Testen von Web Application Frameworks

Peter Kröpfl

3. Februar 2007

Inhaltsverzeichnis

1 Einleitung **4**

2 Testen **5**

3 Ruby on Rails **6**

3.1 Testing in Ruby on Rails 6
3.1.1 Assertion 6
3.1.2 Test/Testmethode 7
3.1.3 TestCase 7
3.1.4 TestSuite 8
3.2 Tests Ausführen 8

4 Cocoon **10**

4.1 Testen in Cocoon 10

5 Frontend Tests **12**

5.1 HTTPUnit 12
5.1.1 Überblick 12
5.1.2 Verwendung 12
5.1.3 Fazit 14
5.2 HTMLUnit 14
5.2.1 Überblick 14
5.2.2 Verwendung 14
5.2.3 Fazit 16
5.3 JMeter 16
5.3.1 Überblick 16
5.3.2 JMeter für Frontend Tests 17
5.4 JMeter am Beispiel JOP 22
5.4.1 Überblick 22
5.4.2 Test 1: Aufruf der Startseite 23
5.4.3 Test 2: Login/Logout 24
5.4.4 Test 3: Registrierung 28
5.5 JMeter Tests ausführen 34
5.5.1 Kommandozeilen Parameter übergeben 36
5.5.2 Tests Aufzeichnen 36

6 Verwandte Literatur **38**

6.1 Analysis and Testing of Web Applications 38
6.2 Improving Web Application Testing with User Session Data . 38
6.3 Automated Replay and Failure Detection for Web Applications 39

6.4 Testing Web Applications focusing on their Specialties 40

7 Fazit **41**

1 Einleitung

In dieser Arbeit soll beleuchtet werden wie Testen von Webapplikationen möglich ist. Dazu wird zunächst Untersucht wie weit Testen in aktuellen Web Application Frameworks bereits integriert ist. Dazu werden zwei verbreitete und gänzlich unterschiedliche Web Applikation Frameworks untersucht. Zum einen Apache Cocoon und zum anderen Ruby on Rails.

Als nächster Schritt wird untersucht welche (opensource) Tools bereits existieren um Web Applikationen testen zu können. Diese werden kurz evaluiert, um dann eines dieser Tools anhand einer selbst entwickelten Web Applikation auf seine praxistauglichkeit Untersuchen zu können.

2 Testen

An dieser Stelle sollen zunächst kurz die wichtigsten Begriffe in Zusammenhang mit Software Testen für diese Arbeit definiert werden:

- Testen: Unter dem Begriff Testen wird in dieser Arbeit das systematische Prüfen und Verifizieren einer gewissen Funktionalität der Software verstanden.
- Frontend Test: beschreibt Testen aus Benutzersicht. Dabei werden Tests so durchgeführt, wie ein realer Benutzer es könnte. Oft auch Blackbox - Test genannt.
- Unit Test: ein Test für eine bestimmte Codeeinheit einer Applikation. Zum Beispiel könnte ein Unit Test überprüfen, ob der Login - Vorgang korrekt funktioniert.
- Assertion: Englisch für Aussage, Behauptung. Assertions werden in Unit Tests verwendet um gewünschtes Verhalten zu definieren und dadurch eine automatische Prüfung zu ermöglichen. So könnte Beispielsweise eine Assertion im Login Unit Test falls das login fehlschlägt muss eine Fehlermeldung angezeigt werdenlauten.
- Mock Obejkt: Ein Mockobjekt wird von einem Entwickler dann erstellt, wenn ein gewisses Objekt benötigt wird, dieses aber beispielsweise nur sehr langsam oder aufwendig erstellt werden kann. An Stelle des originalen Objekts wird dann das Mock Objekt eingesetzt.

3 Ruby on Rails

Ruby on Rails ist ein sehr populäres Web Application Framework, das auf der Programmiersprache Ruby basiert. Es wurde ursprünglich von David Heinemeier Hansson entwickelt und Juli 2004 erstmals veröffentlicht.

Ruby on Rails beeinhaltet bereits einige sehr praktische Features. So wird Ruby on Rails bereits mit einem Persistency Framework (ActiveRecord) ausgeliefert, das die Datenbank - Interaktion bereitstellt. Neben diesem und anderen Features wird Ruby on Rails gleich mit einem integriereten Testing Framework ausgeliefert. Dieses soll nun betrachtet werden.

3.1 Testing in Ruby on Rails

Unit Testing ist ein fester Bestandteil von Ruby on Rails. Erstellt ein Entwickler beispielsweise ein Model, oder einen Controller mit der `script/generate model` oder der `script/generate controller` Methode, so werden automatisch passende Unit Test - Grundgerüste erzeugt.

Ruby on Rails Unit Testing besteht aus vier Grundkomponenten:

3.1.1 Assertion

Eine Assertion ist eine Behauptung, die als Ergebnis nur *true* oder *false* haben kann. Eine Beispiel wäre die Assertion "Die Kundennummer muss 8 Stellen habenÏn einem Test wird kann dann diese Behauptung überprüft werden.

Grundsätzlich folgt eine Assertion immer einem gewissen Schema: Ihr Name beginnt mit der Zeichenfolge *assert*. Als Parameter übergeben wird mindestens ein Objekt und zusätzlich kann optional immer noch eine Nachricht übergeben werden. Diese Nachricht sollte einen Text enhalten, der die Assertion erklärt und bei einem *Fail* angezeigt wird. Ruby verfügt bereits über eine sehr große Anzahl an vordefinierten Assertions:

Ein paar Beispiele:

- `assert (boolean, [Message])` Ist die übergebene bool'sche Variable `boolean` true?
- `assert_same (object1, object2, [Message])` Sind die beiden Objekte `object1` und `object2` gleich?
- `assert_match (regex, string, [Message])` Passt die Zeichenfolge `string` zu der Regular Expression `regex`?

