle Ereignisse ausgeführt worden sind und somit das Fenster fertig aufgebaut wurde.

Dann werden die Aktionen in die Liste eingehängt. Die grafische Oberfläche kann dabei nicht entscheiden, ob die Einträge in der Liste vom Benutzer oder von Jemmy erstellt wurden. Somit lassen sich Benutzereingaben zum Testen simulieren. Als angenehmer Nebeneffekt kann durch das Verwenden der Liste auch noch Threadsicherheit garantiert werden.

3.4 Testfälle mit Jemmy

3.4.1 Finden von Frames

Damit Jemmy in einem Testfall verwendet werden kann, muss das Interface "org.netbeans.jemmy.Scenario" implementiert werden. Dadurch ist es notwendig die Methode „public int runIt(Object param)" zu erstellen. In diese Methode werden die auszuführenden Aktionen eingebettet. Ein einfaches Beispiel soll dies verdeutlichen:

```
import org.netbeans.jemmy.*;
import org.netbeans.jemmy.operators.*;

public class WaitWindowSample implements Scenario {
    public int runIt(Object param) {
                try {
                    // Anwendung starten
                    new ClassReference("org.netbeans.jemmy.explorer.GUIBrowser")
                        .startApplication();
                    // Warten bis der Frame fertig geladen wurde
                    new JFrameOperator("GUI Browser");
                } catch(Exception e) {
                    e.printStackTrace();
                    return(1);
                }
                return(0);
    }
    public static void main(String[] argv) {
                String[] params = {"WaitWindowSample"};
                org.netbeans.jemmy.Test.main(params);
    }
}
```

Beispiel 3.4.1, Quelle: http://jemmy.netbeans.org/samples/WaitWindowSample.java

Mit new ClassReference(..) wird ein Frame erzeugt. Das Objekt „JFrameOperator" wartet bis alle Ereignisse abgearbeitet sind. Danach terminiert das Programm. Als Ausgabe werden alle verfügbaren Parameter des gefunden Fensters ausgegeben.

3.4.2 Finden von Komponenten

Nun müssen aber auch noch die Komponenten in einem Frame gefunden werden. Danach können Ereignisse auf diesen Komponenten ausgeführt werden. Zum Suchen der Komponenten werden einfach alle geschachtelten Komponenten in einem Frame abgearbeitet. Als Suchmuster können viele verschiedene Parameter übergeben werden. Die einfachste Variante ist das Suchen nach Texten. Für Schaltflächen und Fenster ist dies völlig ausreichend. Die Suche wird von Operatoren-Objekte durchgeführt. Die Basisklasse aller Such-Operatoren-Objekte ist „org.netbeans.jemmy.operators.ComponentOperator“. Die abgeleiteten Such-Operatoren haben je nach Komponente verschiedene Parameter zum Suchen der Komponenten.

Diese Parameter sind durch die Internationalisierung je nach Spracheinstellung des verwendeten Computers oft verschieden. Damit Jemmy flexibel genug ist, um mit der Java Internationalisierung mithalten zu können, wurde ein ähnliches System entwickelt. Mit der Methode „org.netbeans.jemmy.Bundle.loadFromFile(String)“ können Zeichenketten aus Dateien gelesen werden. Dieser Mechanismus ist auch für nachträgliche Änderungen in der grafischen Oberfläche sehr nützlich, da man damit sehr einfach und schnell die Testfälle anpassen kann.

Dieses Beispiel zeigt die Verwendung von Stringressourcen in eigenen Dateien:

Ausschnitt aus public int runIt(Object param)

```
 // lade Bündel
Bundle bundle = new Bundle();
bundle.loadFromFile(System.getProperty("user.dir") +
System.getProperty("file.separator") +
"resourcesample.txt");
new ClassReference(bundle.getResource("guibrowser.main_class")).
        startApplication();
// verwende Bündel
JFrameOperator mainFrame =
new JFrameOperator(bundle.getResource("guibrowser.main_window"));
```

Beispiel 3.4.2a, Quelle: http://jemmy.netbeans.org/samples/ResourceSample.java

Eintrag aus der Ressourcendatei:

```
guibrowser.main_class=org.netbeans.jemmy.explorer.GUIBrowser
```

