Johannes Kepler Universität Linz
Institut für Systemsoftware

Testen von Softwaresystemen

JUnit Extensions: GUI Tools

Angelika Brückl
Johanna Fitzinger
Günther Gsenger
Angelika Kusel

Linz, 15. November 2004

1 Inhaltsverzeichnis

2 Einleitung

„Don’t click to test!“, so könnte die Devise lauten, die hinter den Bemühungen für das automatisierte Testen von graphischen Benutzerschnittstellen steckt. Doch wie kann diese Devise in die Realität umgesetzt werden?

Ein Lösungsansatz, der sofort mit dem Automatisieren von Tests verbunden wird, sind Unit Tests. Die Vorteile, die Unit Tests bieten, wie zB sofortiges Feedback, ob ein Refactoring-Schritt an der Funktionalität eines Codestücks Änderungen hervorgerufen hat, sind weitreichend bekannt und geschätzt. Doch wie können Unit Tests auf Graphische Benutzerschnittstellen angewandt werden? Eine Antwort hierauf lautet JFCUnit. JFCUnit stellt eine Erweiterung von JUnit dar und dient dem automatisierten Testen von graphischen Benutzerschnittstellen. Ein andere Variante zur Erzeugung von GUI Unit Tests bietet Abbot oder Jemmy Module.

Ein weiterer Lösungsansatz besteht in speziellen Tools, die das automatisierte Testen von graphischen Benutzerschnittstellen ermöglichen. Zu diesen Tools zählen Marathon, CleverPHL(Jacareto), Costello(Skripteditor von Abbot), Pounder.

Im Folgenden gehen wir auf die oben genannten Lösungsansätze näher ein.

3 Lösungsansätze

3.1 Marathon

Übersicht

Marathon ist ein Test Tool für Java/Swing Applikationen. Die Bestreben des Marathon Entwickler-Teams liegen dabei laut eigenen Angaben bei der Entwicklung „eines der besten End-User Tools“ [MARA04a] für funktionelles Testen.

Marathon ist in Java geschrieben und benützt Python-Scripts [PYTH04] um seine Testfälle zu schreiben. Zusätzlich unterstützt es ein Capturing-Verfahren, mit dessen Hilfe man Testfälle durch Klicken erstellen kann und somit keine Testfälle von Hand schreiben muss.

Marathon ist dabei nicht – so wie man annehmen könnte – ein GUI-Test-Tool, sondern ein Tool für funktionelles Testen. Dies bedeutet im speziellen, dass Marathon keine Mauspositionen o.Ä. aufzeichnet, sondern nur die durchgeführten Aktionen. Marathon eignet sich deshalb nur zum Testen „konventioneller Anwendungen“ [MARA04b], die mit den standardmäßig in Java enthaltenen Steuerelementen, wie Buttons, Textfeldern usw. auskommen.

Installation

Voraussetzungen

Voraussetzung ist ein installiertes Java 2 JRE Version 1.4 (Version 1.5 wird zurzeit nicht, bzw. nur im Kompatibilitätsmodus unterstützt).

Möchte man Marathon selbst kompilieren, so benötigt man eine installierte eclipse-Plattform [ECLI04] bzw. Apache Ant [AANT04].

Beschaffung

Marathon ist freie Software und kann gratis herunter geladen werden [MARA04c]. Neben den Standardpaketen, die in Marathon enthalten sind, müssen auch noch die Support-Pakete von der Homepage herunter geladen werden.

Alternativ kann der Quellcode auch mittels anonymen CVS-Zugangs herunter geladen werden.

Installation der Pakete

Hat man die Pakete herunter geladen, so dekomprimiert man diese in ein Verzeichnis. Ab dann kann Marathon mittels der Datei marathon.bat bzw. marathon.sh für Unix-Systeme gestartet werden.

Installation der Sourcen

Wenn man sich die Quellcode-Dateien herunter geladen hat, so kann man diese entweder in eclipse [ECLI04] als ein Projekt importieren, oder man benutzt Apache Ant [AANT04], um die Quelldateien zu kompilieren und ein jar-File zu erstellen. Dieses kann dann wiederum über marathon.bat bzw. marathon.sh gestartet werden.

Testscripts in Marathon

Wie bereits weiter oben erwähnt, werden Python-Scripts benützt, um die Testfälle zu beschreiben. Dies soll den Vorteil haben, dass diese Scripts leichter lesbar werden und auch von Kunden verstanden werden können.

Diese Testscripts können auf 2 Arten erstellt werden:

- Sie können von Hand geschrieben werden oder
- Durch das Aufzeichnen bestimmter Aktionen aufgenommen werden.

Ein Marathon Projekt ist in 3 Untergruppen aufzuteilen:

- Den *TestCases*: Sie beschreiben tatsächlich die Testfälle für die Applikation.
- Den *CaptureScripts*: Dies sind die aufgezeichneten Scripts

- Den *Fixtures*: Eine Fixture ist eine Art Initialisierungsskript. Darin kann man z.B. Argumente für den Programmaufruf selbst definieren. Die Hauptklasse des zu testenden Programms wird dabei auch in dieser Fixture beschrieben.

Schreiben eines Testscripts von Hand

Ein Testscript ist ein normales Python-Script, hat also die Dateiendung .py.

Es stehen dabei u.A. folgende vordefinierten Funktionen zur Verfügung:

Funktion	**Bedeutung**	**Beispiel**
window	Es wird mit der Ausführung des Programms gewartet, bis das Fenster mit dem angegeben Namen geladen wurde.	window('Simple Widgets')
close	Es wird erwartet, dass das zuletzt geöffnete Fenster geschlossen wird.	close()
click doubleclick rightclick	Führt einen Klick/Doppelklick/Rechtsklick auf eine Komponente aus.	click('country') doubleclick('file') rightclick('username')
select	Fügt einen gegebenen Text in eine Komponente ein.	select('name', 'Mustermann')
assertText	Nimmt an, dass eine bestimmte Komponente den angegebenen Text als Wert enthält.	assertText('name', 'Mustermann')
assertEnabled	Nimmt an, dass eine bestimmte Komponente aktiviert ist	assertEnabled('nameCheckbox', true, '2')
assertContent	Nimmt einen bestimmten Inhalt für eine komplexe Komponente (wie z.B. einem JTree)	assertContent('JTree', [['Root', 'Colors', 'Red', 'Blue', 'Green', 'Yellow', 'Magenta', 'Cyan', 'Sports', 'Individual Sports', 'Tennis', 'Badminton', 'Team Sports', 'Cricket', 'Hockey', 'Football', 'Volley Ball']])

Tabelle 3.1.1 Einige Funktionen, die bei Marathon zur Verfügung stehen

Zusätzlich gibt es noch weitere Assertationen, wie z.B. assertRowCount und assertColor, denen aber keine so große Bedeutung zukommt.

Neben diesen Funktionen kann man natürlich die standardmäßig in Python enthaltenen Features, wie Funktionsaufrufe usw. nützen. So kann man beispielsweise häufig durchgeführte Aktionen zu Funktionen zusammenfassen und diese dann durch einen Funktionsaufruf ausführen lassen.