- `assert_raises (exception1, excetion2,..) block` Wirft der Codeblock `block` eine der Exceptions `exception1, exception2,..` ?

Natürlich kann jeder Benutzer die Liste der vordefinierten Assertions nach Belieben erweitern indem er einfach selbst Assertions schreibt.

3.1.2 Test/Testmethode

Ein Test oder eine Testmethode stellt einen Test für eine spezielle Funktionalität dar. Der Name eines Tests muss mit der Zeichenfolge *test* beginnen. Zum Beispiel: `test_login` oder `test_calcuation`

Nennt man seine Testmethoden nicht beginnend mit *test*, so können sie trotzdem ausgeführt werden, da sie ja trotzdem normale Ruby Methoden sind. Es besteht jedoch die Einschränkung, dass sie nicht automatisiert ausgeführt werden können, da sie nicht als Test erkannt werden, wenn sie nicht mit dem Namen *test* beginnen.

Ein Test muss außerdem mindestens eine Assertion enthalten damit er ein Ergebnis zurückliefern kann. Es ist natürlich auch möglich mehrere Assertions pro Testmethode aufzurufen. Dabei muss man sich aber immer im Klaren darüber sein, dass nicht alle Assertions ausgeführt werden könnten. So ein Fall tritt immer dann auf wenn in einer Methode eine Assertion fehlschlägt. Dann wird die gesamte Testmethode abgebrochen und der Test als *failed* gekennzeichnet. Alle anderen Assertions in dieser Methode werden nicht mehr ausgeführt.

3.1.3 TestCase

Ein Testcase ist eine Klasse, die aus mehreren Testmethoden besteht. Sie muss folgende Attribute aufweisen um eine gültige Testklasse zu sein:

- Sie muss `require 'test/unit'` enthalten
- Sie muss von Test:Unit:TestCase erben: Bsp: `class MeineTestKlasse < Test::Unit::TestCase`
- Ihre Methoden müssen mit der Zeichenfolge `test` beginnen (außer setup und teardown)
- Die Methoden müssen Assertions enthalten.

Diese Regeln treffen auf alle Testmethoden zu, bis auf zwei Ausnahmen. Sie heißen `setup` und `teardown`. Im Unterschied zu allen anderen Testmethoden müssen diese nicht mit der Zeichenfolge *test* beginnen. Diese beiden Methoden haben spezielle Aufgaben im Zusammenhang mit der Ausführung von Tests:

- `setup`: Die Setup Methode wird vor jedem einzelnen Test aufgerufen. In ihr können allgemeine Einstellungen vorgenommern werden, wie zum Beispiel Initialisierung von Variablen oder erstellen von Objekten, die alle Testmethoden benötigen.
- `teardown`: Die Teardown Methode ist das Gegenteil der Setup Methode. Sie wird immer am Ende eines Tests aufgerufen. In ihr könnten zum Beispiel die Variablen und Objekte wieder zerstört werden.

3.1.4 TestSuite

Bis hier her konnten wir lediglich einzelne Testmethoden aufrufen. Durch den Einsatz einer Testsuite kann beispielsweise ein Test für eine ganze Applikation automatisiert ablaufen. Eine Testsuite besteht aus mehreren Testklassen. Wenn nun die TestSuite ausgeführt wird, so wird nacheinander jede einzelne Testklasse und damit auch jede einzelne Testmethode ausgeführt. So kann man mit einem Aufruf eine gesamte Applikation testen.

3.2 Tests Ausführen

Mit Hilfe dieser Grundkomponenten kann man die gesamte Funktionalität einer Rails Applikation abdecken und so automatisiert testen lassen. Selbstverständlich können die einzelnen Tests alleine aufgerufen werden. Der Aufruf eines Tests geschieht genauso, wie der Aufruf von allen anderen Ruby Klassen:

```
ruby test_meine_klasse.rb
```

Jeden Test einzeln aufzurufen wäre aber höchst unpraktisch und oft sogar garnicht möglich. Aus diesem Grund gibt es ja auch TestSuiten. Ruby on Rails bietet aber noch eine weitere Erleichterung. Wie bereits erwähnt erstellt Rails automatisch Test Grundgerüste (Stubs) wenn Models oder Controller mittels des Scripts erstellt werden:

```
script/generate controller
script/generate model
```

Diese Aufrufe erstellen die Stubs für die Models und Controller und gleichzeitig erstellen oder erweitern sie eine Verzeichnisstruktur unter *test*, die sämtliche, für Unit Testing notwendige, Informationen enthält.

```
test
  |--- fixtures
  |--- functional
  |--- mocks
  |--- unit
```

Diese Verzeichnisse enthalten alle benötigten Stubs, die für einen automatischen Test der Models oder der Controllers erforderlich sind.

- fixtures: Fixtures enthält Testdaten, falls dies notwendig ist. Diese können entweder im CSV oder YAML Format sein.
- functional: Controller Tests
- mocks: Etwaige Mock Objekte
- unit: Model Tests

Wenn so eine Verzeichnisstruktur besteht und die Test - Stubs zu sinnvollen Tests erweitert wurden, so kann Ruby on Rails mittels des Tools *rake* automatisch testen. Für einen Testlauf, der alle Models testet muss der Entwickler

```
rake test_units
```

aufrufen. Will ein Benutzer alle Controller testen, so muss er

```
rake test_functional
```

aufrufen.

