Testen mit RIA

Funktionales Testen von Rich Internet Applications

Michael Hüttermann

Dieser Artikel gibt einen Überblick über das Testen als inhärenten Bestandteil eines Entwicklungsprozesses. Im Bereich funktionaler Tests von Rich Internet Applications (RIAs) hat sich der Einsatz der kostenlosen und frei verfügbaren Werkzeuge Selenium und WebTest durchgesetzt. Zunächst wird die Nutzung beider Werkzeuge dargestellt, dann werden sie verglichen. Dazu betten wir einen exemplarischen Selenium-Test in eine Java/TestNG-Infrastruktur ein, um die technische Testbarriere zu überwinden. Den gleichen funktionalen Test erstellen wir auch mit WebTest. Hier schreiben wir den Test einmal mit XML/Ant, dann mit Groovy. Alle Tests binden wir in einen automatisierten „Continuous Integration"-Prozess ein.

Funktionale Tests auf RIAs: Vorüberlegungen

Es ist anzustreben, sowohl funktionale als auch Komponententests komplementär einzusetzen, wobei wir uns im weiteren Verlauf ausschließlich auf die funktionalen Tests konzentrieren. Zur Durchführung funktionaler Tests auf Rich Internet Applications (RIAs) müssen auf der Anwendung (SUT, System Under Test) Benutzerinteraktionen simuliert werden. Das im Vorlauf spezifizierte, erwartete Verhalten wird dabei dem jeweiligen Verhalten der Anwendung zur Laufzeit gegenübergestellt. Der Soll-Zustand kann dabei über einen Mitschnitt auf einer Version der Web-Anwendung erhoben oder programmatisch bestimmt werden.

Neben einer generellen Verfügbarkeit der Anwendung und ihrer Features ist es auch wünschenswert, Zustände zu wohldefinierten Zeiten zu validieren. So kann es beispielsweise interessant sein, ob zu einem bestimmten Augenblick ein Button drückbar ist oder nicht. Diese Testfälle lassen sich ausgezeichnet in einen Smoke-Test einbeziehen.

Es ist eine Infrastruktur bereitzustellen, die die Tests automatisiert durchführt und die Ergebnisse der Testläufe offen verfügbar macht. Das Allerwichtigste ist aber, dass eine enge Kooperation zwischen allen Beteiligten, insbesondere den Entwicklern und Domänen-Experten etabliert wird.

Selenium, Selenium IDE und Selenium TestRunner

Selenium [Sel,KlSt08,KaiLie07] ist ein von der Unternehmung ThoughtWorks bereitgestelltes, in Java geschriebenes Werkzeug. Es steht derzeit als Version 1.0 Beta zur Verfügung. Selenium hängt sich via Proxy und JavaScript in die zu testende Seite. Im Wesentlichen besteht es aus zwei Komponenten:

- der Selenium IDE und
- dem Selenium RC (Remote Control).

Die Selenium IDE ist ein Firefox-Plug-In, das es nach initialer, einfacher Installation ermöglicht, Aktionen auf beliebigen Web-Anwendungen mitzuschneiden. Via Kontextmenü besteht weiterhin die Möglichkeit, aktuelle Geschehnisse oder die Existenz von Text auf der Oberfläche überprüfen zu lassen. Die Testsequenzen werden in Tabellenform (HTML) organisiert, können gespeichert und zu einem späteren Zeitpunkt erneut auf der Anwendung abgespielt werden. Dazu kann ebenfalls die Selenium IDE bedient, aber auch der Selenium TestRunner via Selenium IDE gestartet werden.

Im Gegensatz zur IDE verankert sich der Selenium TestRunner in die Webseite und ist kein eigenständiges Fenster: Er bettet die Zielanwendung in den Browser mit ein. Beide Werkzeuge sind grafisch und funktional überzeugend (s. Abb. 1).