Erstellen eines Testscripts mittels Capturing

Wie bereits erwähnt, kann man Testscripts in Marathon auch mittels Capturing erstellen. Dazu erstellt man zuerst ein neues CaptureScript, in das später die Kommandos, Assertationen usw. aufgenommen werden. Während des Capturing werden automatisch die durchgeführten Aktionen als Python-Kommandos in das Script eingefügt. Das entstandene Script hat also dieselbe Form wie eines, das von Hand geschrieben wurde.

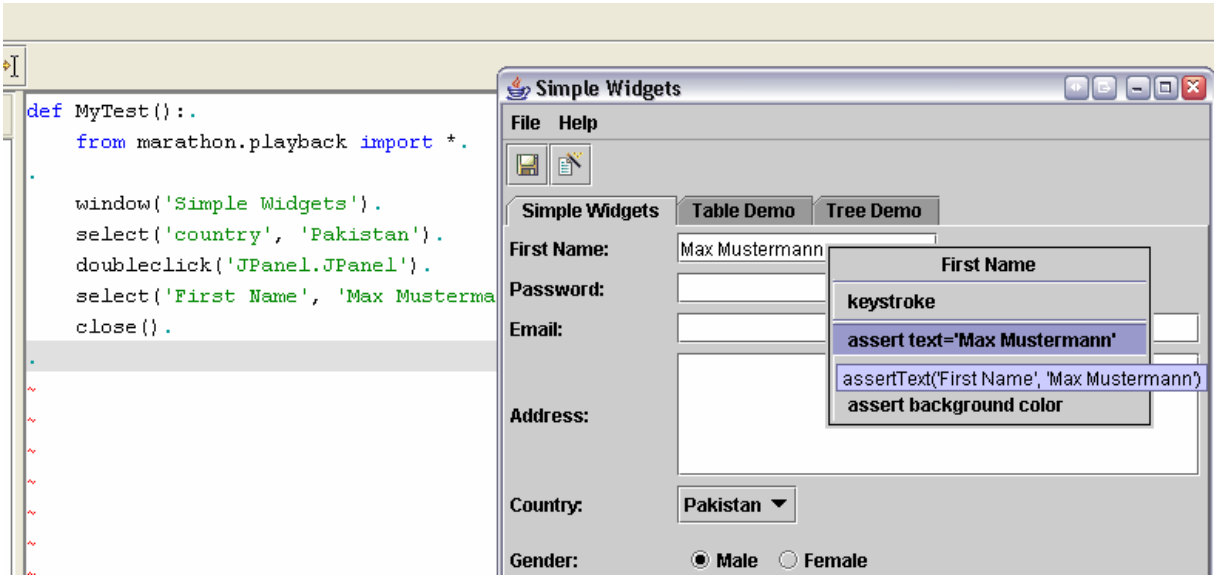


Abbildung 3.1.1 Einfügen einer Assertation in ein Marathon Script

Auf diese Art und Weise sollte sichergestellt werden, dass das Schreiben und Verstehen von Testfällen wirklich von jedem Entwickler, Tester und sogar Kunden verstanden und bewältigt werden kann.

Ausführen der Testfälle

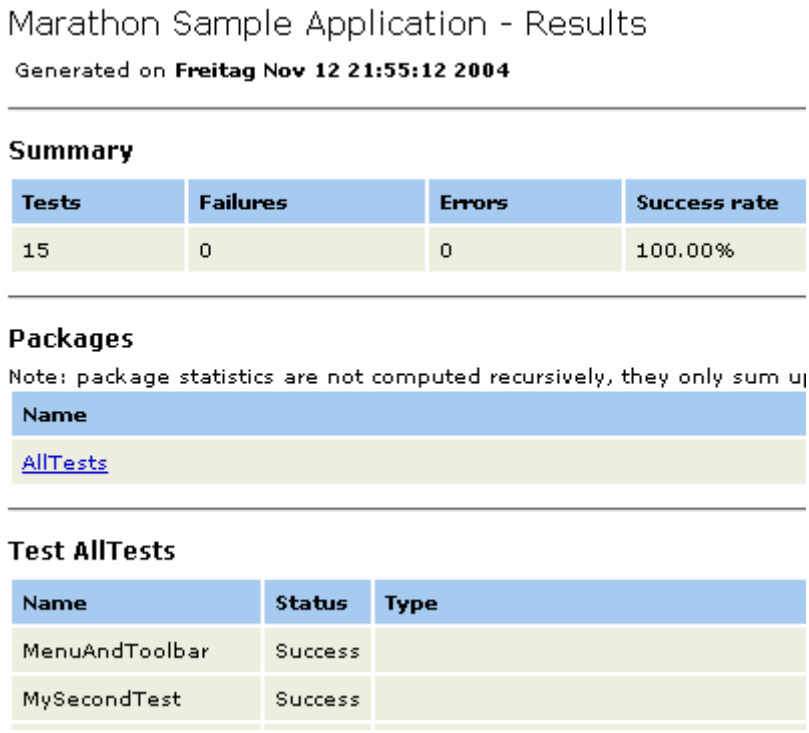
Marathon bietet verschiedene Arten an, Testfälle auszuführen. Diese kann man grob in interaktive und im Batchmodus ausgeführte Tests unterteilen.

Die interaktiv durchgeführten Tests werden direkt mit der graphischen Oberfläche, die Marathon zur Verfügung stellt, ausgeführt. Am Schluss wird das Ergebnis der durchgeführten Tests in einem Fenster dargestellt.

Im Batchmodus werden alle Tests, die sich im TestCases Verzeichnis des Projekts befinden, ausgeführt. Der Output kann auf verschiedene Arten erfolgen.

Im Textmodus gibt Marathon die Ergebnisse in Text-Form aus.

Im HTML-Modus werden die Ergebnisse zu HTML formatiert.



Marathon Sample Application - Results

Generated on **Freitag Nov 12 21:55:12 2004**

Summary

Tests	Failures	Errors	Success rate
15	0	0	100.00%

Packages

Note: package statistics are not computed recursively, they only sum u

Name
AllTests

Test AllTests

Name	Status	Type
MenuAndToolbar	Success	
MySecondTest	Success	

Abbildung 3.1.2 Ausgabe von Marathin im HTML-Format

Im XML-Modus werden die Ergebnisse in eine XML-Datei ausgegeben.

3.2 Jacareto

Bei Jacareto handelt es sich um ein Capture&Replay Framework für Java Applikationen und Applets. Das Framework ermöglicht die einfache Erstellung von Capture&Replay Tools. Capture and Replay bedeutet dass Benutzerinteraktionen, wie zum Beispiel ein Tastendruck oder Mausaktionen aufgezeichnet, und wiedergegeben werden können. Die Erfassung, der vom Benutzer erzeugten Events, erfolgt in Jacareto durch die Registrierung von Eventlistener in der systeminternen Eventqueue von

Java. Alle anfallenden Events werden über die Eventqueue an die registrierten Listener weitergegeben und dort verarbeitet. Auf diese Weise kann Jacareto, durch die Registrierung entsprechender Listener Benutzeraktionen aufzuzeichnen. [BENK03]

Zwei Tools sind im Jacareto Package enthalten: CleverPHL und Picoder, wobei CleverPHL umfassendere Funktionen bietet und über eine komplexere Oberfläche verfügt. Ich werde mich daher in meiner Ausarbeitung auf dieses Tool beschränken.