4 Cocoon

Cocoon wurde 1998 von Stefano Mazzocchi initiiert. Bereits im November 1999 begann die Arbeit an Cocoon 2 und zwei Jahre später war Cocoon von einem kleinen Servlet, das XML - Files mit Hilfe von XSL transformieren konnte zu einem großen, sehr feature-reichen Web Development Framework gewachsen.

4.1 Testen in Cocoon

Im Unterschied zu Ruby on Rails hat Cocoon kein eigenes Testing Framework integriert. Da Cocoon in Java geschrieben ist und für Java bereits mit JUnit (http://junit.org) ein sehr mächtiges Test Framework existiert, war es offenbar noch nicht notwendig ein Testframework in cocoon zu integrieren.

Mit Hilfe von JUnit kann man gewisse Teile von Cocoon sehr gut Testen, leider ist es aber nicht möglich Frontend Tests durchzuführen. Dafür gibt es bereits andere Tools, wie zum Beispiel HTTPUnit, oder HTMLUnit. Dies sind Unit Test Tools, die das Verhalten einen Benutzers simulieren können. Zusätzlich sei noch JMeter erwähnt. Dies ist eigentlich ein Testwerkzeug um Lasttests durchzuführen, doch es kann auch sehr gut für funktionale Tests verwendet werden.

Da sich cocoon streng an das MVC Pattern hält, können die Testtools folgendermaßen kategorisiert werden:

- Model: Zum Testen des Models des Model View Controller - Patterns bietet sich JUnit an, da in Cocoon das Model zumeist aus Java Klassen besteht.
- View: Hier können nur Frontend Testtools wie HTML Unit, HTTP Unit oder JMeter zum Einsatz kommen
- Controller: Derzeit ist es leider nicht möglich den Controller getrennt testen zu können. Es gibt zwar ein Projekt names *counit*, doch das ist noch nicht weit genug entwickelt um in Cocoon die Controller vollständig zu testen.

counit

Counit ist ein Projekt, das Unit Tests, speziell für cocoon ermöglichen soll. Es wurde von Nico Verwer entwickelt und basiert auf einem simplen Prinzip: Es ruft Cocoon Pipelines auf und wertet das Ergebnisse dieser Pipelines aus. Da bekanntlich der Input und er Output von cocoon Pipelines zumeist XML ist, kann dieser schon vorher definiert und so leicht verifiziert werden.

Leider kann counit nicht mit Flow umgehen und ist daher für Tests. die im Rahmen dieser Arbeit durchgeführt werden nicht geeignet, weil das Beispielprojekt sehr stark auf Flow basiert.

5 Frontend Tests

5.1 HTTPUnit

5.1.1 Überblick

HTTPUnit[1] ist ein in Java geschriebenes open source Tool, das dem Benutzer die Möglichkeit bietet Websites automatisiert zu testen. Der Author von HTTPUnit ist Russell Gold.

Die Funktionsweise ist sehr stark mit JUnit[2] verwoben. So ist es nötig erst ein JUnit Test Framework zu erstellen um danach die Test - Methoden schreiben zu können.

Aus diesen Gründen bendeutet die Verwendung von HTTPUnit für Benutzer, die mit JUnit vertraut sind keine große Umstellung, da die grundlegenden Konzepte, wie Tests beschrieben und erstellt werden gleich sind. Die

Grundidee von HTTPUnit ist, das Verhalten eines Benutzers einer Website zu imitieren. Aus diesem Grund unterstützt HTTPUnit neben der Verarbeitung von HTML und dem Senden und Empfangen von HTTP-Reqests auch weitere Funktionalitäten, wie zum Beispiel:

- Grundlegende JavaScript Verarbeitung
- Basic Authentication
- Handling von https und Zertifikaten
- Verschiedene Zeichensätze
- Proxy Handling
- Weiterleitungen (Referrers)

5.1.2 Verwendung

Um HTTPUnit - Tests zu erstellen geht man genauso vor wie bei JUnit.

1. Eine neue Klasse erstellen, die `TestCase` erweitert
2. `com.metaware.httpunit.*` und `junit.framework.*` importieren
3. TestSuite erstellen

[1] HTTPUnit Homepage: http://www.httpunit.org/
[2] JUnit Homepage: http://www.junit.org

4. einzelne Test - Methoden schreiben, die mit der Zeichenfolge `test` beginnen müssen.

Nachdem man wie bei JUnit seine Testumgebung aufgebaut hat, kann man in den einzelnen Test - Methoden seine Tests schreiben.

Das Kernstück eines HTTPUnit - Tests ist die Klasse `WebConversation`, sowie die Klassen `WebRequest` und `WebResponse`. Die Klasse WebConversation ist der Grundstein jedes HTTPUnit - Tests. Sie ist verantwortlich für das Session- und HTTP - Response - Handling. Die Klasse WebRequest ermöglicht die Erstellung von HTTP - Requests.

Beispiel

```
WebConversation wc = new WebConversation();
WebRequest req = new GetMethodWebRequest( "http://test.com/index.html" );
WebResponse resp = wc.getResponse( req );
String siteAsString = resp.getText();
```

Hier sieht man die darüber beschriebenen Klassen in einem konkreten Beispiel:

In der ersten Zeile wird eine WebConversation gestartet. Danach wird in Zeile zwei eine Request an die Adresse der zu testenden Website erstellt. In der dritten Zeile wird ein Response Objekt erstellt. Diesem wird Response des vorher erstellten Requests zugewiesen.