Beispiel 3.4.2b, Quelle: http://jemmy.netbeans.org/samples/resourcesample.txt

3.4.3 Benutzen von Komponenten

Wurde eine Komponente mit einem Operator-Objekt gefunden, so stellt dieses Objekt auch gleichzeitig die Schnittstelle zur Komponente dar. Dadurch können über diese Objekte verschiedene Operationen auf den gefundenen Komponenten ausgeführt werden. Um einen Benutzer realistisch simulieren zu können, muss man natürlich auch alle möglichen Tastatur- und Mauseingaben vortäuschen können. Dazu kann man ein Objekt der Klasse „org.netbeans.jemmy.operators.ComponentOperator“ verwenden. Folgendes Beispiel soll das interagieren mit Komponenten veranschaulichen:

```
// Starten der zu testenden Anwendung
new ClassReference("org.netbeans.jemmy.explorer.GUIBrowser").startApplication();
// Suchen des Hauptfensters und warten bis der Frame geladen ist
JFrameOperator mainFrame = new JFrameOperator("GUI Browser");
// Suchen des Buttons “Reload after” und drücken des Buttons
new JButtonOperator(mainFrame, "Reload after").push();
```

Beispiel 1.4.3a: Quelle: http://jemmy.netbeans.org/samples/FindComponentsSample.java

Auch hier wird wieder das Ergebnis auf der Konsole ausgegeben:

```
Using org.netbeans.jemmy.drivers.DefaultDriverInstaller driver installer
Reproduce user actions through event queue.
Use event dispatching to reproduce user actions
Shortcut test events
Using org.netbeans.jemmy.drivers.DefaultDriverInstaller driver installer
Executed test WaitWindowSample
Test WaitWindowSample has been started
Trace:
Start to wait action "Test WaitWindowSample finished"
Trace:
Start to wait frame "Frame with title "GUI Browser"" opened
Trace:
Frame "Frame with title "GUI Browser"" has been opened in 0 milliseconds
org.netbeans.jemmy.explorer.GUIBrowser[frame0,0,0,800x400,invalid,layout=java.awt.BorderL
ayout,title=GUI
Browser,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.JR
ootPane[,4,30,792x366,invalid,layout=javax.swing.JRootPane$RootLayout,alignmentX=null,alig
nmentY=null,border=,flags=385,maximumSize=,minimumSize=,preferredSize=],rootPaneCheck
ingEnabled=true]
org.netbeans.jemmy.TestCompletedException: Test passed
        at org.netbeans.jemmy.Test.closeDown(Test.java:126)
        at org.netbeans.jemmy.Test.launch(Test.java:353)
        at org.netbeans.jemmy.ActionProducer.launchAction(ActionProducer.java:312)
        at org.netbeans.jemmy.ActionProducer.run(ActionProducer.java:269)
Trace:
"Test WaitWindowSample finished" action has been produced in 907 milliseconds with result
   : 0
```

Beispiel 1.4.3b, Quelle: Ausgabe von Beispiel 1.4.3a

Durch Umleiten der Konsole kann so verglichen werden, ob das gewünschte Ergebnis erreicht wurde.

3.4.4 Verhalten im Fehlerfall

Wenn alle Tests erfolgreich verlaufen, sind die oben gezeigten Ansätze gut durchführbar. Es kann jedoch passieren, dass auf Grund von Fehlern ein Operator-Objekt die gewünschte Interaktion mit einer Komponente nicht ausführen kann. Ein mögliches Szenario für einen solchen Fall wäre ein geöffnetes systemmodales Fenster und ein Operator-Objekt welches auf ein Menü in einem darunter liegenden Fenster zugreifen will, was jedoch nicht möglich ist. In so einen Fall könnte eine gesamte Testsuite blockiert werden. Damit solche Fehler nur eine eingeschränkte Auswirkung haben, kann man eine maximale Wartezeiten für Operator-Objekte vorgeben. Wird diese Wartezeit überschritten, wird der aktuelle Vorgang mit einem Fehler abgebrochen. Damit können andere Testfälle weiterlaufen. Die Wahl dieser Wartezeit muss besonders sorgfältig getroffen werden. Zu lange Wartezeiten würden die Ausführungszeit einer Testsuite extrem verlängern. Eine zu kurze Zeitspanne könnte fälschlicherweise Fehlermeldungen auslösen, da der Aufbau einer grafischen Oberfläche unterschiedlich lange dauert. Der Grund dafür sind die verschiedenen Threads die für das Zeichnen der Oberfläche verantwortlich sind und vom Scheduler des Betriebssystems je nach Situation anders behandelt werden.