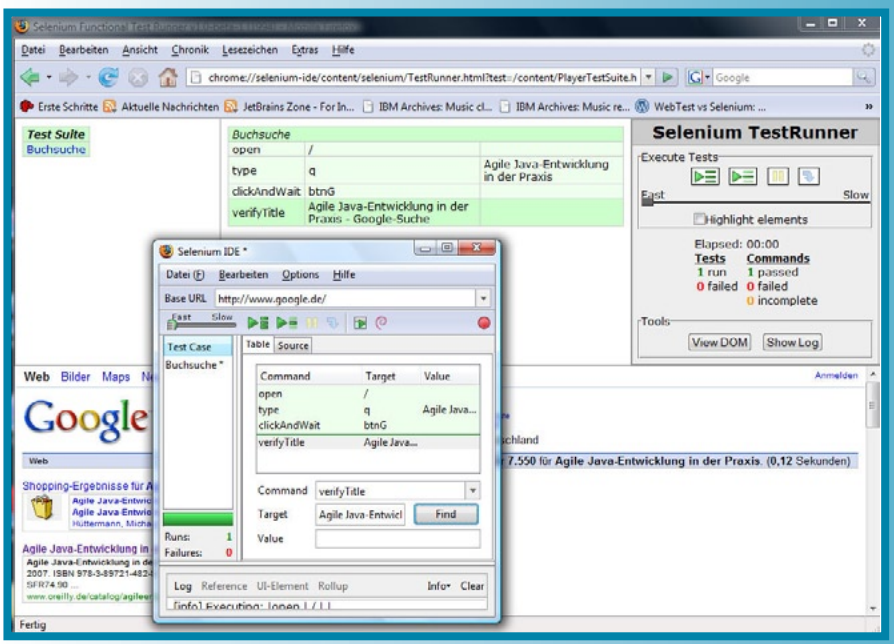

Abb. 1: Die Selenium IDE (im Vordergrund) und der Selenium TestRunner

Selenium RC

Es besteht die Möglichkeit, aus der Selenium IDE heraus die Testreihen in unterschiedliche Sprach-/API-Kontstrukte zu exportieren, beispielsweise nach JUnit [JUnit]. Die Exportfunktionalität ist eine schöne Hilfe, die zunächst nur als HTML-Tabellen vorgehaltenen Selenium-Konstrukte in Ihr Java-Ökosystem zu überführen und automatisch ausführbare Artefakte zu kippen.

Um nicht manuell über den Browser zu starten, sondern beispielsweise aus Ihren (Java)-Klassen heraus, benötigen Sie Selenium RC. Einmal die in der Dokumentation erläuterten Schritte zur Einbindung der nötigen Artefakte durchgeführt, steht das Selenium API in der Entwicklungsumgebung Ihrer Wahl (beispielsweise Eclipse) zur Verfügung.

Der Testlauf via TestNG

TestNG [TestNG] ist ein in Java geschriebenes Test-Framework, das aufgrund seiner Mächtigkeit für viele Test-Szenarien prädestiniert ist. Es hat insbesondere Stärken bei nebenläufigen Tests sowie bei der Definition von Abhängigkeiten und der Test-Aggregation. Selenium steuert einen realen Browser. Entsprechend referenziert das Testskript die Lokation der Firefox-Installation auf dem Rechner, auf dem der Test läuft. Auch andere Browser sind möglich und können angesprochen werden. Darüber hinaus ist es notwendig, einen Selenium RC-Konnektor auf- und wieder abzubauen. Im eigentlichen Test stoßen wir dann die separaten Benutzerschritte mit Hilfe des Selenium-APIs an. Die visuellen Controls können auf verschiedene Weise kontaktiert werden, beispielsweise über den Namen, das Label oder über einen XPath-Ausdruck (die Hierarchie und Position der Elemente zueinander berücksichtigend). Wir öffnen die Startseite, geben einen Wert in ein Eingabefeld und drücken einen Knopf. Hier führen wir eine Suche aus, die uns auf die Suchergebnisseite führt.

Listing 1 zeigt einen Test respektive einen Ausschnitt aus einer TestNG-Klasse, in dem mit Hilfe des Selenium API der aktuelle Titel einer Webseite einem Soll-Wert gegenübergestellt wird. Ferner haben wir in unsere UI-Interaktionen die Erstellung von Screenshots eingeflochten. Dies kann wunderbar zur Protokollierung der Testreihe oder auch zur Erstellung von Handbüchern dienen.