In der letzten Zeile wird die Antwort in einen String umgewandelt und kann weiter verarbeitet oder durch die typischen JUnit Assertions überprüft werden.

Die Klasse Response bietet neben der Methode `getText()` noch viele andere sehr praktische Funktionen. Ein paar davon sind in der nachfolgenden Liste angeführt.

- `getDOM()` Liefert das Ergebnis des Request, also die HTML Site als DOM - Tree zurück
- `getForms()`, `getFormWithName(String name)`, `getFormWithID(String id)` Diese Funktionen bieten direkten Zugriff auf HTML Forms der Response
- `getLinks()` Liefert ein Array mit den Links der Response zurück
- `getElementsWithName(String name)`, `getElementsWithID(String id)` Diese Funktionen liefern jeweils ein Array von HTML Elementen zurück.

5.1.3 Fazit

Durch diese Methoden kann der Benutzer von HTTPUnit sehr komfortabel und sehr genau Tests für WebSites erstellen. Zu berücksichtigen ist wie allen anderen JUnit Tests auch: Die Qualität der Tests hängt von der Qualität und Genauigkeit der Assertions ab!

5.2 HTMLUnit

5.2.1 Überblick

HTMLUnit[3] ist HTTPUnit sehr ähnlich. Auch HTMLUnit ist ein Testtool, das stark an JUnit angelehnt ist, beziehungsweise in dieses integriert ist. Auch HTMLUnit soll es ermöglichen Web Sites oder Web Applikationen automatisiert zu testen indem es einen Benutzer am Browser imitiert.

Der Unterschied zwischen HTMLUnit und HTTPUnit ist der grundlegende Ansatz bei der Implementierung: HTTPUnit konzentriert sich auf das Request - Response Pattern, wobei HTMLUnit sich speziell auf die Response, also das retournierte HTML Dokument und dessen Inhalt konzentriert.

Die ursprüngliche Entwicklung stammt von Mike Bowler, der HTMLUnit unter der Apache License veröffentlichte.

5.2.2 Verwendung

Die Verwendung von HTMLUnit gleicht der von HTTPUnit. Genauso wie vorher muss der Benutzer erst sein JUnit Testframework aufbauen um danach HTMLUnit einbauen zu können. Auch hier eine große Liste von Anhängigkeiten zu berücksichtigen. Die Homepage[4] gibt darüber detailiert Auskunft.

Hat ein Benutzer nun seine JUnit TestSuite und Testklassen erstellt, kann er daran gehen seine HTMLUnit test - Methoden zu schreiben. Auch hier sind starke Parallelen zu HTTPUnit sichtbar, die das folgende Beispiel verdeutlicht:

```
public void testHomePage() throws Exception {
  WebClient webClient = new WebClient();
  URL url = new URL("http://htmlunit.sourceforge.net");
  HtmlPage page = (HtmlPage)webClient.getPage(url);
  assertEquals( "htmlunit - Welcome to HtmlUnit", page.getTitleText() );
}
```

[3] Homepage von HTMLUnit: http://htmlunit.sourceforge.net
[4] http://htmlunit.sourceforge.net/dependencies.html

Dieses Beispiel stammt von der HTMLUnit Homepage und macht folgendes:

Ähnlich den Klassen `WebConversation`, `WebRequest` und `WebResponse` stellt hier `WebClient` den zentralen Dreh- und Angelpunkt des Tests dar. WebClient ist im Unterschied zu HTTPUnit dem Browser eines Benutzers nachempfunden. Durch die Methode `getPage(URL url)` wird die Seite, die durch die URL `url` definiert wurde geholt. Mit `HtmlPage page` wird ein neues HtmlPage Object erzeugt. Durch einen Cast wird dann das Ergebnis von getPage(url) ein HtmlPage Objekt.

Jede weitere Verarbeitung Seite passiert auf diesem HtmlPage Objekt.

Zur weiteren Verarbeitung bietet HTMLUnit sehr viele praktische Methoden. Hier ein kleiner Auszug, der Methoden der Klasse `HtmlPage`

- `getForms()` Liefert eine Liste aller Forms dieser Seite zurück
- `getFormByName(String name)` Liefert das Form Element mit dem Namen ¡name¿ zurück
- `getFrameByName(String name)` Gibt den Frame mit dem Namen ¡name¿ zurück
- `getTabbableElements()` Falls gibt eine Liste aller Tab - Elemente in der Reihenfolge ihrer Darstellung auf der Seite zurück

Weiters ist bei der Verarbeitung zu bemerken, dass `HtmlPage` ein DOM Element ist, was die Navigation in der Page erleichtert.

