Wie von selbst

Automatisierte Integrationstests

Michael Kloss, Steffen Stundzig

Die Entwicklung von Anwendungen, welche verschiedene Backendsysteme integrieren, ist vielfach erläutert und beschrieben. Nach den ersten Hypes sind der Entwicklungsprozess und die Werkzeuge mittlerweile recht stabil und gut entwickelt. Ein Schwachpunkt ist meist der automatisierte Test solcher integrierten Anwendungen. Häufig gibt es für verschiedene Systeme auch verschiedene Testumgebungen. Im vorliegenden Artikel wird eine Testinfrastruktur basierend auf CruiseControl, FitNesse und Selenium beschrieben, die eine Möglichkeit der Realisierung solcher Tests darstellen kann.

Nach einem gelungenen Workshop u. a. zum Thema Continuous Integration mit CruiseControl [CC] wurde im Entwicklerteam überlegt, wie Tests während des Entwicklungsprozesses automatisiert ausgeführt werden können. Es gab bereits einige Methodentests mit JUnit, jedoch waren diese für die Komplexität der Anwendungsarchitektur der Softwaresystemfamilie nicht ausreichend.

Bei den Anwendungen dieser Familie handelt es sich um Java EE-basierte Webanwendungen, die u. a. auf JBoss als Applikationsserver, Hibernate als OR-Mapper und Quartz für zeitgesteuerte Aktionen implementiert sind. Die Anwendungen haben einen sehr starken integrativen Anteil durch die Nutzung verschiedener Backendsysteme. Sie kommunizieren beispielsweise per SAPJCO [JCO] mit SAP oder per Webservices mit Flottennavigations- und Routingsystemen, versenden SMS an mobile Einheiten im Feld, laden E-Mails von Exchangeservern, holen Grafiken aus geografischen Informationssystemen usw.

Ein Großteil der Aktionen des Systems wird allerdings durch Anwender über die Weboberfläche angestoßen. Die Weboberfläche selbst ist mit dem quelloffenen, Ajax-basierten Webframework Echo2 realisiert [Echo2]. Unabhängig von den Anwendern der Weboberfläche sollten auch die Entwickler bereits während der Entwicklungsphase die Möglichkeit bekommen, die Tests ohne große Umstände gegenüber ihrem eigenen Applikationsserver auf ihrem Arbeitsplatz auszuführen.

Momentan ist die Testabdeckung in einem Großteil der Anwendungen recht niedrig. Dies liegt zum Teil daran, dass die bisher propagierten JUnit-Tests per Hand programmiert werden müssen, inklusive Laden von Konfigurationen usw. Dafür gibt es zwar eine gute Werkzeugunterstützung innerhalb der Java-Welt, trotzdem macht es einigen Aufwand.

Sinnvoll erschien auch die Generierung von Testprotokollen, die gegenüber dem Kunden als Bestandteil der Abnahme der Software präsentiert werden können und damit auch als eine Art Vertrag gelten können.

Was war zu tun

Aus den Überlegungen ließen sich nun folgende Anforderungen für die zu schaffende Testinfrastruktur ableiten:

- Tests von Ajax-basierten Webanwendungen, möglichst mit verschiedenen Browsern (Mozilla Firefox und Microsoft Internet Explorer),
- Tests gegen die Backendsysteme, möglichst in Kombination mit den Webtests,
- einfache Sprache für die Tests, möglichst ohne die Notwendigkeit von Programmierkenntnissen,
- automatisiertes Ausführen der Tests,
- zentrale Verwaltung von Konfigurationen und Tests,
- kundentaugliche Präsentation der Testergebnisse.

Basierend auf dieser Anforderungsanalyse startete eine Suche nach Werkzeugen, möglichst aus dem quelloffenen Bereich, die diese Anforderungen weitestgehend abdecken.

Eine einfache Sprache

Als eine Umgebung, in der ohne großen Lernaufwand verschiedene Personengruppen schnell halbwegs strukturierte Texte erzeugen können, hat sich mittlerweile ein Wiki mit der entsprechenden Wikisyntax durchgesetzt. Dabei wird bewusst auf grafische Spielereien oder umfangreiche Layoutmöglichkeiten verzichtet. Der Fokus liegt klar auf den Inhalten.