Selenium: Automatisierung und Reporting

Nun möchten wir aber nicht immer manuell die Klasse aus unserer Entwicklungsumgebung heraus und unter Zuhilfenahme des TestNG-Plug-Ins starten. Auch den Selenium-Server möchten wir automatisiert verwalten. Aus diesen Gründen kommt ein Ant-Skript [Ant] ins Spiel. Das Skript startet den Selenium-Server und führt anschließend in einem separaten Thread die Testreihe aus. Der Selenium-Server wird nicht nur automatisiert gestartet (via Java-Aufruf), er wird auch am Ende der Testreihe automatisiert heruntergefahren. Dazu nutzt das Skript aus, dass sich der Selenium-Server über die zwecks Testdurchführung lokal hochgefahrene Webseite herunterfahren lässt. Sowohl die Test NG-Testklasse als auch das Ant-Skript finden Sie in einem lauffähigen Eclipse-Beispielprojekt unter [HütBspSel].

Der Testzyklus liefert uns verschiedene TestNG-Reportseiten, eine ausführliche, die auf Threading, Testgruppen usw. eingeht, und eine für E-Mails geeignete, aggregierte Testübersicht (s. Abb. 2).

```
@Test
public void searchAndTestSERPTitle(){
  sel.open(url+"/");
  sel.type("q", SEARCH_STRING);
  sel.click("btnG");
  sel.captureScreenshot(SCREENSHOT_PATH+"shot"
                        +counter+++".png");
  sel.waitForPageToLoad("5000");
  sel.captureScreenshot(SCREENSHOT_PATH+"shot"
                        +counter+++".png");
  Assert.assertEquals(SEARCH_STRING + " - Google-Suche",
                      sel.getTitle());
}
```

Listing 1: Test aufbauend auf Selenium und TestNG

WebTest

WebTest [WT] ist ein von der Unternehmung Canoo angestoßenes und bereitgestelltes Test-Framework, das ebenfalls in Java geschrieben ist. Neben Zugriff auf Web-Anwendungen erlaubt WebTest das Testen/Arbeiten mit PDF-, Excel- und E-Mail-Dateien. Im Kontext von Tests auf Web-Anwendungen basiert WebTest im Gegensatz zu Selenium selbst auf einem Test-Framework namens HtmlUnit. HtmlUnit simuliert einen Browser. Heruntergeladen und den Path ergänzt, ist ein Quick-Start sehr schnell möglich. Es kann über WebTest ein Projekt-Testtemplate erstellt werden:

```
webtest -f path\to\webtest\home\webtest.xml wt.createProject
```

Wir betten unsere Tests also nicht in eine TestNG-Struktur ein, sondern nutzen eine für WebTest native Umgebung (die es in dieser Form für Selenium nicht gibt). Für WebTest existiert ebenfalls ein Entwicklungsumgebung (Firefox-Extension) namens WebTestRecorder [WTR]. Diese ist simpler gehalten als das Selenium-Pendant.

WebTest-Tests: Ant und/oder Groovy

WebTest-Skripte kommen in zwei Darreichungsformen daher. Traditionell haben sie die Form von Ant-Skripten. Inhaltlich führt das in Listing 2 dargestellte WebTest-Skript Schritte (in der WebTest-Notation: „steps“) und Prüfungen aus, die wir im Kontext von Selenium bereits kennengelernt haben. Das Skript lässt sich via „ant“ (recht praktisch aus Entwicklungsumgebungen heraus) oder auch via „webtest“ auf der Konsole starten. Es müssen dabei die zur WebTest-Distribution gehörenden

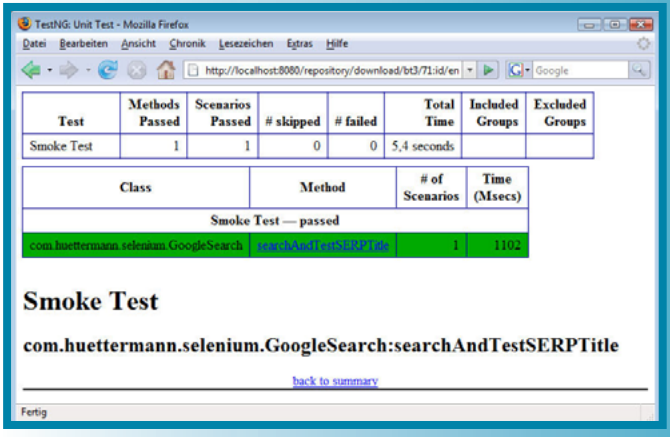

Abb. 2: SeleniumResult

```
<?xml version="1.0"?>

<!DOCTYPE project SYSTEM "../dtd/Project.dtd">

<project default="test">
  <target name="test">
    <webtest name="Buchsuche ...">
      <invoke url="http://www.google.de"
        description="Suche 'Agile Java-Entwicklung in der Praxis'"/>
      <setInputField name="q"
        value="Agile Java-Entwicklung in der Praxis" />
      <clickButton name="btnG" />
      <verifyTitle text=
        "Agile Java-Entwicklung in der Praxis - Google-Suche" />
    </webtest>
  </target>
</project>
```

Listing 2: WebTest-Test auf Basis von XML/Ant

Bibliotheken und vorgefertigten XML-Snippets mit wiederverwendbarer WebTest-Logik im Zugriff sein.