Eine weitere Besonderheit von HTMLUnit ist die besonders elegante Handhabung von Table Elementen. Für diese HTML Elemente bietet HTMLUnit einen Iterator und zusätzlich direkten Zugriff über eine Methode getCell(x,y), wobei x, die Spalte und y die Zeile einer Zelle angibt. Ein weiteres Beispiel, das auch auf der Homepage[5] zu finden ist, soll dies verdeutlichen:

Table Interator

```
HtmlTable table = (HtmlTable)page.getHtmlElementById("table1");
List rows = table.getRows();
for (Iterator rowIterator = rows.iterator(); rowIterator.hasNext(); ) {
    HtmlTableRow row = (HtmlTableRow) rowIterator.next();
    System.out.println("Found row");
    List cells = row.getCells();
    for (Iterator cellIterator = cells.iterator(); cellIterator.hasNext();) {
        HtmlTableCell cell = (HtmlTableCell) cellIterator.next();
```

[5] http://htmlunit.sourceforge.net/table-howto.html

```
        System.out.println("   Found cell: " + cell.asText());
    }
}
```

Direkter Zugriff auf Table Zellen

```
WebClient webClient = new WebClient();
HtmlPage page = (HtmlPage)webClient.getPage( new URL("http://foo.com") );

HtmlTable table = (HtmlTable)page.getHtmlElementById("table1");
System.out.println( "Cell (1,2)="+ table.getCellAt(1,2);
```

5.2.3 Fazit

HTMLUnit bietet sehr viel Komfort beim Verarbeiten Webseiten, da es für sehr viele HTML Komponenten Klassen bereitstellt, die derern Funktionalität verfügbar machen. Außerdem ist der Zugriff auf die Seiten sehr einfach gestaltet und der Benutzer muss sich nicht über Request und Response den Kopf zerbrechen. Dies ist auch eine klare Einschränkung, da ein Pageflow nicht gut dargestellt werden kann.

Vergleicht man HTMLUnit mit HTTPUnit so sind sehr große Überschneidungen erkennbar. HTTPUnit hat durch sein grundlegendes Design den Vorteil der etwas größeren Flexibilität. Dafür leidet hier der Komfort und die Reichhaltigkeit beim Arbeiten mit HTML Elementen einer Seite. Hier kann HTMLUnit ganz klar seine Stärken ausspielen.

5.3 JMeter

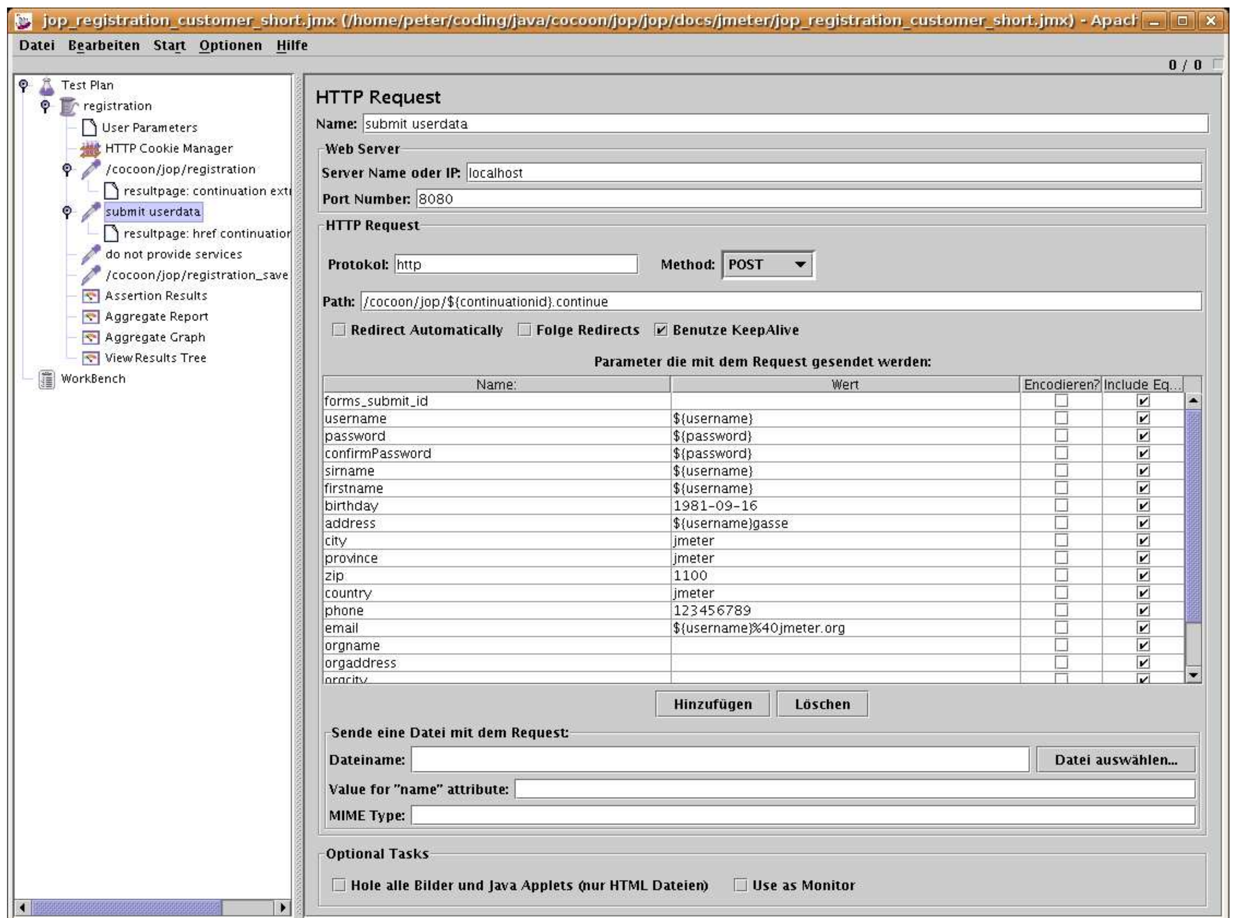
5.3.1 Überblick

JMeter unterscheidet sich ein wenig von den übrigen Testtools, da es sich in erster Linie um ein Werkzeug handelt, das entwickelt wurde um Lasttests durchzuführen. Trotzdem kann man auch funktionale Tests durchführen.

