Reifeprüfung

Agiles Testen von Java-Rich-UI-Anwendungen

Michael Hüttermann

Dieser Artikel möchte das Verständnis für das Testen von Rich-UI-Anwendungen schärfen. Er stellt die drei kostenlosen Werkzeuge TestNG, Fit und Jemmy vor, mit deren Hilfe Swing-Anwendungen umfassend getestet werden können. Eine Integration von Fit/Jemmy-Akzeptanztests und TestNG-Komponententests führt zu schnellen Rückkopplungen. Anhand eines Beispiels wird eine Integration dieser Werkzeuge vorgestellt, die hilft, die technische Barriere zwischen den komplementären Komponenten- und Akzeptanztests zu überwinden.

Komponenten- und Akzeptanztests

Bei den Komponententests, auch Unit-Tests genannt, schreibt der Entwickler automatisierbare Tests, ruft dabei seine eigenen fachlichen Klassen auf und stellt ein Soll-Ergebnis einem Ist-Ergebnis gegenüber. Gewöhnlich werden dabei Testklassen und fachliche Klassen in derselben Sprache geschrieben. Eine umfassende Testabdeckung ermöglicht erst Erweiterungen und Änderungen am Code-Bestand, insbesondere unfallfreie Refactorings.

Auf der anderen Seite existieren die vom Kunden (oder einem Kunden-Proxy) getriebenen und in seiner Sprache vorliegenden fachlichen Akzeptanztests. Im Gegensatz zu einem konventionellen Vorgehen der Anforderungserhebung mit Hilfe eines Lastenhefts werden die Akzeptanztests als fachliche Spezifikation der Anwendung formuliert. Das Charmante dabei ist, dass auf den Medienbruch (Spezifikation vs. Umsetzung) verzichtet wird. Die Tests können jederzeit automatisch gegen das aktuelle Verhalten der Anwendung abgeglichen werden.

Test- und Domain-getrieben

Akzeptanztests sollen sicherstellen, dass eine Anwendung das Richtige macht. Komponententests dienen dazu zu validieren, ob das Richtige dann auch tatsächlich richtig gemacht wird. Es werden durch diese komplementären Testarten nicht nur Defekte (im Design, im Code, in der Anforderung) gefunden. Viel wichtiger ist, durch frühzeitiges Testen diese Defekte überhaupt zu vermeiden.

Beide Testarten werden verzahnt mit der eigentlichen Anwendung entwickelt, durch Refactorings begleitet und treiben die Entwicklung („Test-Driven Development“, vergleiche [Ham05] und [Beck03]). Bei der Erstellung von Akzeptanztests sind ganz andere Skills gefragt als beim Schreiben von Komponententests: Der beteiligte Entwickler muss sich spätestens hier von seiner „Elfenbeinturm-Techniker-Denke“ lösen, sich sehr intensiv mit dem Kunden austauschen und die Denkweise und Business-Domäne des Kunden verinnerlichen. Eine gemeinsame Sprache von Kunde und Entwickler und eine konsequent an der Fachlichkeit austarierte Umsetzung begünstigt den Projekterfolg signifikant (vergleiche auch [Eva03]).

Motivation einer Überbrückung der Test-Barriere

Komponenten- und Akzeptanztests liegen gewöhnlich orthogonal nebeneinander. Dies gilt sowohl für die Durchführung im lokalen Arbeitsbereich eines Testers/Entwicklers als auch im Kontext eines zentralen Build-Servers. Das Komponententest-Framework TestNG (vergleiche [TestNG] und [BeSu08]) bereitet Testergebnisse in einem eigens dafür vorgesehenen Bereich in der Entwicklungsumgebung auf bzw. liefert HTML-Testberichte.

Das Akzeptanztest-Framework Fit (siehe [Fit]) auf der anderen Seite stellt Testergebnisse, die sich vom Konzept und Aufbau von denen der Komponententests unterscheiden, als HTML-Seiten dar.