Alternativ steht die Möglichkeit zur Verfügung, Tests in Groovy-Notation [Groovy] in Ihre Testsequenz einzubinden. Zur Erfüllung der gleichen Testmotivation nutzt Listing 3 Groovy. Das generierte WebTest-Template hilft beim Aufsetzen der Projektstruktur und illustriert exemplarisch, wie Sie über einen zentralen Einstiegspunkt Testartefakte, die sowohl in XML/Ant als auch in Groovy vorliegen, einbinden und starten können.

WebTest: Automatisierung und Reporting

Da WebTest-Skripte Ant-Skripte sind und sich als solche ausführen lassen, steht einer traditionellen automatisierten Ausführung nichts im Weg. Testläufe werden via WebTest-Testreports protokolliert, in denen angenehm navigiert werden kann (s. Abb. 3). WebTest speichert für jedes Submit die zurückkommende Antwortseite. Diese werden ins WebTest-Verzeichnis geschoben und sind über die Report-Seite erreichbar. Unter [HütBspWebTest] finden Sie ein lauffähiges WebTest-Beispielprojekt, das die beiden beschriebenen Tests enthält.

Selenium und WebTest: Eine Gegenüberstellung

Sowohl mit Selenium als auch mit WebTest lassen sich Web-Anwendungen funktional testen. Beide Werkzeuge sind gut nutzbar und in eine Java-Projektinfrastruktur nahtlos integrierbar. Für beide Werkzeuge existiert kommerzieller Support (via Thoughtworks respektive Canoo) und beide sind sehr attraktive Alternativen zu „schwergewichtigen", kommerziellen Testwerkzeugen wie „Quick Test" oder „WinRunner" von HP, „SilkTest" von Borland oder „Rational Functional Tester" von IBM.

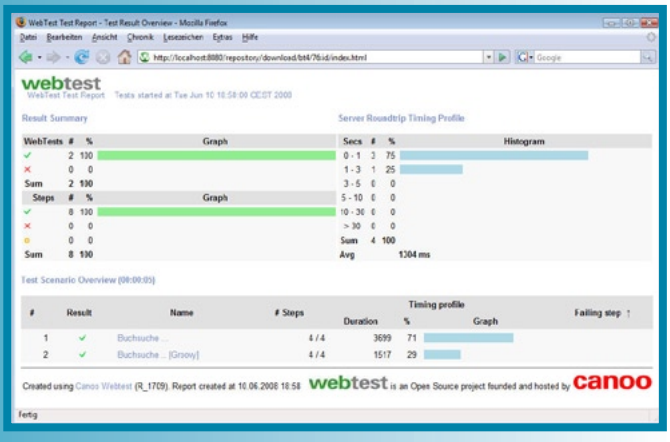

Abb. 3: WebTestResult

Selenium und WebTest erlauben eine flexible Adressierung der visuellen Komponenten beispielsweise über XPath-Ausdrücke. Für eine Einbindung der Selenium-Testsequenzen in Java-Artefakte auf der einen Seite, für das Schreiben von WebTest-Testsequenzen auf der anderen Seite ist technisches Wissen notwendig, sodass die Unterstützung eines Entwicklers bzw. einer Person mit technischer Affinität unumgänglich ist. Wenn wir Selenium in TestNG-Testklassen einbetten, stehen uns alle Möglichkeiten von TestNG zur Verfügung, beispielsweise auch ein Aufruf via Ant. Sowohl TestNG- als auch Ant-Artefakte lassen sich über jede gängige Entwicklungsumgebung anstoßen. Durch die Integration bzw. im Falle von WebTest die intensive Nähe zu Ant steht einem automatisierten Einsatz beider Werkzeuge nichts im Wege. Insbesondere die Kombination Selenium und TestNG erweist sich als hilfreich, eine gewöhnlich vorhandene technische Barriere zwischen funktionalen Tests und Komponententests (die häufig in JUnit oder TestNG-Form vorliegen) zu überbrücken (TestNG lässt sich übrigens sehr gut mit JUnit verheiraten). Gerade durch das Aufsetzen auf TestNG stehen uns reichhaltige Gruppierungs- und Abhängigkeitsdefinitionen zur Verfügung.