Als Alternative zu den Wartezeiten könnte man auch noch die Testfälle in verschiedenen Threads laufen lassen. Diese Variante wäre aber um einiges komplizierter.

Alle anderen Arten von Fehlern können durch eine geworfenen Exception erkannt werden. Diese Excepitons sind von „org.netbeans.jemmy.JemmyException“ abgeleitet. Dadurch kann man das Ergebnis des Testfalles sehr gut auswerten und entsprechend der Testsoftware reagieren.

3.5 Folgerungen

Jemmy ist eine sehr leistungsfähige Bibliothek, die das Testen von grafischen Oberflächen stark vereinfacht. Durch das Tutorial auf der dazugehörigen Homepage können mit nur sehr wenigen Kenntnissen bereits Testfälle generiert werden. Da der Quelltext erhältlich ist, kann man auch leicht Änderungen vornehmen, um z.B. Handlungen aufzuzeichnen und so Makrofunktionen zu implementieren. Der einzige Schwachpunkt liegt in den AWT-Menüs, da diese durch native Funktionen realisiert sind und so die Position der Menüeinträge nicht ermittelbar ist. Durch definieren von Shortcuts für die Menüeinträge kann man aber dieses Problem umgehen.

4 JUnit

4.1 Überblick

4.1.1 Motivation

Das Testen von Software ist ein essenzieller Bestandteil der Softwareentwicklung. Ohne Tests wäre ein Softwareprojekt zum scheitern verurteilt, und wenig Tests bedeuten, dass nur eine geringe Fehleranzahl gefunden werden kann. Und Trotzdem wird das Testen von Programmen während ihrer Entwicklung meist aus Zeitgründen bzw. wegen des Aufwands vernachlässigt. Doch das Testen ist mindestens genauso wichtig wie die Entwicklung selbst. Testet man den Programmcode laufend während der Entwicklung, so wird die spätere, sehr umfangreiche, Fehleranalyse zum Schluss vermindert. Man kann prinzipiell davon ausgehen, dass die Komplexität eines Programms exponentiell mit der Anzahl an Programmzeilen anwächst. Daher ist frühes und ausführliches testen nahezu Unumgänglich.

Ein einfacher Ansatz wäre das Verwenden eines Debuggers, oder die Ausgabe von Debuginformationen. Beide Varianten haben ihre Schwächen. Das Verwenden eines Debuggers ist nicht immer möglich, bzw. bei vielen Threads oft schwierig. Die Ausgabe von Debuginformationen kann bei Schleifen-Konstrukten schnell unübersichtlich werden. Außerdem muss der Entwickler ständig darüber nachdenken, welche Informationen er denn eigentlich in der Ausgabe braucht.

Hier soll JUnit Abhilfe schaffen.

4.1.2 Was ist JUnit

JUnit ist ein Test-Framework, welches dem Entwickler laufende Tests und deren schnelle und einfache Verwendung ermöglichen soll. Für die Tests an sich ist keine Benutzerinteraktion nötig, und Threadsicherheit wird ebenfalls gewährleistet.

Der Entwickler schreibt für jede seiner Methoden eine oder mehrere Testroutinen, den sogenannten Testcases, die in einer von JUnit zur Verfügung gestellten Klasse zusammengefasst werden.

Man geht dabei vom Prinzip des Test Driven Development aus. Das bedeutet, erst testen, dann solange Programmcode verfassen bis der Testfall funktioniert. Daraufhin wieder testen und wiederum Programmcode verfassen. Durch diese Methode wird gewährleistet, dass von Grund auf Fehler vermieden werden, da man immer nur solange Programmcode schreibt, bis der Testfall funktioniert.

Alle Klassen die für einen Test nötig sind, findet man im Paket junit.framework. Die wichtigsten dabei sind:

- TestCase und TestSuite zum Verfassen von eigenen Testfällen
- Assert zum aufstellen von Behauptungen
- Test ist ein Interface welches TestCase und TestSuite vereint
- TestRunner dient zum eigentlichen Ausführen der Tests

4.2 Installation

Die Installation von JUnit gestaltet sich als relativ einfach. Es genügt völlig, das Paket in ein durch die CLASSPATH Variable abgedecktes Verzeichnis, zu entpacken. Alternativ kann man natürlich auch die CLASSPATH Variable entsprechend verändern.