Vor allem im Schreiben der Testfälle unterscheidet sich JMeter sehr stark. Die Testfälle werden nämlich nicht in Ruby oder Java geschrieben sondern in xml. Da diese xml Files recht schnell sehr komplex werden gibt es ein sehr gutes Frontend für JMeter mit dem der Benutzer die Tests definieren kann.

JMeter hat gegenüber der Konkurrenz vor allem dann einen sehr großen Vorteil, wenn nachträglich Tests erstellt werden sollen. Dieser Vorteil besteht darin, dass JMeter bietet eine Recording Funktion besitzt. Mit dieser Recording Funktion können Testfälle nach dem Record / Play Prinzip erstellt werden. Dabei muss ein Benutzer nur noch einmal die zu testende Funktionalität selbst ausprobieren und dabei aufnehmen um dann später diesen

Testablauf immer und immer wieder abspielen lassen zu können. Die Funktionsweise dieses Recording Mechanismus ist denkbar einfach. JMeter kann als Proxy laufen und alle Requests mitschneiden und gleichzeitig auch filtern. So können durch einfaches Benutzen der zu testenden Webapplikation Testfälle erstellt werden.



5.3.2 JMeter für Frontend Tests

Trotzdem JMeter ein Tool für Lasttests ist, kann es für Frontend - Tests benutzt werden. Dabei muss der Benutzer aber berücksichtigen, dass JMeter ursprünglich nicht dafür konzipiert war. Aus diesem Grund müssen ein paar wenige Grundkonzepte verstanden und berücksichtigt werden.

Startet man Jmeter ganz normal, so erscheint ein nahezu leeres GUI. Mit diesem User - Interface können die Tests definiert werden. Im linken Teil des Fensteres sind zwei Einträge zu finden: Test Plan und Workbench.

Workbench ist ein Platz an dem verschiedenste Elemente zwischengespei-

chert werden können. Diese werden nicht ausgeführt und auch nicht gespeichert. Typischerweise wird dort zum Beispiel der Recording Controller von JMeter dort platziert, oder Elemente von JMeter, die bereits konfiguriert wurden, aber momentan nicht benötigt werden.

Test Plan stellt den Wurzelknoten für alle Tests, die später noch definiert werden, dar. Unter dem Testplan kommen alle Testfälle, die in einer Testgruppe gemeinsam ausgeführt werden sollen.

Ein funktionaler Test in JMeter besteht aus vier wichtigen Komponenten:

1. Thread Group
2. Sampler
3. Assertion
4. Listener

Thread Group

Da JMeter ein Lasttesttool ist, bietet es die Möglichkeit multithreaded Anfragen an ein bestimmtest Ziel zu schicken. So können gleichzeitig Anmeldung und Abmeldung von vielen unterschiedlichen Nutzern simuliert werden. Für funktionale Tests ist es nicht notwendig 1000 Mal das gleiche Szenario durchzuspielen. Es soll ja nur die korrekte Funktionsweise überprüft werden. Aus diesem Grund wird für jeden neuen Testfall eine Threadgroup erstellt, deren Thread - Properties alle auf 1 gestellt werden.

[screenshot jmeter thread group]

- Number of Threads (users): Wie viele Threads gemacht werden sollen, die alles, was in dieser Threadgroup definiert ist ausführen.
- Ramp-Up Period (in seconds): Wie lange JMeter brauchen soll, die Anzahl von Threads zu erzeugen.
- Loop Count: Wiederholung des gesamten Vorgangs

Beispiel:
Number of Threads: 1000
Ramp-Up Period: 6000
Loop Count: 1

Hier werden 1000 Threads von dieser Threadgroup innerhalb von 6000 Sekunden erstellt und gestartet. Also pro Minute 10 Threads. Das entspricht einem Thread alle 6 Sekunden.

Innerhalb einer Threadgroup wird alles definiert, was diese Threadgroup ausführen und überprüfen soll. Im einfachsten Fall handelt es sich dabei um einen Sampler und eine Assertion.

Sampler

Ganz allgemein ist die Aufgabe eines Samplers an ein Zielsystem Anfragen zu richten. Dies kann ein simples HTTP Request sein, oder auch eine Anfrage an ein WebService oder ein FTP Request oder auch ein JDBC Request. JMeter unterstützt eine sehr große Anzahl an Samplern, wie zum Beispiel einen HTTP Request Sampler für HTTP Requests oder einen JDBC Request Sampler für Tests von Datenbanken über die JDBC Schnittstelle.

Für funktionale Tests ist es sehr hilfreich den einzelnen Samplern sinnvolle Namen zu geben. So kann später im Reporting leichter herausgefunden werden welche Tests fehlschlugen.

Assertions

Will man funktionale Tests durchführen muss man natürlich überprüfen, ob die gestellte Anfrage auch das erwartete Ergebnis zurückliefert. Dazu bietet JMeter Assertions. Assertions prüfen das Ergebnis eines Requests auf unterscheidlichste Art und Weise, je nachdem um welche Assertion es sich handelt. JMeter unterstützt einfachste Response Assertions, die die Anwort auf ein Request nach bestimmten Zeichenfolgen durchsuchen können. Es sind aber auch komplexere Assertions, wie zum Beispiel XPath Assertions, oder Size Assertions oder Duration Assertions möglich. Letztere untersuchen die Response auf deren Größe und ihre Antwortzeit.