Auch nebenläufige Tests sind auf beiden Tools möglich. Wo wir uns im Falle von Selenium die Report-Möglichkeiten von dessen TestNG-Wirt zu nutze machten, erstellt WebTest via mitgelieferten Stylesheets aus XML-Artefakten inhaltlich und optisch überzeugende Reports selbst. Auch Histogramme sind verfügbar, wie viel Prozent Ihrer Server-Anfragen in wie viel Sekunden zurückkommen. WebTest-Skripte können Sie mit XML-Bordmitteln beliebig gruppieren, Ant-Makros nutzen usw. Hier finden Sie native Unterstützung vor, da es auf den De-facto-Standard Ant aufsetzt.

Anlaufkosten, Dokumentation und Browser-Awareness, -Stickiness

Selenium hat verhältnismäßig hohe Anlaufkosten. Die Installation mit der im Anschluss vorzunehmenden, für die individuelle Umgebung optimalen Konfiguration kann sich als trickreich herausstellen, beispielsweise wenn Proxys ins Spiel kommen. Die Dokumentation ist dabei nicht immer erschöpfend und verteilt sich über mehrere isolierte Einstiegspunkte (Selenium IDE, Selenium RC). Bei WebTest kommt man recht schnell zu einem Ergebnis. Es bietet ferner eine umfassende, informative Dokumentation (Nachschlagedokumentation, Referenz, Beispiele, Code-Templates).

Da Selenium auf realen Browsern aufsetzt, können Web-Anwendungen eben auch auf diesen unterschiedlichen Browsern getestet werden. Dies ist vor dem Hintergrund hilfreich, dass JavaScript (und das DOM) für verschiedene Browser unterschiedlich aussieht („cross-site scripting"). Diese „Browser-Awareness und -Stickiness" ist auch der Grund, warum Selenium mit defektem Html, also non-valide Html-Seiten, besser umgehen kann. Das WebTest-Team plant in diesem Kontext durch Integration des WebDrivers [WD], hier als eine Art Middleware zu verstehen, sowohl weiterhin das Testen mit der Browser-Emulation und HtmlUnit zu ermöglichen als auch zukünftig das Ansprechen realer Browser zu erlauben.

Die enge Integration mit konkreten Browsern kann allerdings auch zu Unbequemlichkeiten führen. Selenium startet beispielsweise standardmäßig das Default-Profil des Browsers. Dies kann dazu führen, dass Integrationen in Ihren Browser (wie ein Virenscanner) bei jedem Selenium-Testlauf neu in-

```
import com.canoo.webtest.WebtestCase
import org.junit.Test

/**
 * @author Michael Huettermann
 */
public class GoogleWebtestTest extends WebtestCase {
  void testSearch() {
    webtest("Buchsuche .. [Groovy]") {
      invoke url: "http://www.google.de", description: "Buchsuche .."
      setInputField name: "q",
      value: "Agile Java-Entwicklung in der Praxis"
      clickButton name: "btnG"
      verifyTitle
        text:"Agile Java-Entwicklung in der Praxis - Google-Suche"
    }
  }
}
```

Listing 3: WebTest-Test auf Basis von Groovy

stalliert werden sollen (Selenium bietet in diesem Kontext die Möglichkeit, ein Browserprofil beim Start mitzugeben).

Web 2.0 und Ajax

Insbesondere bei Ajax-Anwendungen sind intelligente Test-Sequenzen aufzusetzen: So können wir im Falle von Selenium Warteschleifen einführen, die auf die Existenz von Controls warten, wie beispielsweise in Listing 4 dargestellt. WebTest bietet hier neben dem „einfachen" Einfügen von Warteschleifen die Konfiguration via `<config easyajax="true"/>` an. Dies verursacht einen Aufruf nach jedem WebTest-Step und wartet eine definierte Zeit auf die aktuelle Antwort. Die Wartezeit kann mit dem zusätzlichen XML-Attribut `easyajaxdelay` definiert werden.