In Entwicklungsumgebungen wie Eclipse (www.eclipse.org) ist JUnit fix integrierbar. Man wählt dazu „Window“ und „Preferences“ aus, doppelklickt auf „Java“ und selektiert „Classpath Variables“. Dort wählt man „New“ aus, und braucht nur noch einen Namen, plus den Pfad zu der Datei „junit.jar“ angeben. Ist dies erledigt und mit „Ok“ bestätigt, kann man JUnit mit jedem Projekt verbinden, indem man die neu erstellte Classpath Variable seinem Projekt hinzufügt.

JUnit ist ein Open Source Projekt, und daher auch nicht mit finanziellen Kosten verbunden.

4.3 Die wichtigsten Funktionen

4.3.1 Testen mit JUnit

Will man nun Software mit JUnit testen lassen, muss man zuerst einen Testfall entwickeln. Dabei schreibt man eine neue Testklasse die von junit.framework.TestCase abgeleitet wird. In dieser Klasse werden dann alle Testfälle als eigene Methoden ausprogrammiert. JUnit führt diese Testfälle dann durch, und gibt entsprechende Meldungen aus, wie sich die Testfälle verhalten haben. Damit klar ist ob ein Testfall korrekt funktioniert hat, muss der Entwickler mittels bestimmter Methodenaufrufe innerhalb des Testfalls angeben, welchen Wert er sich als Ergebnis erwartet. Hat ein Testfall zwar funktioniert, aber die getestete Methode einen anderen Wert als den vom Entwickler erwarteten zurückgegeben, so meldet JUnit einen Failure. Hat eine Methode überhaupt nicht funktioniert, z.B. weil eine Exception aufgetreten ist, dann meldet JUnit einen Error.

Um dies zu verdeutlichen ein kleines Beispiel:

Es soll ein Programm entwickelt werden, mit dessen Hilfe man logische Schaltungen simulieren kann. Es müssen Methoden geschrieben werden, welche die Gatterbausteine AND, OR, NOT, XOR, usw. simulieren. Bis auf die NOT Methode bekommen alle Methoden zwei boolsche Ausdrücke übergeben, und es soll true oder false zurückgegeben werden, je nachdem wie das entsprechende Gatter schalten würde.

Nun beginnt man mit der Testklasse:

Als erstes möchte man die AND Methode testen.

```
import junit.framework.TestCase;

public class TestLogicalSimulator extends TestCase{

public void testAND(){
                assertFalse(LogicalSimulator.logicalAND(false, false));
              }
}
```

Beispiel 4.3.1, eine Testroutine für den LogicalSimulator

Die Methode assertFalse wird von JUnit zur Verfügung gestellt. Der als Parameter Übergebene Methodenaufruf wird benutzt, und der Rückgabewert geprüft. Ist der Rückgabewert false, so war der Test erfolgreich. Wird von der getesteten Methode true zurückgeliefert, schlägt der Test fehl, und es wird ein Failure ausgegeben. Natürlich gibt es auch eine Methode assertTrue. Hat man ein Methode die keinen boolschen, sondern einen anderen Wert zurückgibt, so kann assertEquals benutzt werden. Dieser Methode übergibt man die zu testende Methode, sowie den erwarteten Rückgabewert.