Anzustreben ist eine Verankerung und Auswertung der Akzeptanztests durch das Komponententest-Framework. Dadurch stehen alle Testergebnisse (also von Komponenten- und Akzeptanztests) an zentraler Stelle in einem einzigen Format zur Verfügung. Die logischen Testgrenzen werden transparent. Eine umfassende Testauswertung und somit eine aussagekräftige Rückkopplung benötigt aufgrund eines „Single-Point-Of-Entry“ deutlich weniger Zeit und fördert somit eine agile Entwicklung.

Testisolation und Oberflächentests

Durch Simulation von Benutzer-Interaktionen und Abbildung einzelner fachlicher Bediensequenzen greifen Akzeptanztests auf die Benutzungsoberfläche zu. Dies trifft auf Komponententests gewöhnlich nicht zu. Ein suboptimal entwickelter Komponententest drückt einen Knopf auf der Benutzungsoberfläche, daraufhin wird dann z. B. eine Datenbankverbindung aufgemacht und der Test überprüft anschließend, ob auf der Oberfläche ein neuer Wert von der Datenbasis übermittelt und dargestellt wurde.

Entkoppeln Sie besser stattdessen die Benutzungsoberfläche von unteren Schichten. Testen Sie beispielsweise direkt eine Transaktionalität im Service-Layer und ersetzen Sie in Ihren Tests Zugriffe auf entfernte Ressourcen situativ durch sogenannte „Mocks“. Mock-Objekte ersetzten und simulieren Teilbereiche der Anwendung, überprüfen, ob sie analog ihrer Spezifikation genutzt wurden, und fördern die Testisolation und rasche Testzyklen. Dies ist insbesondere innerhalb der Entwicklungsumgebung von Bedeutung, denn langlaufende/langsame Tests werden seltener durchgeführt. Ein gängiges Mocking-Framework ist EasyMock (siehe [EM]).

Bei geschickter Entkopplung macht die eigentliche Darstellung auf der Oberfläche eine sehr dünne Architektur-Schicht aus, die unabhängig von der Funktionalität dahinter zu betrachten ist. Dies geht mit der architektonischen Erwägung einher, dass obere Schichten (wie die Benutzungsoberfläche) untere Schichten (beispielsweise Persistenzschicht oder Domain-Layer) „kennen“, aber nicht umgekehrt. Eine übergangslose Integration der disjunkten Testmengen aus Komponenten- und Akzeptanztests bedeutet hier also nicht, via Komponententest die Benutzerinteraktionen zu simulieren. Der Einsatz eines Komponententest-Werkzeugs zur Durchführung von funktionalen Tests ist aber durchaus denkbar, beispielsweise im Rahmen von Tests auf Web-Anwendungen/RIAs mittels Integration eines Komponententest-Frameworks und dem Werkzeug Selenium, siehe [Sel] und [Hüt08].

Die hier vorgestellte Integration ist auf einer infrastrukturellen Ebene zu sehen und bezieht sich auf die gemeinsame, zeitgleiche Durchführung beider gewöhnlich separat angestoßener und ausgewerteter Testreihen.

Die beispielhafte Anwendung

Als Beispielanwendung dient eine einfache Swing-Anwendung, die im Wesentlichen aus einer editierbaren, zweispal-

tigen Tabelle besteht (s. Abb. 1). Daneben existiert auf der Benutzungsoberfläche ein Eingabefeld, in das ein Filter-Token eingetragen werden kann. Mit dem in diesem Feld hinterlegten Wert wird der Tabelleninhalt in Echtzeit gefiltert: Es werden ausschließlich die Tabellenzeilen angezeigt, deren jeweiliger Wert der ersten Spalte mit dem Filter-Token beginnt.