Möchten Sie eine Anwendung testen, die über fortgeschrittenes JavaScript verfügt (Ajax), ist Selenium derzeit die bessere Wahl. Obwohl stark gegenüber der ursprünglich genutzten Browser-Engine HttpUnit verbessert, hat HtmlUnit als WebTest-Wirt keine umfassende Unterstützung von Tests auf sehr JavaScript-lastigen Oberflächen.

Von Capture Replay und Model-Driven Testing

Für beide Werkzeuge existiert ein Recorder. Nachhaltiges funktionales Testen ist mehr als das „einfache" Aufnehmen von Benutzerinteraktionen mit einem Capture-Replay(CR)-Werkzeug und deren Abspielen, wobei dieses Vorgehen ein guter Start sein kann. Testen ist viel eher eine Melange aus unterschiedlichen Aktivitäten, beispielsweise die Erstellung von wiederverwendbarer, aussagekräftiger Testlogik oder die Förderung der Unanfälligkeit der Tests gegen Änderungen auf der Zielanwendung.

WebTest beispielsweise adressiert mit seinem Recorder standardmäßig die Komponenten über das Label. Das kann zu Schwierigkeiten führen, wenn es mehr als eine Komponente mit demselben Label gibt (innerhalb von Tabellen beispielsweise). Hier ist es hilfreich, die Komponente nachträglich über eine eindeutige ID anzusprechen. Auch Tests wie die Überprüfung, ob zu einem bestimmten Zeitpunkt keine NullPointerException auftrat, kann via CR nicht ohne weiteres implementiert werden.

```
for (int second = 0;; second++) {
  if (second >= 60) fail("timeout");
  try {
    if (selenium.isTextPresent("Hier bin ich.")) break;
  } catch (Exception e) {}
    Thread.sleep(1000);
}
```

Listing 4: Manueller Programmiereingriff in einen Test einer Ajax-Anwendung

Model-Driven Testing geht von einem mentalen Modell aus und überprüft Erwartungen gegen die Anwendung (aus Use Cases oder User Stories). Es ist tolerant gegenüber fehlerhaftem Testverhalten. Falls visuelle Komponenten via XPath adressiert werden, so ist beispielsweise anzustreben, ein Element relativ anzusprechen und nicht absolut vom Wurzelelement ausgehend. Auf diesem Wege führt eine Umgruppierung auf der Oberfläche nicht zu einem fehlerhaften Test, obwohl die Funktionalität der Anwendung korrekt ist („fault negative").

Von Data-Driven Testing und Script Automation

Wichtig ist es, Testreihen mit unterschiedlichen Daten laufen lassen zu können, also Datenvarianten einem gleichen Workflow mitzugeben. WebTest zieht hier Vorteile aus seiner Ant-Nähe. So kann der Ant-Task `dataDriven` zum Einsatz kommen. Dieser ist unabhängig von WebTest und kann eine Excel-Datei referenzieren. Die dortigen Tabellenspalten können im Rahmen des WebTests als Parameter referenziert werden.

Nutzen wir Selenium via TestNG, besteht die Möglichkeit, einen TestNG DataProvider einzusetzen. Von Script Automation spricht man, wenn die flexible Möglichkeit besteht, via Skripten die Tests zu steuern/schreiben. Werden Tests via WebTest geschrieben, besteht durch den Groovy Ant-Builder umfassender Komfort. In der Selenium-Welt kann diese Flexibilität mit TestNG erreicht werden.

Ziehen Sie es vor, nicht mit Java, sondern mit einer dynamischen Sprache Ihre Tests zu steuern und Selenium einzubinden, so können Sie auch hier Groovy einsetzen, wobei im vorliegenden Beispiel für beide Szenarien prototypische Lösungen vorgestellt wurden.

Von Continuous Integration und SPOT

Continuous Integration [CI], das frühzeitige Synchronisieren der Quellen, deren Integration auf einer der Zielkonfiguration gleichen Umgebung und das Testen zum frühen Finden der Fehler, hilft die Qualität zu sichern und frühzeitig Rückkopplung an Kunde, Projektleitung und Entwicklung zu liefern. Neben den Smoke-Tests kommen hier insbesondere auch die gewöhnlich langlaufenden Regressions- und Integrationstests ins Spiel.