Es gibt im Allgemeinen sehr viele Methoden solche Assertions zu definieren. Es liegt dabei im Ermessen des Erstellers der Tests die Robustheit und Genauigkeit solcher Assertions zu definieren und so ein sinnvolles Verhältnis zwischen Aufwand und Genauigkeit für jede einzelne Assertion herzustellen.

Listener

Nachdem Tests durchgeführt worden sind, muss natürlich noch in geeigneter Form ein Reporting stattfinden, damit der Tester erfährt, welche Tests geklappt haben und welche nicht. Dazu werden in JMeter Listener eingesetzt. Diese Listener können die durchschnittliche Antwortzeiten, Request Results, etc. anzeigen.

Natürlich sind nicht alle Listeners für funktionale Tests passend, hier muss der Tester eine Auswahl treffen. Als sehr praktisch hat sich der Listener *View Results Tree* erwiesen, da dieser für jedes Request den Inhalt und auch de der dazugehörigen Response anzeigen kann. Dies ist fürs Debugging manchmal sehr hilfreich.

In jedem Fall sollte der Tester den Assertion Results Listener verwenden. Dieser Listet alle Assertions auf. Sollte man die Übersicht verlieren, kann man

die Option *Log Errors Only* aktivieren. Dann werden nur die fehlgeschlagenen Assertions angezeigt.

Mit diesen vier Bausteinen ist es bereits möglich einfache funktionale Tests mit JMeter durchzuführen. Zusätzlich verfügt JMeter noch über weitere sehr praktische Bausteine. Die wichtigsten, die auch in den folgenden Tests verwendet wurden, sollen hier kurz erklärt werden.

PreProcessor

Ein PreProcessoer wird, wie der Name vermuten lässt, vor einem Sampler verarbeitet. Dieser PreProcessor ermöglicht es gewisse Veränderungen dynamisch vorzunehmen, bevor beispielsweise ein HTTP Request abgefeurt wird.

JMeter bietet sieben verschiedene PreProcessors. Hier wird exemplarisch der User Parameters PreProcessor beschrieben, der später in den Beispielen auch Verwendung findet.

Mit dem *User Parameters PreProcessor* ist es möglich Variablen zu definieren, die später von anderen Komponenten der selben Thread Group verwendet werden können. *Der User Parameters PreProcessor* besitzt zwei Parameter. Zum einen kann sein Name definiert werden und zum anderen einen boolean Value: Update Once Per Iteration. Ist dieses Kontrollfeld aktiviert, wird sichergestellt, dass für jede neue Iteration die Werte verändert werden. Dazu werden für jeden Request der in der gleichen Gruppe, wie der User Parameters PreProcessor ist, neue Werte verwendet. Falls die Anzahl der Threads, die Anzahl der Werte übersteigt, so werden die gleichen Werte erneut verwendet. Die Variablen und deren Werte werden folgendermaßen definiert:

Name	Thread_1	Thread_2	Thread_3
variable1	variable1_wert1	variable1_wert2	variable1_wert3
variable2	variable2_wert1	variable2_wert2	variable2_wert3

PostProcessor

Ein PostProcessor ist einfach ausgedrückt das Gegenstück zu einem PreProcessor. Seine Aufgabe ist also die Antworten der Systeme an die JMeter seine Anfragen schickt zu verarbeiten.

Am einfachsten kann man das am Beispiel HTTP sehen. Hier schickt JMeter ein Request und erwartet eine Response von dem Zielsystem. Kommt eine Response, so wird die vom PostProcessor verarbeitet.

JMeter stellt verschiedene PostProcessoren bereit, die ganz unterschiedliche Aufgaben übernehmen können:

- *Regular Expression Extractor*: Dieser PostProcessor kann den Body, den Header und die URL parsen und ein Matching anhand von Regular Expressions durchführen. Die Ergebnisse dieser Matchings werden dann in Variablen gespeichert und stehen zur weiteren Verarbeitung zu Verfügung. In den Beispielen weiter unten wird der Regular Expression Extractor PostProcessor verwendet um die aktuelle Continuation ID aus der Response herauszufiltern und diese mit dem nächsten Request mitzuschicken.

- *Save Responses To File*: Der Save Responses To File Post Processor speichert eine Response in eine Datei. Dieser PostProcessor ist sehr einfach aufgebaut und bietet nur zwei Möglichkeiten der Konfiguration:

  1. Name: Hier kann man dem PostProcessor einen Namen geben, um die Lesbarkeit der Tests zu erhöhen.
  2. Filename Prefix: hier kann man ein Prefix definieren. Dieses Prefix wird dem Dateinamen vorangestellt und erlaubt so eigene Dateinamen. Der Dateiname wird folgendermaßen aufgebaut:

     `<FilenamePrefix><Nummer>.<Endung>`

     Die Endung wird aus dem DocType der Response generiert und automatisch erstellt. Darauf hat der Benutzer keinen Einfluss. Vor der Endung wird ein Punkt eingefügt. Vor diesem Punkt wird dann eine Nummer eingefügt. Auch diese wird automatisch erstellt. Sie representiert die Reihenfolge der Responses. Vor dieser Nummer wird das vom Benutzer konfigurierte Prefix eingefügt.