Diese Filter-Funktionalität für Tabellen ist seit Java 1.6 out-of-the-box nutzbar. Wer beispielsweise auf Java 1.5 etwas Vergleichbares selbst realisieren möchte, muss durch Tests sicherstellen, dass seine Individuallösung ordnungsgemäß funktioniert. Wohlwissend, dass wir ein Java-Standard-API (im Fall von Java 1.6 oder höher) keiner Tests unterziehen müssen, möchten wir anhand des Beispiels die hier thematisierte Test-Konfiguration skizzieren.

Fit

Fit („Framework for Integrated Testing“) ist ein Akzeptanztest-Framework, mit dessen Hilfe Kunden aktiv ihre Anwendung beschreiben können. Mit diesem Specification-By-Example-Ansatz spezifizieren die Domäneexperten (beispielsweise mit Winword) die Anwendung in HTML-Syntax aus fachlicher Sicht. Der Entwickler schreibt dabei Java-Fixtures, die diese Spezifikation an die Anwendung binden. Dem Kunden bleibt die eigentliche Realisierung verborgen.

Mit dem Durchlauf einer HTML-Spezifikation erstellt Fit eine Ergebnisseite, die ebenfalls in HTML-Form vorliegt. Sie informiert über das Testergebnis, also den Abgleich zwischen dem aktuellen Verhalten der Anwendung und der Spezifikation. Fit unterstützt die Sichtbarkeit des Projektfortschritts. Jederzeit kann automatisch überprüft werden, welche Funktionalität die Anwendung aktuell anbietet.

Fit bringt von Hause aus mehrere Fixture-Basisklassen mit. Dabei bietet die `ActionFixture` eine Möglichkeit, Zustandsübergänge auszulösen und Zustandsüberprüfungen zu realisieren. Dies ist prädestiniert für unser Szenario, Interaktionen auf der Anwendung durchzuführen und Zustände zu wohldefinierten Zeiten einem Sollwert gegenüberzustellen.

Jemmy

Jemmy (siehe [Jemmy]) ist ein von Sun Microsystems implementiertes, im Rahmen des NetBeans-Projektes entstandenes Framework. Jemmy findet und steuert Swing-Komponenten zur Laufzeit auf der Benutzungsoberfläche. Auf diesem Wege können komplexe Benutzer-Interaktionen auf der Oberfläche simuliert werden. Jemmy bietet für alle Swing-Komponenten eigene Delegates an, sogenannte „Operator“, sodass beim Arbeiten mit Jemmy nicht direkt auf die Swing-Instanzen zugegriffen werden muss (obgleich das möglich ist).

Für das Setzen und Abgreifen des Inhalts bzw. die Bedienung der jeweiligen visuellen Komponente bieten die Operator Typ-sensitive Zugriffsmechanismen an. Jemmy selbst dient lediglich der Steuerung der Oberfläche. Um Aktionen auf der Oberfläche zu überprüfen, muss Jemmy in ein Test-Framework wie TestNG eingebunden werden.

Zur Identifikation der Komponenten im Objektgeflecht der gestarteten Anwendung unterstützt Jemmy unterschiedliche Mechanismen. Um die Tests nicht zu fragil zu machen, kann sich ein Zugriff auf die grafischen Swing-Controls via Label anbieten, wobei das Label sowohl in der Anwendung selbst als auch von Jemmy als technische ID genutzt und internationalisiert/lokalisiert wird.

TestNG

TestNG („Test New Generation“) ist ein Komponententest-Framework, dessen Nutzung mit der von JUnit (siehe [JUnit]) vergleichbar ist. Testmethoden werden mit `@Test` annotiert, Vor- und Nachbedingungen aufgesetzt und Assertions überprüfen Zustände. TestNG ist maßgeblich für die Weiterentwicklung von JUnit vom 3.x-Zweig hin zum 4.x-Zweig verantwortlich und hat die Java-Annotationen in die Komponententest-Welt eingeführt. TestNG besitzt die Fähigkeit, Tests zu gruppieren und Abhängigkeiten und Ausführungsreihenfolgen zwischen diesen Gruppen aufzusetzen. Es ist ferner sehr mächtig im Bereich nebenläufiges Testen, so können beispielsweise im Rahmen eines Tests von x Threads y Instanzen erstellt werden, die z mal eine Aktion durchführen.