Es wurde also ein Wiki gesucht, mit dem Tests formuliert werden können. Und exakt diese Nische besetzt das quelloffene Werkzeug Fit [FIT] und dessen Erweiterung FitNesse [FITN].

Fit, Framework For Integrated Tests, ist ein Javatool von Ward Cunningham, mit dem auf einfache Weise Tests geschrieben werden können. Testfälle für das Fit-Framework werden einfach als HTML-Tabellen geschrieben, welche die Ein- und erwarteten Ausgangswerte enthalten. Zur Auswertung und Umsetzung dieser Testfälle setzt Fit auf das Adapter-Muster und bietet als Basis einen zugeschnittenen HTML-Parser, welcher aus HTML-Tabellen relevante Daten ausliest. Diese Daten werden anschließend als Variablen per Java Reflection in entsprechenden Fixtures, in Java implementierte Adapterklassen, gesetzt oder die Methoden der Fixture werden ausgeführt. Dies führt zu der komfortablen Situation, dass die Testsprache vollkommen losgelöst vom eigentlichen System ist. Nur die Adapterklassen bilden die Schnittstelle. Der große Vorteil dieser Variante ist, dass keine anwendungsbezogene Nutzeroberfläche benötigt wird, um Anwendungen zu testen. Die Testfälle selbst bilden hier den Ersatz. Im Idealfall wird beim Einsatz von Fit die Testabteilung ihre eigene Testsprache definieren und mit dieser gegen die vom Team entwickelte Anwendung testen.

Eine Weiterführung von FIT ist FitNesse. FitNesse ist ein Wiki, in dem die Testsuiten und Testfälle in einfacher Wiki-Schreibweise erstellt, verwaltet und ausgeführt werden kön-

Abb. 1: Test im FitNesse

nen. Es bietet eine komplette Plattform für die Verwaltung der Tests. Außerdem bietet es zusätzliche Funktionen, wie das inkludieren von anderen Tests, symbolische Links auf andere Seiten im Wiki und Symbole (Variablen), um nur ein paar der Erweiterungen zu nennen. Abbildung 1 zeigt einen solchen Test im FitNesse. Dieser enthält hier zusätzlich noch weitere Dokumentation für den Nutzer des Tests.

FitNesse kann so konfiguriert werden, dass es einen zentralen FitNesse-Server gibt, für beispielsweise die automatisierten Tests, und eine lokale Instanz pro Entwicklerarbeitsplatz. Die Testfälle werden auf dem zentralen FitNesse gepflegt und automatisch in die lokalen Instanzen importiert. Somit ist gewährleistet, dass die Entwickler zügig Tests erstellen und auch ausführen können. Andererseits stehen diese Tests oder Testfragmente parallel allen Entwicklern und Testern zur Verfügung. Zur Unterstützung dieser zentralen Bearbeitung besitzt FitNesse eine eingebaute Versionsverwaltung.

Webtests

Aufgrund der Nutzung von Ajax in der Weboberfläche und dem Wunsch nach verschiedenen Browsern schieden viele Frameworks wie beispielsweise HttpUnit oder WebUnit aus. Als tatsächlich tauglich für diese Anforderungen blieb nur das quelloffene Selenium, von [SELE], übrig.

Das Testframework Selenium ist ursprünglich 2004 bei Thoughtworks entstanden und wurde 2005 an die Open-Source-Community gegeben. Ziel des Projekts ist es, Webanwendungen möglichst einfach zu testen, ohne unnötigen Overhead zu erzeugen. Im Laufe des letzten Jahres haben sich folgende Teilprojekte herauskristallisiert:

- Selenium Core,
- Selenium IDE,
- Selenium RemoteClient,
- Selenium on Rails (wird hier nicht näher betrachtet).

Wie der Name Selenium Core schon andeutet, handelt es sich hierbei um das Kernstück von Selenium. Dieser Teil des Frameworks ist vollständig einsetzbar und besitzt einen eigenen TestRunner. Dieser muss mit der zu testenden Webanwendung auf demselben Webserver installiert sein. Grund dafür ist die JavaScript-Sicherheitsbeschränkung SameOriginPolicy. Da dieser TestRunner von einigen sicherheitsrelevanten Funktionen Gebrauch macht, die diese Policy verletzen würde, ist diese Einschränkung mit dem reinen Selenium Core unabdingbar.