CleverPHL ist kein reines GUI-Test Tool. Die Capture & Replay Funktionalität des Tools kann beipielsweise auch zur Analyse von Benutzerverhalten oder zur Erstellung von Demonstrationen genutzt werden.

Jacareto ist als Open Source in Version 0.7.08 verfügbar

Testen von GUIs mit CleverPHL

Von CleverPHL werden die Benutzer-Interaktionen in einem Record gespeichert. Man hat dann die Möglichkeit in diesen Record zusätzlich Elemente einzufügen: z.B. eine Anmerkung, einen Audio-Clip, eine Pause oder eben einen Test.

Abbildung 3.2.1 zeigt die Oberfläche von CleverPHL. Links sieht man den aufgezeichneten Record, mit den einzelnen Events.

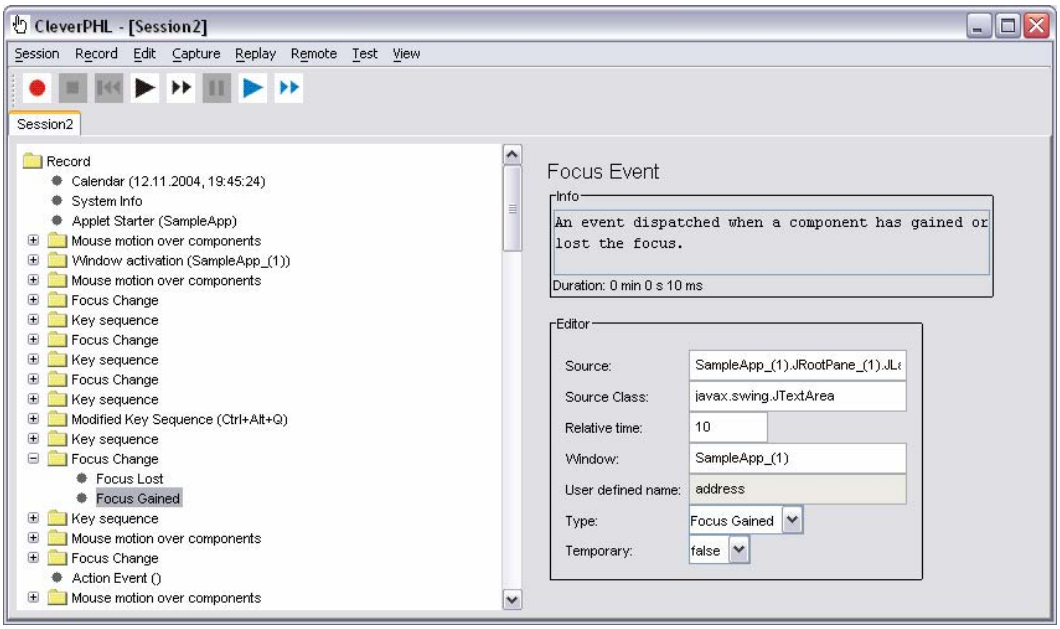


Abbildung 3.2.1 Oberfläche von CleverPHL

Durch das Einfügen von Tests in den Record kann dann der Status der GUI Elemente überprüft werden. Wenn der Record wiedergegeben wird, werden die Tests ausgeführt.

Es existieren vordefinierte Tests für einzelne GUI Elemente. Beispiele sind unter anaderem:

- **Toggle Button Tests:** Der Test überprüft den Status des Toogle Buttons (ausgewählt oder nicht ausgewählt)
- **Text Component Tests:** Der Inhalt von text fields und text areas kann mit Hilfe eines 'Text Component Tests' mit einem String verglichen werden. Der Text schlägt fehl wenn dir Zeichenketten ungleich sind. Der Ziel-String kann mit Hilfe des 'Text Component Tests' auch gegen einen regulären Ausdruck geprüft werden.
- **Tabbed Pane Tests:** Vergleicht den ausgewählten Index eines tabbed pane mit einem gegebenen Index.

Einfügen von Tests in den Record

Es gibt 2 Möglichkeiten um einen Test in den Record einzufügen:

1. Erstellung von Tests durch Auswahl von Komponenten in der Zielapplikation:

Die Vorgehensweise sieht so aus, dass zuerst so lange Benutzerinteraktionen aufgezeichnet werden, bis man zu einem Punkt kommt wo man den Status einer Komponente durch einen Test überprüfen möchte. Man stoppt den Capture Vorgang und fügt dann durch Auswählen der zu testenden Komponente in der Zielanwendung einen Test in den Record ein. Dabei wird von CleverPHL automatisch der passende Test für die Komponente ausgewählt. Durch Anwählen des eingefügten Tests im Record können die einzelnen Attribute des Tests festgelegt werden. (siehe Abbildung 3.2.2) Für jeden Test kann festgelegt werden ob, wenn der Test im Replay-Vorgang fehlschlägt, der Fehler ignoriert werden und die Wiedergabe fortgesetzt werden soll und ob Fehler korriegiert werden sollen.

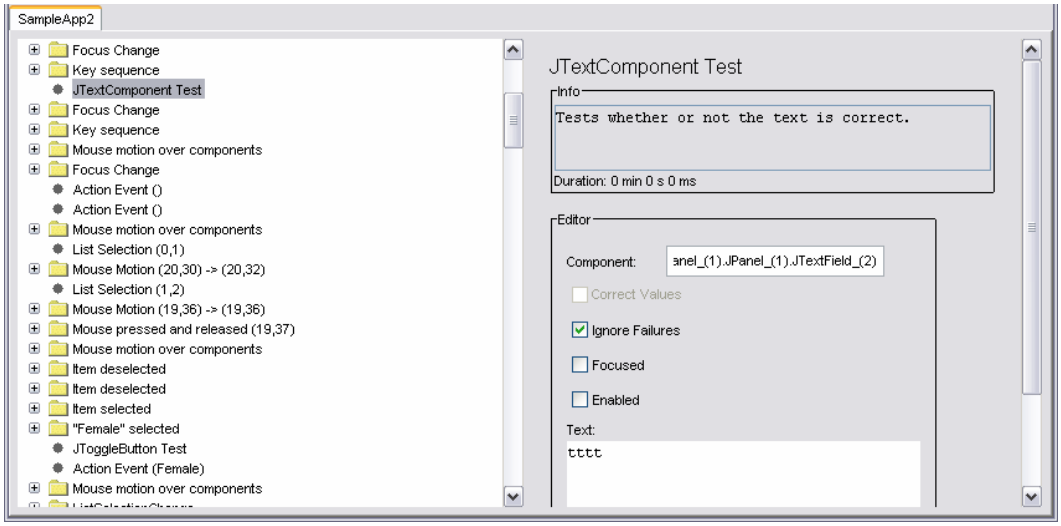


Abbildung 3.2.2 Vornehmen von Testeinstellungen

2. Einfügen von Tests in das Component Window:

Im Component Window werden alle Komponenten der Zielanwendung aufgelistet.Über das Component Window kann man auf einmal Tests für mehrere Komponenten oder für einen ganzen Baum von Komponenten einfügen.