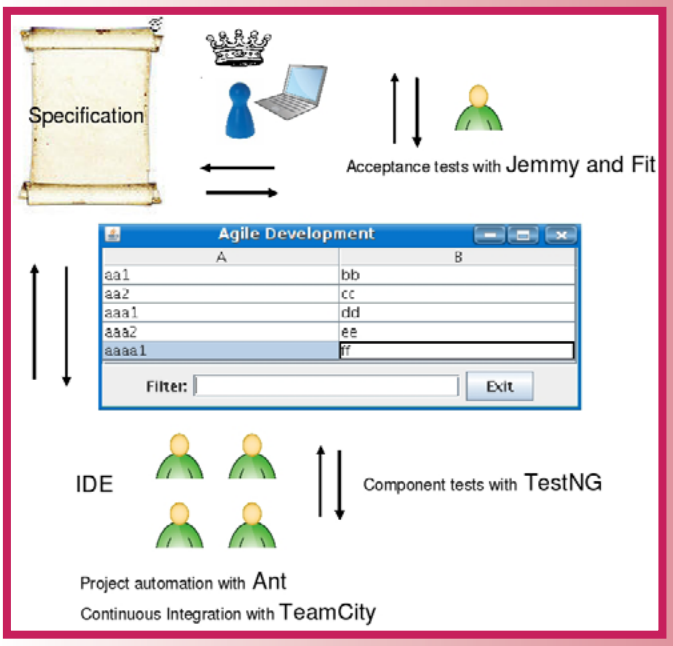

Abb. 1: Überblick über die Beispielanwendung

fit.ActionFixture		
start	com.huettermann.fit.FitTestActionFixture	
enter	fuegeEinZeile1Spalte1	Peter
enter	fuegeEinZeile1Spalte2	31
enter	fuegeEinZeile2Spalte1	Frank
enter	fuegeEinZeile2Spalte2	28
enter	fuegeEinZeile3Spalte1	Kurt
enter	fuegeEinZeile3Spalte2	12
enter	fuegeEinZeile4Spalte1	Felix
enter	fuegeEinZeile4Spalte2	66
enter	fuegeEinZeile5Spalte1	Friedrich
enter	fuegeEinZeile5Spalte2	71
check	pruefeZeile1Spalte1	Peter
enter	fuegeEinFilterText	F
check	zaehleAnzahlZeilen	3
check	pruefeZeile1Spalte1StartetMit	F
check	pruefeZeile2Spalte1StartetMit	F
check	pruefeZeile3Spalte1StartetMit	F
press	schliesseAnwendung	