Der TestRunner, siehe Abbildung 2, besteht aus den folgenden vier Teilen:

- der Testsuite,
- dem aktuellen Test,
- einem Controlpanel,
- einer Live-Ansicht der Webanwendung unter Test.

Abbildung 2 zeigt zugleich ein für die Akzeptanz sehr wichtiges Feature von Selenium: Während der Ausführung einer Testsuite oder eines Tests ist direkt das Verhalten der Anwendung zu sehen. Außerdem sind während des Tests sogar direkt Eingriffe in den Ablauf möglich. Gerade auch bei der Erstellung der Tests ist es beispielsweise sehr vorteilhaft, Breakpoints in den Tests zu setzen.

Der wirkliche Vorteil von Selenium besteht aber in der Beschreibung der Testfälle. Diese werden, ähnlich wie bei FitNesse, in HTML-Tabellen beschrieben. Allerdings gibt es bei Selenium bereits ein umfangreiches Set von Kommandos, auf das sehr bequem zurückgegriffen werden kann. Im Gegensatz zu Fit ist es hier nicht mehr notwendig, eigene Adapter zu implementieren. Dies ist auch einer der wesentlichen Vorteile gegenüber den anderen bekannten Testwerkzeugen wie HttpUnit oder WebUnit. Bei diesen muss immer noch Java programmiert werden, um Tests zu erstellen.

Selenium-Testfälle bestehen, wie schon erwähnt, aus HTML-Tabellen. Diese bestehen aus genau drei Spalten, wobei Spalte 1 immer den so genannten Selenese-Befehl enthält, Spalte 2 den ersten Parameter und Spalte 3 den zweiten Parameter, siehe Tabelle 1.

Ob ein oder zwei Parameter übergeben werden müssen, hängt lediglich von dem aufgerufenen Kommando ab. Das Beispielkommando `open` aus Tabelle 1 bekommt lediglich einen Parameter, das Kommando `type` hingegen zwei. Zum Einen den so genannten ElementLocator und zum Anderen den dort einzutragenden Wert. Eine Liste von ElementLocatoren enthält

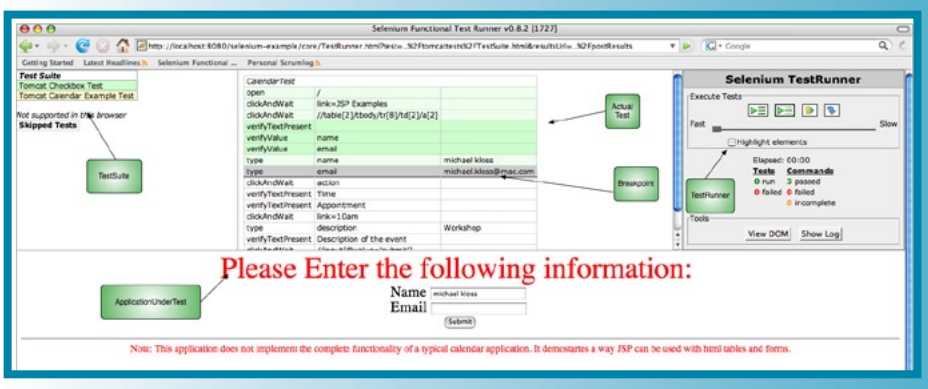

Abb. 2: TestRunner

Kommando	Parameter 1	Parameter 2
open	/meineAnwendung	
type	loginUserId	meinUser

Tabelle 1: Selenese-Befehle

ElementLocator-Type	Beispiel
DOM	document.forms[,myForm'].myInput
XPath	//a[@href = ,/mylink.html']
ID	myId
Name	myName
Link	link=Link Beschreibung
css	css= a[href="#myId"]

Tabelle 2: Aufruf von Selenese-Befehlen

Tabelle 2. Die Referenzseite in der Dokumentation von Selenium [SELR] bietet hier mehr als genug Unterstützung.

Erwähnenswert ist noch, dass es zu fast jedem Kommando die Möglichkeit gibt, ein `waitFor` aufzurufen. Dies ist besonders hilfreich bzw. sogar notwendig bei Ajax-basierten Anwendungen. Bei diesen wird bekanntlich, siehe [AJAX], nach einer Aktion innerhalb eines gewissen Zeitraumes ein Ereignis zurück erwartet, die gesamte Seite aber nicht neu geladen. Die `waitFor`-Befehle akzeptieren zusätzlich Parameter für eine Zeitüberschreitung, die begrenzt, bis wann spätestens eine Rückmeldung erfolgen muss.