Ausführung von Tests während des Replay Vorgangs

Während des Replay Vorgangs werden die Tests ausgeführt. Die Ergebnisse der Tests warden gesammelt und können nach der Ausführung betrachtet werden. [JACA04]

Fazit

CleverPHL wäre vermutlich nicht das GUI Test Tool meiner Wahl. Die Capture& Replay Fähigkeit steht im Vordergrund bei CleverPHL – das Testen ist ein zusätzliches Feature, das noch nicht sehr ausgereift ist. Positiv ist anzumerken, dass das Tool mit freiem Sourcecode verfügbar ist, und so von den Usern gemäß ihren Bedürfnissen angepasst werden kann.

3.3 Abbot

Das Abbot Framework stellt eine eine Java Bibliothek für UNIT Tests sowie für Funktionstests von GUIs zur Verfügung. Es liefert Methoden, um Benutzertätigkeiten zu reproduzieren und den Zustand der GUI Elemente zu überprüfen. Das Framework kann direkt im Java Code aufgerufen werden, oder ohne Programmierung mittels Scripts verwendet werden.

Es gibt im wesentlichen 2 Möglichkeien des Testens mit Abbot:

- Programmierung von GUI Unit Tests mit Hilfe der Abbot-Bibliothek (Test - First-Development)
- Erstellen von Skripten mittels eines Scripteditors. Als Scripteditor steht Costello zur Verfügung.

Erstellung von GUI Test Code mittels Programmierung (Test-First Development):

Das Testen mit dem Abbot Framework möchte ich anhand eines einfachen Beispiels erklären. Betrachten wir den einfachen Test eines Arrowbuttons.

Testen eines Butttons mit folgender Funktionalität:

- Der Button zeigt in eine von vier Richtungen
- Feuert ein Event wenn er gedrückt wird

Um das Verhalten des Buttons zu überprüfen schreiben wir eine Klasse, die von ComponentTesFixture erbt.

```
public class ArrowButtonTest extends ComponentTestFixture {

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

    public static void main(String[] args) {
        TestHelper.runTests(args, ArrowButtonTest.class);
    }
```

Um Aktionen auf der Komponente zu ‘simulieren’ wird Klasse ComponentTester verwendet.

```
    private ComponentTester tester;
    protected void setUp() {
        tester = ComponentTester.getTester(ArrowButton.class);
    }
```

In der Methode testClick() wird überprüft ob der Button ein Event feuert, wenn er gedrückt wird.

```
private String gotClick;
public void testClick() {
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            gotClick = ev.getActionCommand();
        }
    };
    ArrowButton left = new ArrowButton(ArrowButton.LEFT);
    ArrowButton right = new ArrowButton(ArrowButton.RIGHT);
    ArrowButton up = new ArrowButton(ArrowButton.UP);
    ArrowButton down = new ArrowButton(ArrowButton.DOWN);

    left.addActionListener(al);
    right.addActionListener(al);
    up.addActionListener(al);
    down.addActionListener(al);

    JPanel pane = new JPanel();
    pane.add(left);
    pane.add(right);
    pane.add(up);
    pane.add(down);
    // This method provided by ComponentTestFixture
    showFrame(pane);

    gotClick = null;
```

Ausführen der Tests auf der Komponente. Mit assertEquals() wird überprüft ob ein Event abgefeuert wurde.

```
    tester.actionClick(left);
    assertEquals("Action failed", ArrowButton.LEFT, gotClick);
    gotClick = null;
    tester.actionClick(right);
    assertEquals("Action failed", ArrowButton.RIGHT, gotClick);
    gotClick = null;
    tester.actionClick(up);
    assertEquals("Action failed", ArrowButton.UP, gotClick);
    gotClick = null;
    tester.actionClick(down);
    assertEquals("Action failed", ArrowButton.DOWN, gotClick);
}
```

Testen mit Hilfe von Scripten mit demm Scripteditor Costello

Im Gegensatz zum ersten Ansatz, dem Testen auf Codeebene, das den Test-First-Development Ansatz unterstützt, kann man mit Costello Testscripts für bestehende Anwendungen für Funktionstests erzeugen.
Mit Costello kann man Benutzerinteraktionen aufzeichnen, Tests einfügen, und somit einfach Testscripte erezeugen.

Die Erzeugung von Testscripts mit Costello sieht so aus, dass man zuerst Interaktionen aufzeichnet, bis zu einem Punkt an dem man einen Test einfügen möchte. Danach wählt man, die Komponente, deren Zustand man überprüfen möchte in der Zielapplikation aus, und fügt anschließend in das Script einen Test einfügen. Dazu wählt man zuerst die Eigenschaft der Komponente, die man überprüfen möchte aus, und fügt dann das gewünschte Assert-Satetement in den Test ein. (siehe Abbildung 3.3.1)

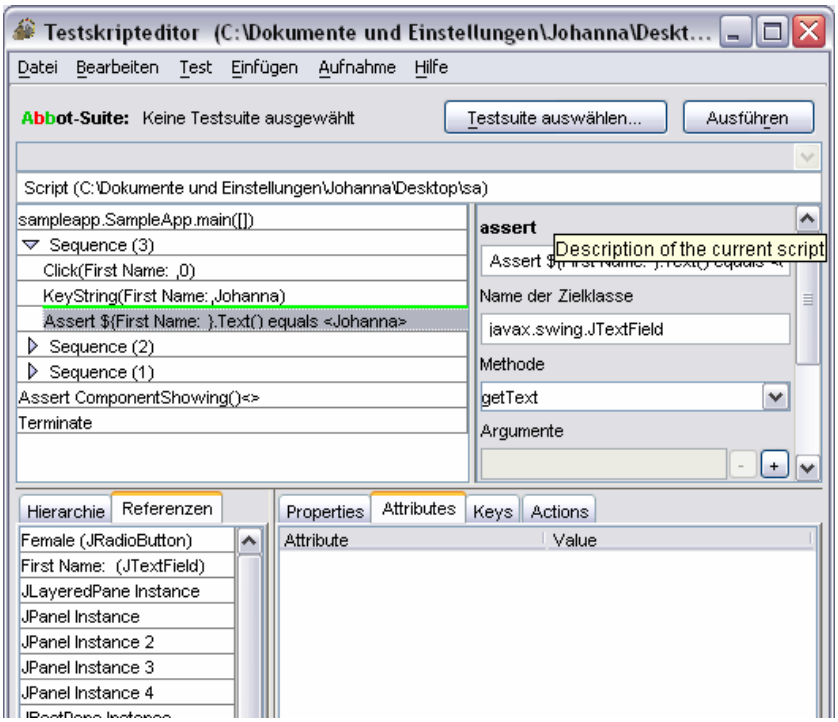


Abbildung 3.3.1 Erzeugen eines Scripts mit Costello

Die von Costello erzeugten Scripte werden in Form von XML-Dateien gespeichert.