Abb. 2: Fit

```
package com.huettermann.fit;

import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.RowFilter;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;

import org.netbeans.jemmy.operators.JButtonOperator;
import org.netbeans.jemmy.operators.JFrameOperator;
import org.netbeans.jemmy.operators.JTableOperator;
import org.netbeans.jemmy.operators.JTextFieldOperator;

import com.huettermann.fit.application.TableFilter;
import fit.ActionFixture;

/**
 * Fixture.
 *
 * @author Michael Huettermann
 */
public class FitTestActionFixture extends ActionFixture {
  private JFrameOperator frame;
  private TableRowSorter<TableModel> sorter;
  private JTextField filterText;
  private JTableOperator mainTable;
  private JTextFieldOperator inputField;

  public FitTestActionFixture() {
    TableFilter.main(new String[] {});
    frame = new JFrameOperator(0);

    mainTable = new JTableOperator(frame, 0);
    JTable table = (JTable) mainTable.getSource();

    //fill with test data
    TableModel model = new DefaultTableModel(5, 2);
    mainTable.setModel(model);
    //new model means new filter stuff
    // i.e. row filter and document
    //listener(supressed here)
    //...
  }

  public void fuegeEinZeile1Spalte1(String param) {
    mainTable.changeCellObject(0, 0, param);
  }

  public void fuegeEinZeile1Spalte2(int param) {
    mainTable.changeCellObject(0, 1, param);
  }

  //...
  //analog for table rows 2-4
  //...

  public String pruefeZeile1Spalte1() {
    return (String)(mainTable.getValueAt(0, 0));
  }

  public void fuegeEinFilterText(String param) {
    inputField.typeText(param);
  }

  public String pruefeZeile1Spalte1StartetMit() {
    return ((String) mainTable.getValueAt(0, 0)).substring(0, 1);
  }

  public String pruefeZeile2Spalte1StartetMit() {
    return ((String) mainTable.getValueAt(1, 0)).substring(0, 1);
  }

  public String pruefeZeile3Spalte1StartetMit() {
    return ((String) mainTable.getValueAt(2, 0)).substring(0, 1);
  }

  public int zaehleAnzahlZeilen() {
    return mainTable.getRowCount();
  }

  public void schliesseAnwendung() {
    JButtonOperator cancelButton = new JButtonOperator(frame, "Exit");
    cancelButton.push();
  }
}
```

Listing 1: Fit.ActionFixture greift mit Jemmy auf die Benutzungsoberfläche zu

```
package com.huettermann.fit;

import org.testng.Assert;
import org.testng.annotations.Test;

import com.huettermann.fit.framework.FitRunner;
import com.huettermann.fit.framework.Utils;

/**
 * Fit-Tests
 *
 * @author Michael Huettermann
 */
public class AllFitTests_Integration {

  /** fit runner */
  private FitRunner runner;

  @Test(groups={"gui"})
  public void testTable() throws Exception {
    runner = new FitRunner(Utils.getFolder(this),
                           "FitTestActionFixture");
    String result = runner.runFitTest();
    if (result != null)
      Assert.fail(result);
  }
}
```

Listing 2: Ein TestNG-Test, der auf einen Fit-Test zugreift

Ein weiteres Core-Asset von TestNG ist das umfangreiche Reporting der Testergebnisse. TestNG hat ferner eine offene Architektur. Es kann einerseits recht einfach erweitert werden, andererseits lässt es sich ausgezeichnet mit JUnit verheiraten (beispielsweise durch Integration von JUnit-Tests oder durch eine Transformation der Reports in einen JUnit-Report-Style).

Unsere fachliche Spezifikation

Wie sieht die Nutzung von Fit in unserem Szenario aus? Unsere HTML-Tabelle zur Anwendungsspezifikation basiert auf diversen Fit-Konventionen. So muss für unser Beispiel die erste Tabellenzeile (s. Abb. 2) fixieren, dass wir eine **`ActionFixture`** schreiben möchten. Diese **`ActionFixture`** hinterlegen wir namentlich in der zweiten Zeile. Die weiteren Zeilen beschreiben die Zustandsübergänge. Wir geben das Wort „Peter“ in eine Tabellenzelle ein. Der Kunde sucht sich den Namen für diese Aktion aus, hier „fuegeEinZeile1Spalte1“. Es ist nun die Aufgabe des Entwicklers, diese Anforderungsausprägung umzusetzen, das heißt eine gleichnamige Methode in der Java-Fixture zu entwickeln und die Aktion durch Aufruf der geeigneten Funktionen auf die Applikation anzuwenden.