Selenium ist ein JavaScript-Framework. Und wie sooft bei Frameworks, ist vielleicht nicht alles zu finden, was benötigt wird. Auch an dieser Stelle wartet Selenium mit extrem gutem Stil auf. Es gibt standardmäßig eine Datei `user-extensions.js`, in welcher eigene Erweiterungen abgelegt werden können. Dies ist auch nicht besonders schwierig: Um ein eigenes Kommando zu schreiben ist nur Listing 1 notwendig. In einem Test wird dieses Kommando wie in Tabelle 3 aufgerufen.

Das Selenium-IDE-Projekt beschäftigt sich mit der Entwicklung einer Erweiterung für den Mozilla Firefox. Mit dieser ist es möglich, Testfälle für Selenium mittels „Capture & Replay"-Verfahren aufzuzeichnen. In der neuesten Version ist diese IDE sogar so weit entwickelt, dass selbst ein manuelles Schreiben von Tests mit der IDE als Editor sehr komfortabel geworden ist. Die IDE verfügt inzwischen über eine Autovervollständigung und Auswahl der Selenese-Befehle, sowie eine eingebaute kontextsensitive Hilfe zu dem gerade gewählten Kommando. Die mit der IDE aufgezeichneten Tests lassen sich standardmäßig als HTML-Tabelle abspeichern, aber auch als Java-, Python-, C#-, Perl- oder Ruby-Datei, um diese direkt für den Selenium RemoteClient zu nutzen.

Abbildung 3 zeigt die Benutzung der Selenium IDE bei der Aufzeichnung eines Webtests gegen die Suchmaschine Google. Neben der Aufzeichnung der gedrückten Links und Schaltflächen, bietet die IDE über Kontextmenüs die Möglichkeit, Tests gegen Textbausteine oder Tests auf Vorhandensein bestimmter HTML-Elemente per Klick zu erfassen.

Da die IDE derzeit noch einige kleine Probleme bei der Aufzeichnung von Ajax-Sessions hat, konnte diese im Weiteren leider nicht verwendet werden. Zukünftige Versionen der IDE werden diese Mängel aber sicher ausräumen.

```
Selenium.prototype.doMeinKommando = function (param1, param2) {
  // here comes your Javascript
});
```

Listing 1: MeinKommando

meinKommando	param1Value	param2Value
type	id	...

Tabelle 3: Übersicht der ElementLocator

Unterstützte Browser	Windows	Linux	Mac OS X
Firefox	x	x	x
Internet Explorer	x	–	–
Opera	x	x	–
Camino	–	–	(x)
Konqueror	–	x	–
Safari	–	–	(x)
Mozilla Suite	(x)	(x)	(x)
Seamonkey	(x)	–	(x)