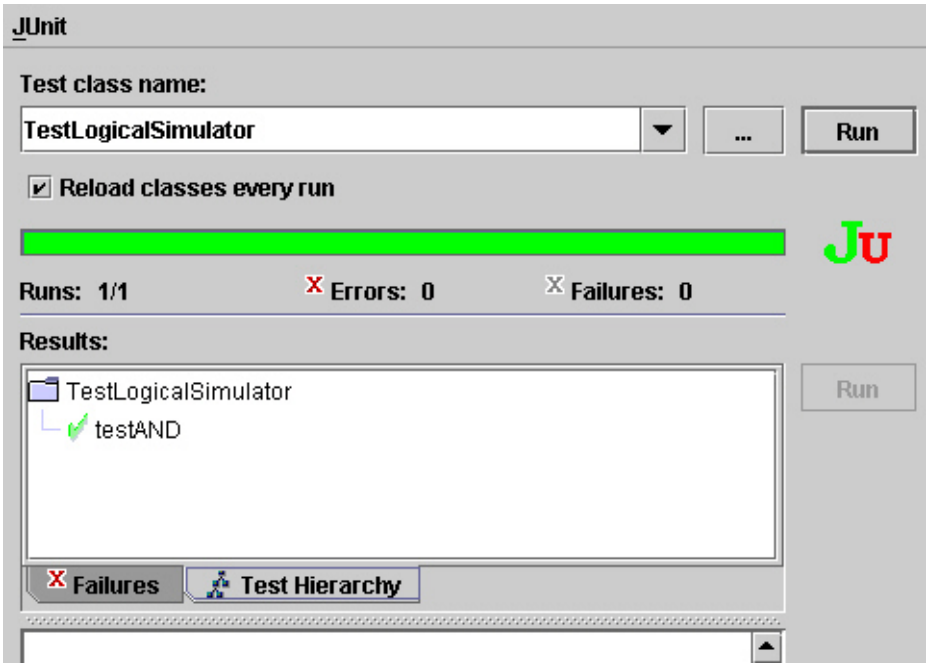
Der Test aus Beispiel 4.3.1 wird sich jedoch nicht kompilieren lassen, da die passende Methode in LogicalSimulator fehlt. Also programmieren wir nun diese:

```
public class LogicalSimulator{

              public static boolean logicalAND(boolean x, boolean y){
                    return false;
              }
}
```

Beispiel 4.3.2, die erste Methode in LogicalSimulator

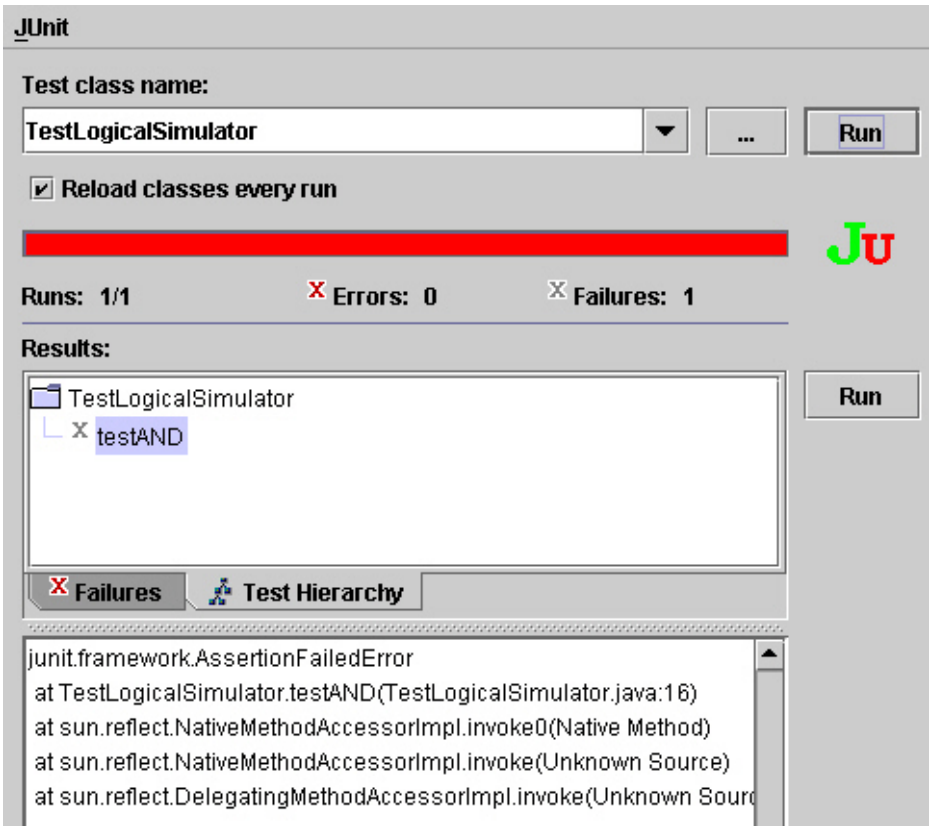
Nun kann man den Testfall laufen lassen, und er wird erfolgreich durchlaufen.