Nachfolgend soll die Tabelle gefüllt werden, jeweils mit Vornamen in der ersten Spalte und Zahlen in der zweiten. Im weiteren Verlauf führen wir eine erste Überprüfung durch

(„check“). Wir geben nun den Buchstaben „F“ in unser Filtertextfeld ein. Da wir bis hierhin drei Vornamen erfasst haben, die mit einem „F“ beginnen, soll die Tabelle nun drei Positionen besitzen („Frank“, „Felix“, „Friedrich“). Um auch tatsächlich gegenzuprüfen, dass die nun sichtbaren Zeilen mit „F“ beginnen, prüfen wir auch dies. Anschließend möchten wir mit „Press“ das Schließen der Anwendung auslösen.

Sobald die Fixture fertig (besser: lauffähig) ist, kann Fit auf der Tabelle laufen gelassen werden. Die HTML-Ergebnisseite beinhaltet die Ausgangsinformationen und farblich hinterlegte Zellen: Sind Spezifikation und Anwendungsverhalten synchron, wird dies mit grüner Farbe symbolisiert (wie in unserem Fall), ansonsten kommt Rot zum Einsatz.

Die Java-Fixture und Jemmy

Listing 1 zeigt die Fixture-Klasse, die die Beispielanwendung an die HTML-Spezifikation leimt. Sie leitet von `fit.ActionFixture` ab und startet im Konstruktor die zugrundeliegende Anwendung. Das Frame und die Tabelle werden von Jemmy identifiziert. Dabei kommen die Operator `JFrameOperator` und `JTableOperator` zum Einsatz. Die mehrfach überladenen Zugriffsmethoden bieten die Möglichkeit, eine Komponente im Objektbaum ausgehend von einer anderen zu suchen.

Im weiteren Verlauf wird für jede HTML-Zeile (`press`, `enter`, `check`) eine Methode implementiert, die via Jemmy den Inhalt von Komponenten setzt, zurück gibt oder die Komponente bedient (im Fall des Buttons). Dabei kommen Fit-Konventionen zum Einsatz. So werden beim Setzen von Inhalt (`enter`) Methoden geschrieben, die über einen Parameter verfügen und keinen Rückgabewert besitzen. Fehlerhafte Ausführungen (z. B. durch nicht beachtete Fit-Konventionen) werden im HTML-Outputdokument gesprächig dokumentiert.

Integration in TestNG

Neben Ihren bereits vorliegenden Komponententests schreiben Sie eine neue TestNG-Testklasse, die pro HTML-Spezifikation eine Testmethode besitzt. In Listing 2 ist eine solche Testklasse dargestellt. Der Name der Klasse endet auf „_Integration“ und suggeriert, dass diese Tests architektonisch mehrere Ebenen überbrücken. Die Testmethode wird der Gruppe „gui“ zugeordnet. Auf diese Zuordnung können wir beim Teststart zurückkommen und nur die Tests anstoßen, die dieser Gruppe zugeordnet sind. Im Test rufen wir einen `FitRunner` (s. u.) auf, dem wir im Wesentlichen die Koordinaten der Test-Spezifikation mitteilen. Wir starten den `FitRunner` und lassen den Test scheitern (`fail`), falls die Rückgabe ungleich `null` ist.

Wie feingranular Sie die Tests hinterlegen, bleibt dabei Ihren individuellen Bedürfnissen vorbehalten. So könnten Sie Ihre Tests (bzw. den `TestRunner`) so anpassen, dass alle HTML-Seiten in einem Verzeichnis ausgelesen und verarbeitet werden, statt die einzelnen HTML-Spezifikationen namentlich einzeln zu übergeben. Denkbar ist auch, einen TestNG-`@DataProvider` einzusetzen, der eine Testmethode mit jeweiligen HTML-Dokumenten parametrisiert.