x = Getestet, (x) = Sollte laufen, – = nicht unterstützt

Tabelle 4: Aufruf eines eigenen Selenese-Befehls

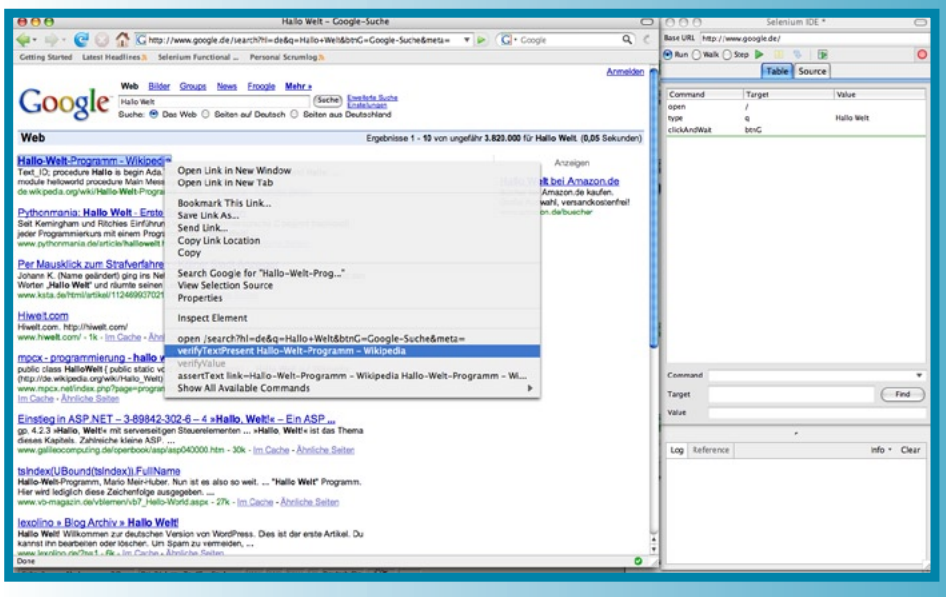

Abb. 3: Selenium IDE

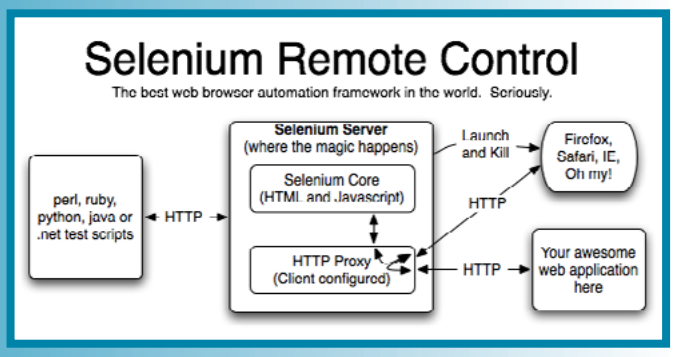

Abb. 4: SeleniumRC

```
java -jar selenium-server.jar -htmlSuite "*firefox"
  "http://www.google.com" "c:\absolute\path\to\my\HTMLSuite.html"
  "c:\absolute\path\to\my\results.html"
```

Listing 2: SeleneseRunner

Der Selenium RemoteClient, kurz RC, dient dazu, eine Selenium-Testsuite auch auf entfernten Rechnern ausführen. Der RC besteht aus einem Server, welcher Browser automatisch starten, stoppen und kontrollieren kann, siehe Tabelle 4. Hierbei wird dem Server mitgeteilt, welcher Browser benutzt werden soll, und welche Tests ausgeführt werden sollen. Dies ist im einfachsten Fall über eine interaktive Konsole, bis hin zu Testsuites in HTML-Form oder Quellcode einer Programmiersprache, welche HTTP-Requests absetzen kann (z. B. Java, Ruby, Perl, Python, .NET), möglich. Der RemoteClient startet den eingebetteten Webserver Jetty [JETT], der als HTTP-Proxy fungiert. Dadurch wird auch die oben bereits genannte `SameOrigin Policy` umgangen.

Die Tests und JavaScript-Kommandos von Selenium laufen nun im Jetty ab und gehen gegen den Proxy, siehe Abbildung 4. Allerdings sollte hier Vorsicht walten, denn die erste URL, die vom Test geöffnet wird, bestimmt die `Origin` für das JavaScript.

Um eine Testsuite mit dem Selenium RC gegen eine Anwendung laufen zu lassen, wird der so genannte SeleneseRunner benutzt. Dieser wird ähnlich wie der Server gestartet, allerdings mit ein paar mehr Optionen, siehe Listing 2.

Eine andere Möglichkeit, aus seinen vielleicht schon vorhandenen HTML-Tests andere RC-Tests zu generieren, bietet die IDE. Diese kann die HTML-Dateien laden und in einer anderen RC-Sprache exportieren.

Integrierte Anwendungen verlangen integrierte Tests

Wie spielen die genannten Werkzeuge nun zusammen? Ein weiterer großer Vorteil in der täglichen Nutzung von FitNesse ist die leichte Erweiterbarkeit. Für jedes neue System, gegen das getestet werden soll, muss nur eine Adapterklasse programmiert werden, und für die gebräuchlichsten Anwendungsfälle gibt es bereits vorgefertigte Fixtures, beispielsweise:

- webservicefixture,
- jdbcfixture,
- junitfixture.