Grafik 4.3.1, Screenshot von JUnit mit erfolgreichem Test

Ein Test alleine reicht jedoch meist nicht, und wie man schon an der logicalAND Implementierung erkennen kann, sind nicht alle möglichen Fälle abgedeckt. Fügt man z.B. folgenden Testfall hinzu:

assertTrue(LogicalSimulator.logicalAND(true, true)); müsste logicalAND eigentlich true zurückliefern. In der jetzigen Form kann das die Methode jedoch nicht. Also muss man wieder Programmieren, bis auch der neue Test funktioniert. Führt man die Entwicklung aller Methoden auf diese Weise durch, ist sichergestellt, dass alle Testfälle abgedeckt sind. Da auch immer alle Tests durchlaufen werden, sieht man sofort ob die Änderung einer Methode Auswirkungen auf eine andere hatte.



Grafik 4.3.2, Screenshot von JUnit mit fehlgeschlagenem Test

Anhand des fehlgeschlagenen Tests sieht man auch schon, wie JUnit einen Failure handhabt. Sämtliche assert-Methoden versuchen die als Parameter übergebene Methode mittels invoke auszuführen. Das Ergebnis wird mit dem erwarteten Ergebnis verglichen. Unterscheiden sie sich, wirft die assert-Methode eine AssertionFailedError Exception, die dazu führt, dass das Framework den Failure mitprotokolliert. Dafür ist die Klasse TestResult zuständig. Sie zählt auch mit wie viele Tests bereits durchgeführt, und wie viele davon Failures oder Errors gemeldet haben. Zu beachten ist, dass durch diese Art des Exceptionhandlings sämtliche, auf den fehlgeschlagenen Test folgende, Asserts nicht mehr ausgeführt werden. Dadurch kann es passieren, dass man trotz des Behebens eines Failures erneut einen Failure gemeldet bekommt, da eine assert-Methode, die nach der korrigierten ausgeführt wird, fehlschlägt. So könnte man vermuten, dass der Fehler nach wie vor vorhanden ist, und man sucht ihn eigentlich an der falschen Stelle. Um dies zu vermeiden gibt JUnit einen Failure Trace aus, der im Grunde nichts anderes ist, als der normale Java Exception Trace.

Verfügbare Asserts in JUnit:

- True: Wert gleich true
- False: Wert gleich false
- Null: Wert gleich null
- NotNull: Wert ungleich null
- Same: Referenz muss übereinstimmen
- NotSame: Referenz darf nicht übereinstimmen
- Equals: Testet die Gleichheit mittels Object.equals

Die hier gezeigte Variante (Beispiel 4.3.1) ist noch durch die Methoden setUp() und tearDown() erweiterbar. Diese Methoden dienen dabei zum Einrichten der Testumgebung. Muss vor einem Test z.B. eine Datenbankverbindung initialisiert werden, so erledigt man dies in der setUp-Methode. Das bereinigen der Testumgebung, z.B. das Schließen der Datenbankverbindung, wird in der tearDown-Methode erledigt.

Es besteht auch die Möglichkeit, mehrere Tests in einer Testsuite zu vereinigen. Dazu schreibt man eine neue Klasse die man wieder von TestCase ableitet, diesmal jedoch nicht die Testfälle ausprogrammiert, sondern eine Testsuite erstellt. In diese Klasse kann man dann noch eine main-Methode einbauen, welche die Testsuite startet und die Ergebnisse z.B. mittels der Swing-Oberfläche von JUnit ausgibt.

Programmbeispiel:

```
public class TestAll extends TestCase{

        public static Test suite(){
                TestSuite suite = new TestSuite();
                suite.addTestSuite(Test1.class);
                suite.addTestSuite(Test2.class);
                suite.addTestSuite(Test3.class);
                return suite;
        }

        public static void main (String[] args) {
                junit.swingui.TestRunner.run(TestAll.class);
        }
}
```

Beispiel 4.3.3, Eine JUnit Testsuite

Wie man in Beispiel 4.3.3 sieht, werden die gesamten Testklassen zur Suite hinzugefügt. Somit werden die dort befindlichen Tests ganz normal ausgeführt, als würde man den Test alleine starten.

4.3.2 Ablauf eines JUnit Tests

Durch den Aufruf von z.B. junit.swingui.TestRunner.run(TestAll.class); in einer Main-Methode wird das Framework aktiv und beginnt mit seiner run() Methode. Diese ruft dann TestRunner.runTest() auf, welche die als Parameter übergebene Klasse nach Testfällen durchsucht. Dabei wird nach Methoden gesucht, welche mit „test“ beginnen (z.B. testAND), keine Parameter erwarten, und keine Werte zurückliefern. Alle, dieser Konvention entsprechenden Methoden, werden dann in beliebiger Reihenfolge aufgerufen. Für jede gefundene Methode wird setUp() und tearDown() benutzt, und somit für jeden Testfall eine unabhängige Testumgebung erstellt.