Hier ein Auszug aus dem so erzeugten Script (Als Beispiel wurde wieder das SampeApp Applett von Marathon verwendet):

```
<?xml version="1.0" encoding="UTF-8"?>
<AWTTestScript>
  <component class="javax.swing.JRadioButton" id="Female" index="1" parent="JPanel Instance 4"
text="Female" window="Simple Widgets 4" />
  <component class="javax.swing.JTextField" id="First Name: " index="1" label="First Name: " parent="JPanel
Instance 3" window="Simple Widgets 7" />
…..
  <launch args="[]" class="sampleapp.SampleApp" classpath="D:\Eigene Dateien\UNI\SEM 9\Testen von
Softaweresystemen" desc="sampleapp.SampleApp.main([])" method="main" />
  <sequence>
    <action args="First Name: ,0" class="javax.swing.text.JTextComponent" method="actionClick" />
```

```
   <action args="First Name: ‚Johanna" method="actionKeyString" />
   <assert component="First Name: " method="getText" value="Johanna" />
  </sequence>
  <terminate />
</AWTTestScript>
```

Das erzeugte Script besteht hauptsächlich aus Referenzen auf GUI Komponenten, den aufgezeichneten Aktionen, und Assertions. [ABBO04]

Fazit

Abbot ist meiner Meinung nach ein sehr gutes Tool. Es bietet sowohl eine Unterstützng für Test-First-Development auf Codeebene, als auch eine Unterstützung für das Automatisieren von Funktionstests mittels Scripts.
Die Bedienung von Costello ist leicht erlernbar und macht so das Testen einfach und schnell.

3.4 Pounder

Übersicht

Pounder ist ein weiteres Testtool, welches Java GUIs testet. Genauer ist es ein Record-Playback Tool [MARC04], das heißt mit diesem Tool ist es möglich Java GUI Komponenten zu laden und von einer bestimmten Aktionsreihenfolge ein Skript aufzuzeichnen. Diese Skripte kann man dann für JUnit Tests verwenden, oder auch von Pounder wiedergeben lassen [POUN04]. Pounder ist lizenziert unter GNU Library General Public License. Die aktuelle Version ist 0.95 und zum freien Download verfügbar unter [POUN04a].

Installation von Pounder:

Pounder wird in Form eines *.jar-Files gelierfert, oder in einem *.zip-File. Das jar-File braucht man nur mit einem Doppelklick öffnen, oder mit dem Befehl java –jar pounder_0.95.jar. [POUN04b]

Funktionsumfang von Pounder

Pounder zeichnet Aktionen, die auf einem Applet, oder einer AWT-Platform durchgeführt werden auf. Diese Aktionen können sein:

- Mausaktionen, wie ziehen, klicken, oder Bewegungen mit der Maus, oder dem Wheelrad

- Window events, wie „Wiederhestellen“, „Maximieren“ oder ziehen eines Fensters
- Key events

Skript erstellen mit Pounder

Hierzu verwenden wir das Applet „SampleApp“, wie in den vorher durchgeführten Tests mit den anderen Tools.

Aufnahme des Skriptes:
Um diese Klasse dynamisch laden zu können, muss sie in einem Classpath-Verzeichnis liegen. Nach der Eingabe des Klassennamens „sampleapp.SampleApp“ drückt man die Eingabetaste.

Um die Aufnahme zu beginnen drückt man auf die Taste „Aufnahme“, dann öffnet sich das SampleApp-Applet und man kann die Testaktionen durchführen. Um einen Zwischenstopp zu machen, kann man auf „Pause“ drücken, oder wenn man fertig ist, dann drückt man auf „Stopp“. Um das Skript wieder verwenden zu können, muss man es natürlich speichern. Dieses Skript kann man dann wiedergeben mit Pounder.

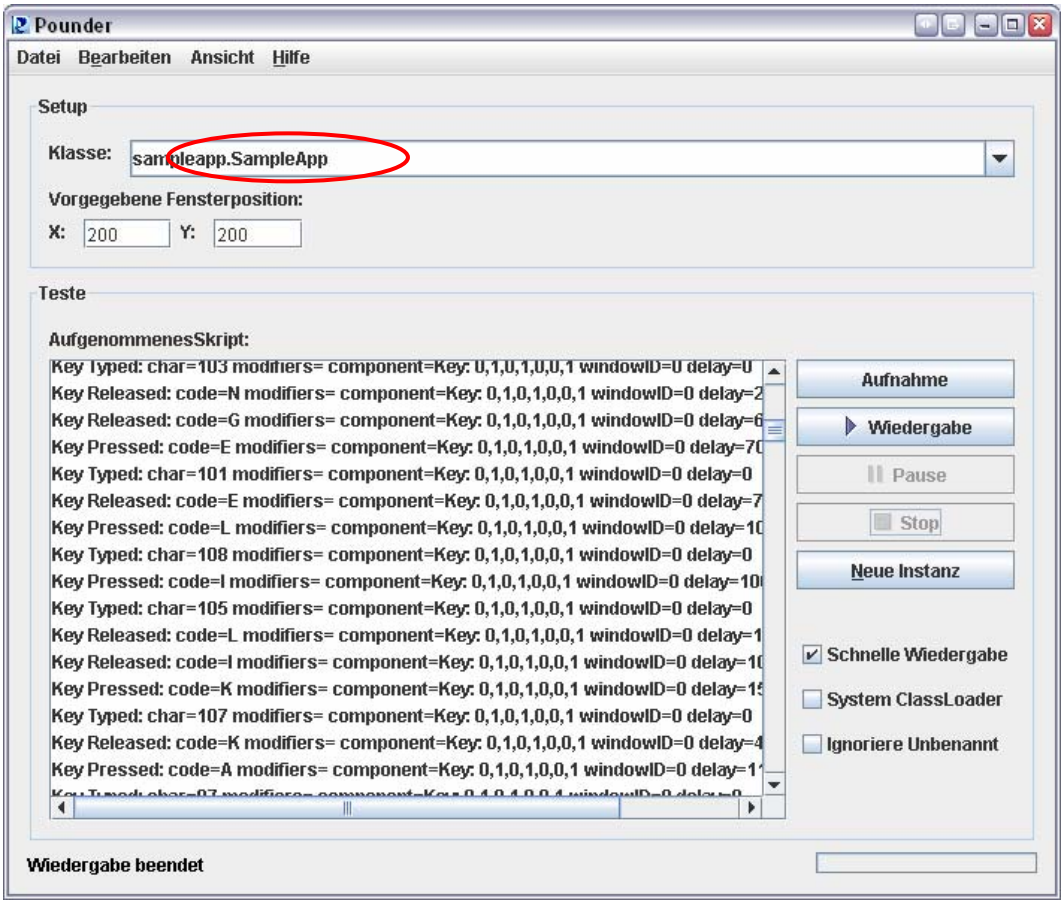


Abbildung 3.4.1 Aufnahme von Skript

Einbinden eines Pounder-Skriptes in JUnit

Mit unserem Beispiel würde das folgendermaßen ablaufen: Die Klasse Player von Pounder muss importiert werden um das Skript abspielen zu können. Bei der Instanzierung Player player = new Player („scripts/simpleapp1.pnd“), kann man dann das entsprechende Skript übergeben.