Auf einer dedizierten Maschine wird der „komplette Build" angestoßen. Dieser umfasst beispielsweise das Auschecken der Quellen aus dem Versionskontrollsystem (VCS), die Kompilierung, die Ausführung von Testreihen, die Überprüfung der Code-Qualität (beispielsweise mit Checkstyle oder FindBugs), die Paketierung und das Deployment auf ein Testsystem. Der Build-Server fungiert dabei als „Single Point of Truth" (SPOT), da private Builds in den Workspaces einzelner Entwickler oder dort manuell angestoßene Aktivitäten wie die Ausführung von Checkstyle schon mal vergessen werden oder gar über andere Test-Konfigurationen verfügen, als die für das Projekt zentral vorgegebene.

Die zentralen Builds werden gewöhnlich zeitlich gesteuert, indem beispielsweise ein solcher Build jede Nacht einmal („Nightly Build") oder jede Stunde bei vorliegenden Änderungen am Code-Bestand im VCS angestoßen wird. Auch das Anstoßen eines Build bei jedem neuen Commit ist durchaus verbreitet. Hier kann eine gestaffelte Build-Architektur aufgesetzt werden, die nach jedem Commit einen Smoke-Test durchführt und erst in der Nacht einen umfassenden Testlauf startet (Build Staging).

Continuous-Integration-Server wie TeamCity [TC] ermöglichen dabei, die Builds auf einem Build-Grid verteilt zu erstellen,

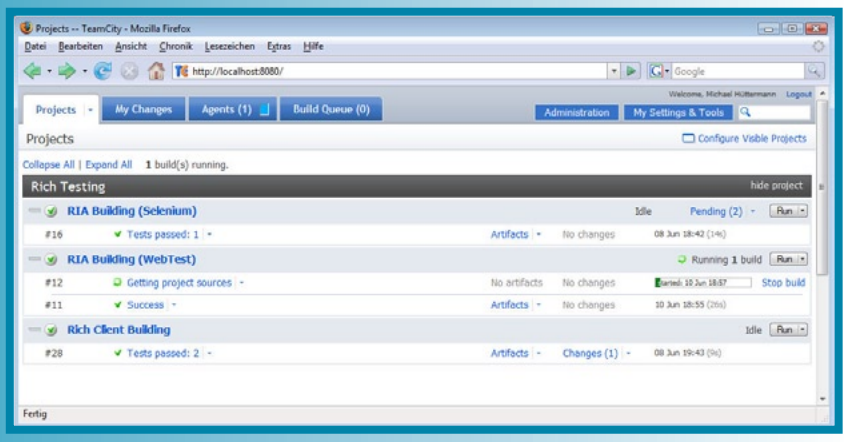

Abb. 4: Continuous Integration

geben über den Lauf interaktives Feedback und stellen die erstellten Artefakte und Reports zur Verfügung, auch von älteren Builds. TeamCity erlaubt die Aufnahme unterschiedlichster Build-Steuerungsartefakte inklusive Ant. Auch Maven-getriebene Projekte können eingebunden werden (sowohl Selenium als auch WebTest lassen sich mit Maven verheiraten). TeamCity verfügt weiterhin über eine Integration für die Entwicklungsumgebungen Eclipse und IntelliJ IDEA und ermöglicht das Anstoßen von privaten, auf einem entfernten Build-Agent durchgeführten Builds aus der Entwicklungsumgebung heraus.

Da sich sowohl Selenium als auch WebTest direkt bzw. indirekt in Ant einbetten lassen respektive sich fundamental an Ant orientieren, ist eine Aufnahme in einen Continous-Integration-Prozess reibungslos möglich. Selenium setzt zwar einen Browser voraus, Sie können es aber auch „headless" betreiben, beispielsweise mit Hilfe von Xvfb [Xvfb]. Sie sollten die funktionalen Tests als Smoke-, Integrations- oder Systemtests entsprechend sinnvoll in die chronologische Sequenz Ihres Build-Prozesses einbinden (beispielsweise nach dem Kompilieren und den Komponententests bzw. dem Deployment auf ein Testsystem).

Abbildung 4 zeigt exemplarisch, wie die zwei Testprojekte für WebTest und Selenium mit Hilfe des in dieser Ausbaustufe kostenlosen TeamCity organisiert werden und das WebTest-Projekt gerade neu „gebaut" wird.