An Fit andocken

Was versteckt sich hinter dem `FitRunner`? Dies ist ein von uns entwickeltes, wiederverwendbares Modul, das an das Fit-Framework andockt und dabei neben dem Namen der ausführbaren Spezifikation (das Eingangs-HTML) auch den Namen des Ausgangs-HTML empfängt. Das Eingangsdokument wird bei uns im Package `FitTestSpec` gesucht, das Ausgangsdokument ins Package `FitTestResult` gelegt. Fit führt mit seinem `FileRunner` die übergebene Spezifikation aus und wird das Ausgangs-HTML erstellen. Der Testerfolg wird programmatisch ausgewertet und als `result` an den aufrufenden TestNG-Test zurückgegeben.

```
/**
 * FitRunner. Glue. Connects the Fit library runner with the
 * test application.
 *
 * @author Michael Huettermann
 */
public String runFitTest() throws IOException {
  String result = null;
  FileRunner runner = new FileRunner();

  runner.args(new String[] {inputFileName, outputFileName});
  runner.process();
  runner.output.close();
  if (runner.fixture.counts.wrong +
      runner.fixture.counts.exceptions > 0) {
       result = "" + runner.fixture.counts.wrong +
       " errors and" + " " +
       runner.fixture.counts.exceptions + " exceptions for "
       + fitTestName + ". See output " + outputFileName;
  }
  return result;
}
```

Listing 3: Rahmenwerk: Ein Fit-Adapter

Listing 3 zeigt das Herz des Moduls, das für den Teststart verantwortlich ist und das Testergebnis zurückliefert.

Integration in Eclipse

Die komfortable Nutzung von TestNG ist nach Aufrüstung Ihrer Eclipse-Installation mit dem TestNG-Plug-In (über [TestNG] beziehbar) möglich. Die Testläufe lassen sich über das Plug-In umfassend konfigurieren, beispielsweise welche Test-Gruppen Sie laufen lassen möchten. Ein TestNG-View liefert Ihnen Informationen zum Testergebnis.

Abbildung 3 zeigt unseren Test `testTable`, der erfolgreich durchlaufen wurde (grüner Balken sowie numerischer Verweis auf einen erfolgreich durchgeführten Test). Daneben wird ein umfangreicher HTML-Report ins Verzeichnis geschrieben. Ihre Fit-Tests werden dabei technologisch nicht von anderen TestNG-(Komponenten-)Tests unterschieden. Per Konvention wissen Sie, dass es sich hier um einen Fit-Test handelt, beispielsweise weil der Test im Package `AllFitTests_Integration` liegt. Die eigentliche Technik hinter den Fit-Tests ist auf dieser Ebene vollkommen transparent.

Kontinuierliche Integration

Sie sollten die Tests nicht ausschließlich über die Entwicklungsumgebung, z. B. Eclipse, startbar vorhalten. Im Rahmen einer kontinuierlichen Integration (CI, [Fow06]) möchten Sie die Tests schließlich, unabhängig von Entwicklungsumgebungen, in wohldefinierten Abständen automatisiert laufen lassen. Dazu kann ein Ant-Skript dienen, das nach Erzeugung eines sauberen Ausgangszustands (Neukompilierung der fachlichen Klassen und Testklassen) alle Komponententests (und somit die Fit-Tests) laufen lässt.

Dieses Ant-Skript ist Bestandteil des Projekts und als solches ebenfalls Teil der Menge der Konfigurationselemente im Versionskontrollsystem, wie es die Tests selbst auch sind. Das Ant-Skript können Sie einerseits aus Ihrer Entwicklungsumgebung anstoßen, andererseits von einem CI-Server (wie beispielsweise TeamCity [TC]) aufrufen lassen.