Die Java-Datei dazu würde wie folgt ausschaun:

```
import junit.framework.TestCase;
import junit.framework.Test;
import junit.framework.TestSuite;

import com.mtp.pounder.Player;

import my.package.MyComponent;

public class TestSomething extends TestCase {

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

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

  /** Test something. **/
  public void testThatFeature() throws Exception {
    Player player = new Player("scripts/simpleapp1.pnd");
    MyComponent mc = (MyComponent)player.play();

    assertEquals(1, mc.someMethod());
  }

}
```

Mit den Assert-Statements kann man dann, wie gewohnt einen JUnit-Test durchführen. [POUN04b]. Den ganzen Umfang der Klasse Player findet man unter [POUN04c].

3.5 Jemmy Module

Das Jemmy Framework stellt eine Java Bibliothek für GUI-Applikationen zur Verfügung. Jemmy bietet Methoden an mit denen man alle Benutzeraktionen reproduzieren kann, die auf Swing/AWT Komponenten ausgeführt wurden, wie z.B.: drü-

cken eines Buttons, schreiben eines Textes. Mit Jemmy lassen sich dabei hauptsächlich Tests durchführen, die die Präsenz einer gewissen Komponente überprüfen. So kann man mit Jemmy beispielsweise überprüfen, ob nach dem Drücken eines Menüeintrags der geforderte Dialog erscheint, oder ob das Anwählen eines Reiters die gewünschten Komponenten zum Vorschein bringt. Erscheint eine Komponente, die erwartet wird, nach einer gewissen Zeit nicht, so wirft Jemmy eine TimeoutExpiredException. Diese Exception wird im Testfall normalerweise abgefangen und zurückgegeben, dass der Test gescheitert ist.

Es besteht auch die Möglichkeit ein Demo, wo man Kommentare einfügen kann, wie „Drücken Sie hier!“, mit diesem Tool zu erstellen. [JEMM04]. Jemmy ist ein Open Source Projekt von NetBeans™ und steht zum Download zur Verfügung unter [JEMM04a]. Jemmy kann auch gemeinsam mit NetBeans IDE verwendet werden.

Die Jemmy-API

Die Jemmy-API setzt sich aus folgenden Packeten zusammen: [JAPI04]

org.netbeans.jemmy (Main Jemmy Package)

Dieses Package beinhaltet find/wait Komponenten.

org.netbeans.jemmy.demo (Demo package)

In diesem Package beinhaltet die Interface, Klassen und Methoden mit deren Hilfe man eine Applikationsdemo erstellen kann.

org.netbeans.drivers

Dieses Packet bietet low-level-Funktionen an, welche man bei einem normalen Test nicht braucht.

org.netbeans.jemmy.explorer

Dieses Package beinhaltet die zwei Klassen GuiBrowser und TrialListenerManager. GuiBrowser untersucht eine Java GUI Applikation und TrialListenerManager findet die Event Sequence, welche die ganzen Aktionen wiedergeben sollte.

org.netbeans.jemmy.image

Dieses Package stellt Klassen zur Verfügunng mit deren Hilfe man zwei Images vergleichen kann.

org.netbeans.jemmy.operators

In diesem Package befinden sogenannte Operatoren, die auf Testseite eine Komponente darstellen. Jede Operator-Klasse deckt die Funktionalität einer Java-Komponente ab, wie z.B.: JButtonOperator wird verwendet für javax.swing.JButton.

org.netbeans.jemmy.util

Testen mit Jemmy

Ein einfaches Beispiel mit dem Jemmy Framework [JEMM04b].
Am Anfang jedes Testprogrammes muss man natürlich die erforderlichen Namensräume importieren. Weiters muss man auch noch das org.netbeans.jemmy.TestScenario Interface inkludieren. Dieses Interface beinhaltet die Methode runIt(java.lang.Object param), welche die Application startet, welche getestet werden soll. Dieser Test überprüft, ob der Dialog Properties im Menü Tools verhanden ist.

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

public class WaitDialogSample implements Scenario {
    public int runIt(Object param) {
  try {
      //start application
      new ClassReferen-
ce("org.netbeans.jemmy.explorer.GUIBrowser").startApplication();
      //wait frame
      JFrameOperator mainFrame = new JFrameOperator("GUI Browser");
      //push menu
      //pushMenuNoBlock is used, because dialog is modal
      //see tutorial for more information
      new JMenuBarOperator(mainFrame).pushMenuNoBlock("Tools|Properties", "|");
      //wait dialog
      new JDialogOperator(mainFrame, "Properties");
  } catch(Exception e) {
      e.printStackTrace();
      return(1);
  }
  return(0);
   }
   public static void main(String[] argv) {
  String[] params = {"WaitDialogSample"};
  org.netbeans.jemmy.Test.main(params);
   }
}
```

Zukunftspläne

Implementierung eines „Recording modules“ und Einbindung von Abbot.

3.6 JFCUnit

Einleitung

JFCUnit stellt – wie schon in der Einleitung erwähnt - eine direkte Erweiterung des JUnit Frameworks dar. JUnit selbst bietet ja keine direkte Unterstützung für das Testen von graphischen Benutzerschnittstellen. JFCUnit erweitert nun JUnit um genau diese Funktionalität. JFCUnit ist ein Open Source Projekt unter SourceForge. Aktuell ist es in der Version 2.06 unter [JFCU04] erhältlich.

Das JFCUnit-Framework im Überblick

Angenommen, wir möchten unsere graphische Benutzerschnittstelle mit JUnit – also ohne JFCUnit – testen. So entstehen laut [HAMM04] folgende zwei Probleme:

- Das erste Problem liegt darin, dass die JUnit Tests zu laufen beginnen würden, bevor die Swing Applikation läuft, weil die Swing Applikation eine bestimmte Zeit zum Starten benötigt. Aus diesem Grund würden die Tests schief gehen, ohne dass die graphische Benutzerschnittstelle wirklich getestet worden wäre.
- Das zweite Problem liegt in der Interaktion mit der Swing Applikation. Es gibt keine einfache Möglichkeit, auf Swing Komponenten von außerhalb zuzugreifen bzw. sie zu manipulieren.

JFCUnit löst beide Probleme und bietet folgende Funktionalität an:

- Bestimmung von aktuell geöffneten Fenster- und Dialog-Objekten
- Finden von Swing Komponenten (zB Buttons, Labels, …) in einer Komponentenhierarchie
- Simulation von Eingabeereignissen (zB Klicken auf einen Button)
- Threadsicheres Testen von Komponenten

Wie diese Funktionalität benutzt werden kann, zeigt der weiter unten angeführte Testfall. Doch zuerst noch ein paar Worte zur Installation.

Installation

Da JFCUnit auf JUnit aufbaut, benötigt man natürlich zuerst JUnit. JFCUnit benötigt JUnit in der Version 3.7 oder höher - zu beziehen unter [JUNI04]. Des Weiteren setzt JFCUnit das Jakarta „Regular Expression“ jar-File ab der Version 1.2 voraus. Dieses erhält man unter [REGE04]. JFCUnit selbst erhält man unter [JFCU04].

Nach dem Download extrahiert man die Pakete in beliebige Verzeichnisse (zB c:\junit, c:\ jakarta-regexp-1.3, c:\jfcunit) – und schon ist man startklar!

Anmerkung: Möchte man JFCUnit unter Eclipse oder JBuilder verwenden, so stehen hierfür eigene PlugIn-Pakete zur Verfügung.

Erstellen eines einfachen Testfalls

Angenommen man möchte folgende (zugegebenermaßen triviale) Swing-Anwendung (Abbildung 3.6.1), welche nur aus einem Button mit der Aufschrift „OK“ besteht, testen (übernommen von [JFCU04a]):