Ergänzt wurde diese Liste noch um:

- seleniumfixture,
- pdfunitfixture,
- webservicefixture.

```
public class GenericSeleneseAdapter extends ColumnFixture {
/**
 * Selenese command.
 */
  private String command;
  /**
   * Argument string. Arguments are &-separated.
   */

  private String argument;

  @Override
  public void doCell( Parse content, int pos ) {
    if(pos == 1){
      argument = content.text();
    }
    super.doCell(content, pos);
  }

  public String argument() {
    if (SeleniumRcServerAdapter.HTTPCMD == null) {
      return "SeleniumRcServerAdapter.HTTPCMD == null";
    }
    try {
      String ret = SeleniumRcServerAdapter.HTTPCMD
        .doCommand(command, argument.split("&"));
      if ("OK".equals(ret)) {
        return argument;
      } else {
        return ret;
      }
    } catch (SeleniumException e) {
      return "Exception: " + e.getMessage();
    }
  }
}
```

Listing 3: GenericSelenseAdapter.java

Wie einfach die Implementierung einer solchen spezifischen Fixture ist, zeigt Listing 3. Dieser Adapter muss nichts weiter tun, als die bekannten Selenese-Befehle direkt an einen Selenium RC zu delegieren und dessen Antwort an FitNesse zurückzugeben. Durch diese Verbindung ist es möglich, im Wiki Tests zu schreiben, die anschließend auf einem ganz anderen als dem FitNesse-Server per Selenium RC ausgeführt werden.

Diese Flexibilität durch die Fixtures ist eine große Stärke. Somit ist es sehr leicht möglich, innerhalb eines Tests zuerst eine Weboberfläche zu bedienen, anschließend Daten in einer Datenbank abzufragen oder diese dort auch initial einzufügen und abschließend eventuell noch einen Webservice aufzurufen, um Ereignisse von außerhalb zu simulieren.

Mit dieser einfachen Fixture ist es bereits möglich, alle Vorteile der beiden Systeme FitNesse und Selenium zu vereinen. Man erhält eine definierte Ablaufumgebung für die Tests und die Möglichkeit, per Selenium RC mit verschiedenen Browsern die Weboberfläche zu testen.

Durch die Hinzunahme weiterer Fixtures in den Test ist die Integration der verschiedenen zu testenden Artefakte bzw. Systeme, wie Datenbank, Weboberfläche, SAP RFC, überhaupt kein Problem mehr

Das Ziel, Tests für integrierte Anwendungen zu schreiben, wird damit erreicht und das einfache Schreiben der Tests optimal unterstützt.

Automation

Die Tests im FitNesse sind für alle Entwickler zentral nutzbar. Jeder Einzelne kann jederzeit einen Test oder eine Suite gegen

```
java -cp $PATH_TO_FITNESSE_JAR$ fitnesse.runner.TestRunner
  [options] $HOST$ $PORT$ $TEST_SUITE$
```

Listing 4: TestRunner

```
<target name="FitNesseTests">
  <exec executable="java" failonerror="true">
    <arg value="-cp"/>
    <arg value="${fitnesse.path}/fitnesse.jar"/>
    <arg value="fitnesse.runner.TestRunner"/>
    <arg value="${wikihost}"/>
    <arg value="${wikiport}"/>
    <arg value="${testsuite}"/>
    <arg value="-html"/>
    <arg value="${basedir}/artifacts/fitnesseresults.html"/>
  </exec>
</target>
```

Listing 5: Ant-Skript

seine eigene individuelle Konfiguration der Anwendungslandschaft oder gegen die zentrale Konfiguration ausführen. Das reicht aber natürlich nicht aus.

Ein klares Ziel war die automatische Ausführung der Tests. An dieser Stelle kommt das oben bereits kurz erwähnte Continuous Integration-Framework CruiseControl wieder ins Spiel. Die spezifischen Adapter für FitNesse werden von den Entwicklern ins Versionskontrollsystem, hier CVS, eingecheckt. CruiseControl überwacht diese Aktionen, kompiliert bei Bedarf die eingecheckten Adapterklassen und installiert diese im Klassenpfad des zentralen FitNesse. Dadurch stehen unmittelbar danach allen Nutzern des FitNesse die neuen Adapter zur Verfügung.

FitNesse verfügt über den konsolenbasierten TestRunner **`fitnesse.runner.TestRunner`**. Der Konsolenbefehl dazu ist in Listing 4 dargestellt. Über die Optionen kann das Ausgabeformat **`RAW (-result <filename | ,stdout‘>)`** oder **`HTML (-html <filename | ,stdout‘>)`** festgelegt werden. Der Debug-Modus wird mit dem Schalter **`-debug`** aktiviert. Mit den Parametern $HOST$, $PORT$ und $TEST_SUITE$ wird die URL der Testsuite/Testseite angegeben. Der Rückgabewert des TestRunner entspricht der Anzahl der aufgetretenen Fehler. Ein Wert von 0 symbolisiert ein erfolgreiches Absolvieren aller Tests.