Zusammenfassung

Aufbau und Pflege einer umfassenden funktionalen Testabdeckung ist nicht kostenneutral, zahlt sich aber sehr schnell aus. Auch wenn Sie keine umfassende Testabdeckung anstreben, so kann zumindest ein Smoke-Test überprüfen, ob die rudimentären Funktionen der Anwendung nutzbar bzw. erreichbar sind und Ihre Erwartungen gegen die Anwendung erfüllt werden. Smoke-Tests sind bestens dafür geeignet, vor jedem Einspielen von Änderungen ins zentrale Code-Repository die Qualität der Anwendung im lokalen Workspace zu validieren. Testen ist Software Engineering.

Eine herkömmliche Aufnahme von Benutzerinteraktionen mit einem Rekorder ist ein guter erster Schritt, aber nicht der letzte. Es muss immer klar sein, was Sie von der zu testenden Seite erwarten, und dies müssen Sie in Spezifikationen ausdrücken. Es gilt zu unterscheiden zwischen garantiertem und versehentlichem Verhalten. Der Software-Engineering-Teil beginnt, wenn Sie Duplikate im Test-Skript beheben, Module zwecks Wiederverwendbarkeit extrahieren, Adressierungen schleifen. Die Anwendung wird es Ihnen danken, sie wird ein besseres Design erhalten und die laufenden Entwicklungskosten werden reduziert.

Sowohl Selenium als auch WebTest erlauben das Schreiben funktionaler Tests in Form von ausführbaren Spezifikationen und können Bestandteil sein einer umfassenden, agilen technischen Infrastruktur [ASE]. Continuous-Integration-Werkzeuge wie TeamCity helfen, von der Software in regelmäßigen Abständen ein vollständiges „Blutbild" zu erstellen.

Welche Werkzeuge Sie letztendlich einsetzen, hat neben den Anforderungen schließlich auch mit den Rahmenbedingungen und persönlichen Präferenzen zu tun. Noch wichtiger als das Tooling und der umgebende Prozess ist allerdings immer eine gesunde Kommunikation und Teamwork in Form von konstruktiven, kooperativen Interaktionen und ein kollektives Verantwortungsbewusstsein.

Links

[Ant] The Apache Ant Project, `http://ant.apache.org/`
[ASE] M. Hüttermann, Agile Java-Entwicklung in der Praxis, O'Reilly, 2007, `http://www.oreilly.de/catalog/agileentwger/`
[CI] M. Fowler, Continuous Integration, 2006, `http://martinfowler.com/articles/continuousIntegration.html`
[Groovy] Groovy, An agile dynamic language for the Java Platform, Codehaus Foundation, `http://groovy.codehaus.org/`
[HtmlU] HtmlUnit, `http://htmlunit.sourceforge.net/`
[HütBspSel] Lauffähiges Eclipse-Beispielprojekt für Selenium, `http://87.230.78.21:4711/svn/projects/trunk/RIATesting/`
[HütBspWebTest] Lauffähiges Eclipse-Beispielprojekt für WebTest, `http://87.230.78.21:4711/svn/projects/trunk/RIATesting2/`
[JUnit] JUnit, `http://www.junit.org/`
[KaiLie07] M. Kain, Th. Lieder, Testen vereinfachen mit Selenium, in: JavaSPEKTRUM, 04/2007
[KlSt08] M. Kloss, St. Stundzig, Automatisierte Integrationstests, in: JavaSPEKTRUM, 03/2008
[Sel] Selenium, OpenQA, `http://selenium.openqa.org/`
[TC] TeamCity: Powerful Solution for Continuous Integration & Build Management, `http://www.jetbrains.com/teamcity/`
[TestNG] Test-Framework der „nächsten Generation" TestNG, `http://testng.org/`
[WD] WebDriver, A developer focused tool for automated testing of webapps, Google, `http://webdriver.googlecode.com/`
[WT] Canoo WebTest, `http://webtest.canoo.com`
[WTR] WebTestRecorder, `http://webtestrecorder.canoo.com/`
[Xvfb] X virtual framebuffer, Wikipedia, the free encyclopedia, `http://en.wikipedia.org/wiki/Xvfb`

Sun Java Champion **Michael Hüttermann** ist freiberuflicher Experte für Java/JEE und agile Softwareentwicklung. Er ist im Board der JetBrains Development Academy, Organisator der Java User Group Köln, java.net JUGs Community Leader, Sprecher auf internationalen Konferenzen sowie Autor zahlreicher Artikel und des Buchs „Agile Java-Entwicklung in der Praxis". E-Mail: michael@huettermann.net.