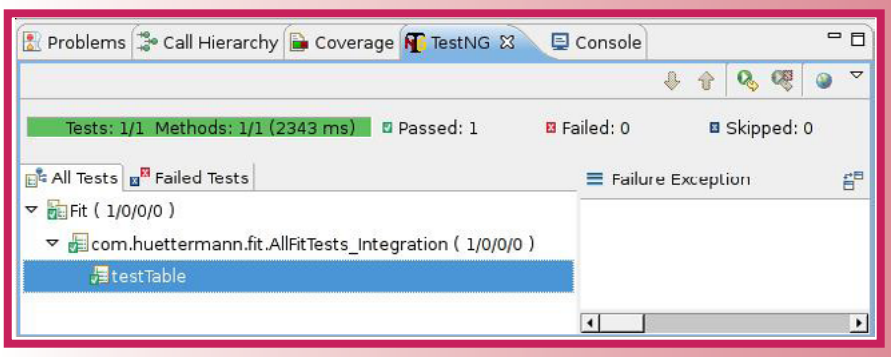

Abb. 3: Testauswertung in Eclipse

Zusammenfassung

Java auf dem Client erlebt nicht erst seit dem Java SE Update N oder den JSRs „Beans Binding API“ und „Swing Application Framework“ eine Renaissance. Je komplexer Ihre Fachlichkeit bzw. die Anwendung desto mehr werden das Schreiben von Komponenten- und Akzeptanztests zur Pflicht. Die vorgestellte Konfiguration zeigte einen Weg auf, Ihre mit TestNG geschriebenen Komponententests und die mit Fit und Jemmy implementierten Akzeptanztests zu einer reibungslosen technischen Gesamtlösung zu integrieren. Durchlaufzeiten und manuelle Kontextwechsel werden reduziert und die Qualität insgesamt erhöht.

Obwohl der Autor mit der beschriebenen Zusammenstellung sehr gute Erfahrungen gemacht hat, sind die Facetten der Gesamtkonfiguration durchaus austauschbar: Die Entwicklungsumgebung IntelliJ IDEA besitzt von Hause aus eine TestNG-Unterstützung. Bei entsprechender Präferenz könnten Sie auch TestNG durch JUnit oder Jemmy durch FEST (siehe [FEST]) ersetzen.

Die vollständige Beispielkonfiguration (inkl. Ant-Skript, Fit-Adapter und exemplarischer Tests) ist im Web frei verfügbar (siehe [HütBsp]).

Literatur und Links

[Beck03] K. Beck, Test-Driven Development, Addison-Wesley, 2003
[BeSu08] C. Beust, H. Suleiman, Next Generation Java Testing, TestNG and Avanced Concepts, Addison-Wesley, 2008
[EM] Mocking-Framework EasyMock, `http://www.easymock.org`
[Eva03] E. Evans, Domain-Driven Design, Addison-Wesley, 2003
[FEST] Java UI Testing Bibliothek FEST (Fixtures for Easy Software Testing), `http://fest.easytesting.org/`
[Fit] Akzeptanztest-Framework Fit (Framework for Integrated Testing), `http://fit.c2.com`
[Fow06] M. Fowler, Continuous Integration, 2006, `http://martinfowler.com/articles/continuousIntegration.html`
[Ham05] T. Hammell, Test-Driven Development, A J2EE Example, Apress, 2005
[Hüt08] M. Hüttermann, Agile Java-Entwicklung in der Praxis, O‘Reilly, 2008
[HütBsp] Das komplette Beispiel online, `http://huettermann.net/agilejavaentwicklung/FitJemmyTestNG.zip`
[Jemmy] Java UI Testing Bibliothek Jemmy, `http://jemmy.netbeans.org`
[JUnit] Komponententest-Framework JUnit, `http://junit.org/`
[Sel] Framework zum Testen von RIAs, Selenium, `http://www.openqa.org/selenium/`
[TC] CI-Server TeamCity, JetBrains, `http://www.jetbrains.com/teamcity/`
[TestNG] Komponententest-Framework TestNG, `http://testng.org`

Sun Java Champion **Michael Hüttermann** ist freiberuflicher Experte für Java/JEE und agile Entwicklung. In seinem Buch „Agile Java-Entwicklung in der Praxis“ beschreibt er Methodik und Infrastruktur der agilen Entwicklung mit Java.
E-Mail: michael@huettermann.net.