Der übergebene Klassenpfad wird zur Laufzeit durch den TestRunner erweitert. Alle Pfadelemente in den Wikiseiten, siehe Schlüsselwort **`!path`**, werden dem Klassenpfad hinzugefügt. Handelt es sich dabei um absolute Pfadangaben, ist ein Aufruf des Tests aus jedem Verzeichnis des FitNesse-Servers möglich. Für relative Pfadangaben ist es erforderlich, dass der TestRunner aus dem Arbeitsverzeichnis von FitNesse gestartet wird.

Wird der Aufruf des TestRunner in ein Ant-Skript eingebettet, siehe Listing 5, kann dieser auch völlig problemlos aus CruiseControl heraus initiiert werden. Die Eigenschaft **`failonerror`** steuert das Verhalten im Falle eines Fehlers. CruiseControl stellt verschiedene Verfahren zur Veröffentlichung der Ergebnisse eines solchen Aufrufs zur Verfügung. Die so genannten Publisher können die Ergebnisse, welche im Ordner für Artefakte landen, beispielsweise per E-Mail versenden oder in die Webansicht eines Projekts integrieren.

Somit ist sowohl die automatisierte Ausführung der Tests als auch das Aktualisieren der eigentlichen Testinfrastruktur durch CruiseControl sichergestellt.

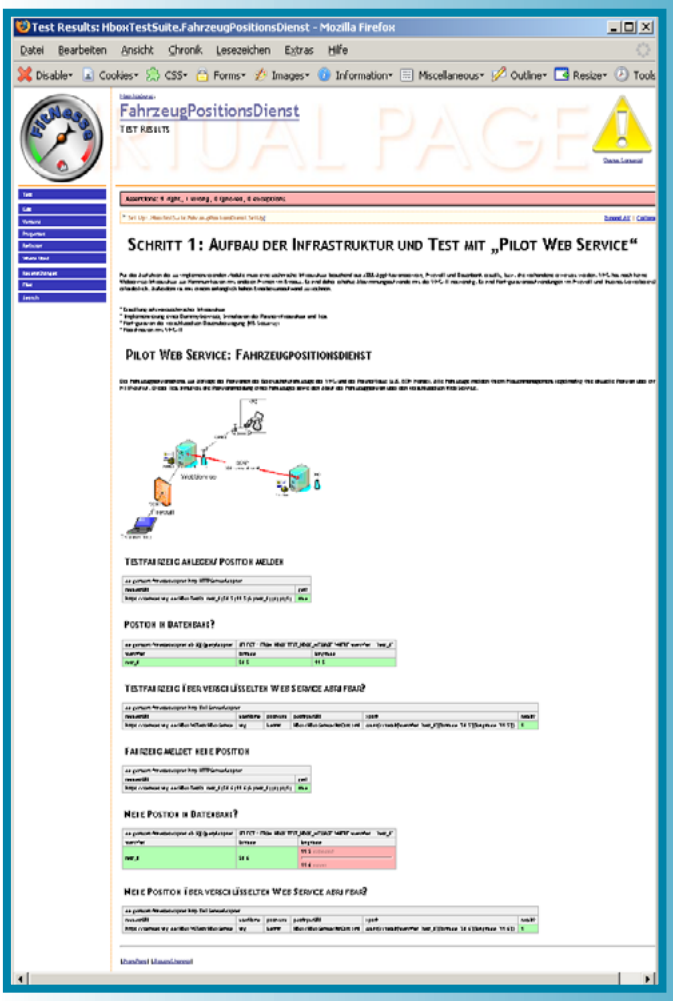

Abb. 5: FitNesseTestRun

Und kundentauglich?

Die Tests werden, wie bereits ausgeführt, in einem Wiki erfasst. Neben der Erfassung der Tests ist es selbstverständlich auch möglich, einfachen Text, Grafiken oder Verweise auf externe Ressource einzubinden. Dadurch können die Tests sehr einfach um eventuell notwendige Beschreibungen erweitert werden.