```
import javax.swing.*;

public class MyApp {

  private JFrame mainFrame;
  private JButton ok;

  public static void main(String[] args){
        new MyApp();
  }

  private MyApp(){
        mainFrame = new JFrame();
        ok = new JButton("OK");
        mainFrame.getContentPane().add(ok);
        mainFrame.show();
  }

}
```

Abbildung 3.6.1 triviale Swing-Anwendung

Nun stellt sich natürlich die Frage, was man an dieser trivialen Anwendung testen soll. Nun man könnte zB testen, ob die Anzahl der gezeigten Fenster beim Starten der Applikation korrekt ist – in unserem Fall: 1. Dies ist zwar für diese Mini-Applikation trivial, jedoch zeigt es den Aufbau eines JFCUnit-Testfalls. Folgender JFCUnit-Testfall überprüft genau dies:

```
import junit.extensions.jfcunit.*;
```

```
import java.util.*;

public class MyAppT extends JFCTestCase{

  private JFCTestHelper helper;

  public MyAppT(String test){
        super(test);
  }

  public void setUp(){
        helper = new JFCTestHelper();
  }

  public void testMain(){
        List windows;

        MyApp.main(new String[0]);

        awtSleep();

        windows = helper.getWindows();
        assertEquals("Number of windows showing up is incorrect", 1,
        windows.size());
  }

}
```

Um mit den Klassen des JFCUnit-Frameworks arbeiten zu können, muss natürlich zuerst der Namensraum importiert werden. Im Gegensatz zu JUnit, wo Klassen, die Testfälle enthalten, von der Klasse TestCase erben, erben JFCUnit Testklassen von der Klasse JFCTestCase. Allerdings gibt es eine Verbindung zwischen der Klasse TestCase und JFCTestCase. Die Klasse JFCTestCase erbt nämlich direkt von der Klasse TestCase. Aus diesem Grund stellt die Klasse JFCTestCase auch alle Möglichkeiten von TestCase zur Verfügung.

JFCTestHelper ist wie der Name schon erahnen lässt, eine Hilfsklasse für Testfälle. Diese Hilfsklasse dient beispielsweise dazu GUI-Elemente (zB einen Button) in einer GUI-Hierarchie zu finden (also eine Referenz darauf zu erhalten) oder Events (zB einen Klick auf einen Button) auf GUI-Elementen auszulösen.

Die Methode setUp() dient diversen Initialiserungsarbeiten wie zB dem Erstellen einer Instanz von JFCTestHelper. Das Pendant zur Methode setUp() stellt tearDown() dar, welches für diverse Abschlussarbeiten eingesetzt werden kann.

In der Testmethode selbst (void testMain()) wird die zu testende Applikation gestartet. awtSleep() stellt eine Methode dar, mit deren Hilfe man mit dem AWT-Thread interagieren kann. Beim „normalen“ Lauf einer GUI-Applikation hat einzig und allein der AWT-Thread Zugriff auf die GUI-Elemente. Da wir aber beim Testen von

graphischen Benutzeroberflächen auch Zugriff auf die GUI-Elemente benötigen bzw. mit ihnen interagieren, ist es wichtig, dass der AWT-Thread in der Zwischenzeit schläft, damit keine Multi-Threading-Probleme auftreten. Aus diesem Grund wird der AWT-Thread während der Ausführung unserer Testfälle von JFCUnit geblockt. Damit sich jedoch Ereignisse, die wir auf GUI-Elementen auslösen, sich im GUI auch auswirken, muss der AWT-Thread zeitweise weiterarbeiten. Die Methode awtSleep() lässt hier den AWT-Thread für eine bestimmte Zeit wieder aufleben – die Zeitdauer für dieses Aufleben kann über die Methode setSleepTime(long time) eingestellt werden.

Der Aufruf helper.getWindows() gibt eine Liste aller Fenster zurück. Nun brauchen wir nur noch mit einer gewöhnlichen assertEquals(…) Methode, wie wir sie aus JUnit-Tests kennen, überprüfen, ob die Anzahl der Fenster in der erhaltenen Liste gleich 1 ist.

Um nun den Testfall laufen zu lassen, müssen zuerst die Dateien MyApp.java und MyAppT.java unter Einbindung der entsprechenden jar-Files übersetzt werden:

```
javac -classpath c:\junit3.8.1\junit.jar;c:\jfcunit\jfcunit.jar;c:\jakarta-regexp-1.3\jakarta-regexp-1.3.jar *.java
```

Anschließend kann der Testfall mit folgendem Aufruf ausgeführt werden (es kann natürlich auch der textui TestRunner von JUnit verwendet werden):

```
java -cp .;c:\junit3.8.1\junit.jar;c:\jfcunit\jfcunit.jar;c:\jakarta-regexp-1.3\jakarta-regexp-1.3.jar junit.swingui.TestRunner MyAppT
```

Das Ergebnis der Ausführung zeigt Abbildung 3.6.2

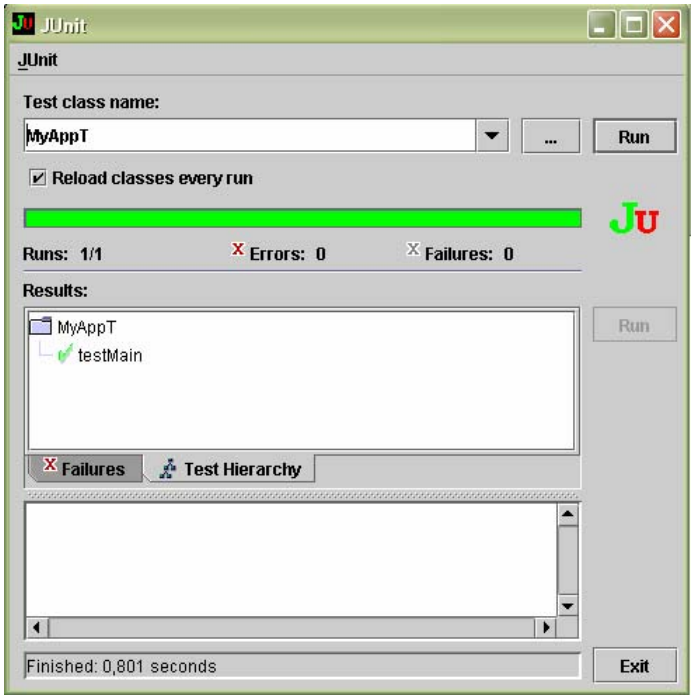


Abbildung 3.6.2 Ergebnis der Ausführung des Testfalls

Vom einfachen Testfall zu komplexeren Aufgaben

Obwohl dieser einfache Testfall den grundsätzlichen Aufbau eines JFCUnit Testfalls bereits gut illustriert, gibt er noch keinen Überblick über die tatsächlich von JFCUnit angebotene Funktionalität. Dies soll an dieser Stelle nachgeholt werden. Wie bereits zuvor erwähnt, bietet uns JFCUnit folgende Funktionalität:

- Bestimmung von aktuell geöffneten Fenster- und Dialog-Objekten
- Finden von Swing Komponenten (zB Buttons, Labels, …) in einer Komponentenhierarchie
- Simulation von Eingabeereignissen (zB Klicken auf einen Button)
- Threadsicheres Testen von Komponenten

Die ersten drei Punkte dieser Aufzählung werden über die Klasse JFCTestHelper realisiert. JFCTestHelper bietet für jeden dieser Punkte eigene Methoden an. Im Folgenden einige Beispiele zu den einzelnen Punkten:

- Bestimmung von aktuell geöffneten Fenster- und Dialog-Objekten
  ```
  List getWindows();
  //gibt eine Liste aller gerade sichtbaren Fenster zurück
  ```
- Finden von Swing Komponenten (zB Buttons, Labels, …) in einer Komponentenhierarchie
  ```
  java.awt.Component findNamedComponent(java.lang.String name, int index);
  ```