Will man als Entwickler jedoch eine bestimmte Reihenfolge der Tests, so kann man sich mittels individueller Tests Abhilfe schaffen. Dabei bietet JUnit zwei Arten an. statische und dynamische individuelle Tests. Bei statischen Tests überschreibt man dabei die Abstrakte Methode TestCase.runTest() mittels anonymer, innerer Klasse.

```
TestCase test = new IndividualTest("something test”) {
        public void runTest() {
                testSomething();
        }
};
```

Beispiel 4.3.4, Individueller Test mit anonymer Klasse

Der String der an new IndividualTest übergeben wird, dient als Name für den Test. Später bei der Ausführung der Tests scheint dieser Name in der Ausgabe auf. Diese Variante ist Typsicher, dauert in der Ausführung jedoch länger als die dynamische Variante.

```
TestCase test = new IndividualTest("testSomething");
```

Beispiel 4.3.5, dynamische Variante des individuellen Tests

Diese Variante ist schneller in der Ausführung, jedoch nicht mehr Typsicher.

4.4 Folgerungen

Unit Testing ohne JUnit ist heute nur noch schwer Vorstellbar, denn die Vorteile liegen auf der Hand. Automatisches Durchlaufen der Tests und Filtern der Informationen, Fehlerhandling mit genauem und übersichtlichem Failure- bzw. Errortrace, einfach zu installieren und zu handhaben, und der wahrscheinlich wichtigste Punkt für das Management, JUnit ist kostenlos. Verbunden mit dem Test Driven Development lassen sich Fehler schon in den Anfangsstadien der Entwicklung aufspüren, und so die Testzeiten des fertigen Produktes verkürzen.

Man sollte sich jedoch immer in Erinnerung rufen, dass JUnit die Testfälle nicht selbst entwickeln kann, dazu benötig man weiterhin Softwareentwickler und Tester. Doch durch die strikte Trennung von Programmcode und Testroutinen bedarf es keiner Nachbearbeitung des Programmcodes mehr vor Fertigstellung (z.B. entfernen der Codestücke, wenn man mittels Ausgabe auf die Standardausgabe getestet hat), und die Tests sind auch nach Projektabschluss weiterhin verwendbar

5 Quellen

5.1 JMeter

http://javaboutique.internet.com/tutorials/JMeter/index.html
http://clif.objectweb.org/load_tools_overview.pdf
http://woolfel.blogspot.com/2004/04/go-go-jmeter-if-youre-reading-this.html
http://jakarta.apache.org/jmeter/usermanual/index.html

5.2 JStyle

http://www.mmsindia.com/default.html
http://www.mmsindia.com/JStyle/docs/jstyleguide.html

5.3 Jemmy

http://jemmy.netbeans.org
http://jemmy.netbeans.org/samples/WaitWindowSample.java
http://jemmy.netbeans.org/samples/ResourceSample.java
http://jemmy.netbeans.org/samples/ResourceSample.txt
http://jemmy.netbeans.org/samples/FindComponentsSample.java

5.4 JUnit

http://www.junit.org
http://www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/6.junit/layout0.htm
(Funktioniert leider seit dem 18.11.04 nicht mehr, alternativ ein Link aus dem Cache von www.google.de)
http://216.239.59.104/search?q=cache:VKj90cBediIJ:www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/6.junit/layout0.htm

6 Anhang

6.1 Beispiel für JMeter

- Start JMeter 2.0 Right
- Select the first node "Test Plan"
- Right click -> Add -> Thread Group
- Select "Thread Group"
- uncheck "Forever" and enter 1
- Select "Thread Group" in the test plan
- Right click -> Add -> Config Element -> HTTP Cookie Manager
- Select "Thread Group"
- Right click -> Add -> Sampler -> HTTP Request (do this 4 times)
- Select the first "HTTP Request"
- Change the name to "check out page 1"
- Enter the hostname, and port
- Enter "checkout_1.html" for the path, or use your own page
- Add the request parameter by clicking "Add". Repeat the process until all the parameters are set
- Repeat this process for the remaining requests
- Select "Thread Group"
- Right click -> Add -> Listener -> View Results Tree