Auch die Präsentation der Testergebnisse ist so aussagekräftig wie nötig und so einfach wie möglich. Die Tabellenzellen mit den erwarteten Ergebnissen sind im positiven Falle einfach mit einer grünen Hintergrundfarbe versehen. Im negativen Falle ist der Zellenhintergrund rot und zusätzlich zum erwarteten Ergebnis ist das tatsächliche Ergebnis zum einfachen Vergleich eingefügt.

Diese Art der Präsentation ist bei den Kunden sehr positiv aufgenommen worden, da sie auch optisch gut gestaltet werden kann, ohne dabei an Funktionalität einzubüßen.

Das Adaptermuster in Fit/FitNesse erleichtert es auch, deutsche Übersetzungen für die eher technischen Selenese-Befehle zu gestalten. Zur Realisierung müsste einmalig ein Mapping zwischen deutscher Notation und tatsächlichem Selenese-Befehl im Adapter implementiert werden.

Wie so ein Testergebnis aussehen kann, zeigt Abbildung 5. Gegen Ende eines Softwareprojektes sollte natürlich die grüne Hintergrundfarbe dominieren.

Zusammengefasst und weiter geschaut

Durch die recht nahe liegende Kombination der Werkzeuge FitNesse, Selenium und CruiseControl und durch die Erweiterung bzw. Nutzung entsprechender Adapter wurde das Ziel der automatisierten Tests von integrierten Anwendungen vollständig erreicht. Zusätzlich wurde das Schreiben der Tests und notwendiger Testprotokolle erheblich vereinfacht.

Zukünftig werden noch weitere spezifische Adapter entstehen, beispielsweise der erwähnte Übersetzungsadapter oder ein Adapter zur Ansteuerung von Ereignissen in einem Workflowmanagementsystem. Es gibt auch Überlegungen, die Selenium IDE dahingehend zu erweitern, dass mit dieser direkt Testseiten für FitNesse erstellt werden können. Oder dass aus den für die Softwareentwicklung vorhandenen Modellen, beispielsweise für Geschäftsprozesse, perspektivisch auch Vorlagen für Tests im FitNesse generiert werden, für die ein Entwickler anschließend einen spezifischen Adapter schreibt.

Alles in allem hat die Erfahrung gezeigt, dass mit recht überschaubarem Aufwand die Vorteile von vorhandenen Testframeworks exzellent dazu genutzt werden können, die eigenen Testanforderungen nicht nur besser zu erfüllen, sondern auch die Aufwände für die eigene Testerstellung zu optimieren.

Literatur und Links

[AJAX] B. Daum, Rich Client Platforms und Rich Internet Applications, in: JavaSPEKTRUM, 6/2006, Seite 10 ff.
[CC] T. Michelmann, Kontinuierliches Feedback mit CruiseControl, in: JavaSPEKTRUM, 5/2006, Seite 71 ff.
[Echo2] Echo2-Webframework, `http://nextapp.com/platform/echo2/echo/`
[FIT] Fit: Framework for Integrated Test, `http://fit.c2.com/`
[FITN] FitNesse, `http://fitnesse.org/`
[JCO] St. M. Heldt, Th. Lieder, SAP JCO im Praxiseinsatz – Teil 3: ein RFC-Server auf Basis des SAP-Java-Konnektors, in: JavaSPEKTRUM, 6/2006, Seite 24 ff.
[JETT] Jetty-Webserver, `http://jetty.mortbay.org/`
[SELE] OpenQA: Selenium, `http://selenium.openqa.org`
[SELR] Selenium-Referenzseite, `http://openqa.org/selenium-core/reference.html`

Michael Kloss ist bei der itemis AG als Architekt und Coach tätig. Er hat sich auf die Architektur und Anwendungsentwicklung im Bereich Java EE und Tomcat spezialisiert und arbeitet intensiv im Bereich Test, Build-Management und Build Verification für Java EE-Applikationen.
E-Mail: michael.kloss@itemis.de.

Steffen Stundzig leitet die Leipziger Niederlassung der itemis AG. Er ist Softwarearchitekt und -entwickler mit Schwerpunkten in Java EE, Systemintegration und Identitymanagement. Er beschäftigt sich intensiv mit den Themen MDSD und Testautomation. Außerdem berät er Unternehmen bzgl. des gesamten Softwareentwicklungsprozesses.
E-Mail: steffen.stundzig@itemis.de.