```
//sucht nach einer Komponente mit dem Namen name, index gibt an, die wievielte
//Komponente, die dem Namen entspricht, zurückgegebenen werden soll
```

- Simulation von Eingabeereignissen (zB Klicken auf einen Button)

```
void enterClickAndLeave(AbstractMouseEventData evtData);
//simuliert einen Mausklick; AbstractMouseEventData spezifiziert Details zu
//Mausklick (zB aus welchem Objekt er ausgeführt werden soll)
```

Der vierte Punkt der Aufzählung – das threadsichere Testen von Komponenten – wird vom JFCUnit-Framework in der Art und Weise sichergestellt, dass der AWT-Thread während der Ausführung der JFCUnit-Tests blockiert wird - wie es im einfachen Testfall beschrieben wurde.

Wie man schon aus diesen Beispielen für Methoden, die die Klasse JFCTestHelper anbietet, erkennen kann, ist das Erstellen von komplexen Testfällen mühsam. Aus diesem Grund gibt es seit Version 2.00 einen capture and replay Mechanismus, mit dem Testfälle aufgenommen werden können. Die aufgenommenen Testfälle werden in XML-Form abgelegt.

Das Aufnehmen von Testfällen funktioniert mit folgendem Befehl:

```
java -Djfcunit.xmlroot.record=true
    -Djfcunit.xmlroot.classname=demo.SwingSet
    -Djfcunit.xmlroot.testsuite=testcases.xml
    -classpath jfcunit.jar;SwingSet.jar;jarkarta-regexp-1.2.jar;junit.jar
      junit.swingui.TestRunner junit.extensions.jfcunit.tools.XMLRoot

//demo.SwingSet -> Swing-Applikation
//testcases.xml -> resultierende xml-Datei mit aufgenommenen Testfällen
```

Das Wiedergeben der Testfälle erfolgt mit folgendem Befehl:

```
java -Djfcunit.xmlroot.classname=demo.SwingSet
    -Djfcunit.xmlroot.testsuite=testcases.xml
    -classpath jfcunit.jar;SwingSet.jar;jarkarta-regexp-1.2.jar;junit.jar
      junit.swingui.TestRunner junit.extensions.jfcunit.tools.XMLRoot
```

Fazit

JFCUnit kann als eine echte Ergänzung zu JUnit gesehen werden und JFCUnit stellt damit einen großen Schritt in Richtung automatisiertes Testen von graphischen Benutzerschnittstellen dar. Einziger Wehrmutstropfen ist, dass sich JFCUnit nach wie vor im Umbruch befindet und die Dokumentation noch als eher mittelmäßig bis mangelhaft betrachtet werden kann.

4 Vergleich

Tabelle 4.1 gibt einen vergleichenden Überblick über die Tools.

	Marathon	**Jacareto**	**Abbot**	**Pounder**	**Jemmy Module**	**JFCUnit**
Einarbeitungsaufwand	mittel	mittel	mittel	mittel	mittel	mittel
Dokumentationsqualität	schlecht	schlecht	mittel	mittel	gut	mittel
Bedienbarkeit	gut	mittel	gut	gut	mittel	mittel
GUI	ja	ja	GUI-Scripteditor	ja	nein	nein
Aktuelle Version	0.90	0.7.08	0.13.1	0.95	2.2.4	2.06
Umfang	mittel	mittel	groß	mittel	mittel	mittel
Ausgereiftheit	mittel	schlecht	gut	mittel	mittel	mittel
Entwickler	Source Forge	C. Spannagel	Source Forge	Source Forge	NetBeans	Source Forge
Verfügbarkeit	Open Source	Open Source	Open Source	Open Source	Open Source	Open Source

Tabelle 4.1: Überblick über die Tools

Literatur

[MARA04a] http://marathonman.sourceforge.net/docs/history.html (12.11.2004)

[AANT04] http://ant.apache.org/ (12.11.2004)

[ABBO04] http://abbot.sourceforge.net/ 19.11.2004

[BENK03] http://www.ph-ludwigs-burg.de/mathematik/personal/spannagel/Diplomarbeit_Benkert_Bois.pdf 30.09.2003

[ECLI04] http://www.eclipse.org/ (12.11.2004)

[HAMM04] http://www.developer.com/java/other/article.php/1016841

[JACA04] http://www.ph-ludwigsburg.de/mathematik/personal/spannagel/jacareto/ 19.11.2004

[JAPI04] http://www.netbeans.org/download/dev/javadoc/jemmy-api/index.html 19.11.2004

[JEMM04] http://jemmy.netbeans.org/ 19.11.2004

[JEMM04a] http://jemmy.netbeans.org/downloads.html19.11.2004

[JEMM04b] http://jemmy.netbeans.org/tutorial.html 19.11.2004

[JFCU04] https://sourceforge.net/projects/jfcunit/

[JFCU04a] https://sourceforge.net/docman/display_doc.php?docid=5200&group_id=28662

[JUNI04] http://www.junit.org/index.htm

[MARA04b] http://marathonman.sourceforge.net/docs/script.html (12.11.2004)

[MARA04c] http://sourceforge.net/project/showfiles.php?group_id=46616 (12.11.2004)

[MARC04] http://www.informatik.tu-cottbus.de/~rrichter/teaching/test2004/vortraege/pounder/pounder.pdf 15.11.2004

[POUN04] http://pounder.sourceforge.net/ 15.11.2004

[POUN04a] http://sourceforge.net/project/showfiles.php?group_id=51479. 15.11.2004

[POUN04b] http://pounder.sourceforge.net/howto.php 15.11.2004

[POUN04c] http://pounder.sourceforge.net/api/com/mtp/pounder/Player.html 19.11.2004

[PYTH04] http://python.org/ (12.11.2004)

[REGE04] http://jakarta.apache.org/regexp/