     Die Dateien, die die gespeicherten Responses enthalten werden immer in das Verzeichnis gespeichert, von dem aus JMeter gestartet wurde. Es gibt aber einen kleinen Trick, wie man dieses Verzeichnis ändern kann. Es ist möglich als Prefix einen Pfad zu definineren, dann wird die Response in woanders hin gespeichert. Gibt man beispielsweise als Prefix

     `/home/peter/mytest/responses/LoginTest_`

     an wo werden sämtliche Responses in das Verzeichnis

     `/home/peter/mytest/responses/`

     gespeichert und ihre Dateinamen beginnen mit der Zeichenfolge

     `LoginTest_`

HTTP Cookie Manager Viele Web Applikationen brauchen um korrekt funktionieren zu können Cookies. Diese werden zum Beispiel benötigt

um eine Servlet Session halten zu können oder den Inhalt eines Warenkorbs zu speichern.

Jmeter bietet mit dem HTTP Cookie Manager eine Komponente an, die es Jmeter ermöglicht Cookies wie ein Browser zu senden und zu empfangen. Damit können Cookies die in Responses mitgeschickt werden abgespeichert werden und so mit dem nächsten Request mitgeschickt werden.

Mit diesen Cookies kann man natürlich auch weiterarbeiten. Sie werden nämlich auch als Variablen gespeichert. Als Variablennamen wird der Name des Cookies herangezogen. Heißt ein Cookie beispielsweise jmetercookie, so erfolgt der Zugriff wie bei JMeter üblich in dem Format: `${jmetercookie}`.

Berücksichtigen muss man als Benutzer jedoch den Scope in dem man auf diese Cookies zugreifen kann. Der ist nämlich nur im selben Thread möglich.

Der HTTP Cookie Manager ermöglicht aber nicht nur das Speichern und Senden von erhaltenen Cookies. Er bietet auch die Möglichkeit an selbst erstellte Cookies manuell hinzuzufügen. Auf solche Cookies ist der Zugriff dann aus allen Threads möglich.

5.4 JMeter am Beispiel JOP

Um zu zeigen, wie mittels JMeter funktionale Tests durchgeführt werden können, werden hier verschiedene konkrete Beispiele beschrieben. Diese beginnen bei einfachsten Beispielen, die lediglich ein HTTP Request und eine Assertion beinhalten und werden nach und nach zu einem komplexen Beispiel erweitert, das einen gesamten Anmeldevorgang bei einer bestehenden Web Applikation simuliert.

5.4.1 Überblick

Die folgenden Beispiele verwenden eine bestehende Web Applikation. Diese trägt den Namen JOP und ist eine Jobbörse. Entwickelt wurde die Webapplikatikon mit Cocoon, Hibernate und Spring. Zur persistenten Speicherung der Daten dient die relationale Datenbank HSQLDB. In diesem Beispiel wird nicht die gesamte Webapplikation getestet, sondern zwei representative Teile:

1. Die Registrierung: Neue Anmeldung eines Kunden
2. Login/Logout: Login bzw. Logout eines bestehenden Kunden

Diese beiden Teile beinhalten bereits alle Problemstellungen, die beim Testen der übrigen Komponenten auftreten können. Es handelt sich dabei um die folgenden Punkte:

- HTTP Requests absetzen
- HTTP Response prüfen
- Request Parameter Handling
- Session bewahren, die in einem Cookie gespeichert wird
- Berücksichtigung und Behandlung von Cocoon Continuations

Um nun mit JMeter Tests durchzuführen muss erstens JMeter laufen und zweitens die Webapplikation. In den folgenden Tests laufen beide Systeme auf dem selben Rechner. Die Webapplikation ist läuft innerhalb des Servlet Containers Tomcat unter Port 8080 und ist erreichbar unter der Adresse *http://localhost:8080/cocoon/jop/*.

5.4.2 Test 1: Aufruf der Startseite

In diesem ersten Test wird lediglich die Startseite aufgerufen und anhand einer Zeichenfolge, die nur auf dieser Seite vorkommt überprüft, ob die Startseite korrekt aufgerufen wurde.

Dazu werden in JMeter vier Teile benötigt:

1. Thread Group: Die Thread Group beinhaltet den HTTP Request Sampler. Ihr wird der Name ¨jop start¨zugewiesen und alle Thread Properties auf 1 gesetzt. Dieser Thread Group werden dann als Kinder ein HTTP Request Sampler und eine Response Assertion zugewiesen.

2. HTTP Request Sampler: Der HTTP Request Sampler hat die Aufgabe ein HTTP Request an die Webapplikation zu stellen. Die Konfiguration des Samplers sieht folgendermaßen aus:

   (a) Name: call startpage
   (b) Server Name of IP: localhost
   (c) Port Number: 8080
   (d) Method: GET
   (e) Path: /cocoon/jop/start

3. Response Assertion: Die Response Assertion durchsucht die HTTP Response nach der Zeichenfolge ¨News¨. Diese kommt nur auf der Startseite von JOP vor und ist somit einigermaßen representativ dafür. Findet die Assertion die Zeichenfolge News in der Response nicht, so wird der Test als failed gekennzeichnet.

   Konfiguration:

- Response Field to Test: Text Response
- Pattern Matching Rules: contains
- Patterns to Test: News

4. Listener 1: Show Results Tree: Hier ist keine Konfiguration nötig. Nach dem Starten des Tests kann das HTTP Request und die HTTP Resonse angesehen werden. Bei einem Problem, oder falls die Assertion den Test als "failed"kennzeichnet wird der Name des Samplers (hier: call startpage) rot dargestellt.

5. Listener 2: Assertion Results: Hier werden alle fehlgeschlagenen Assertions aufgelistet.

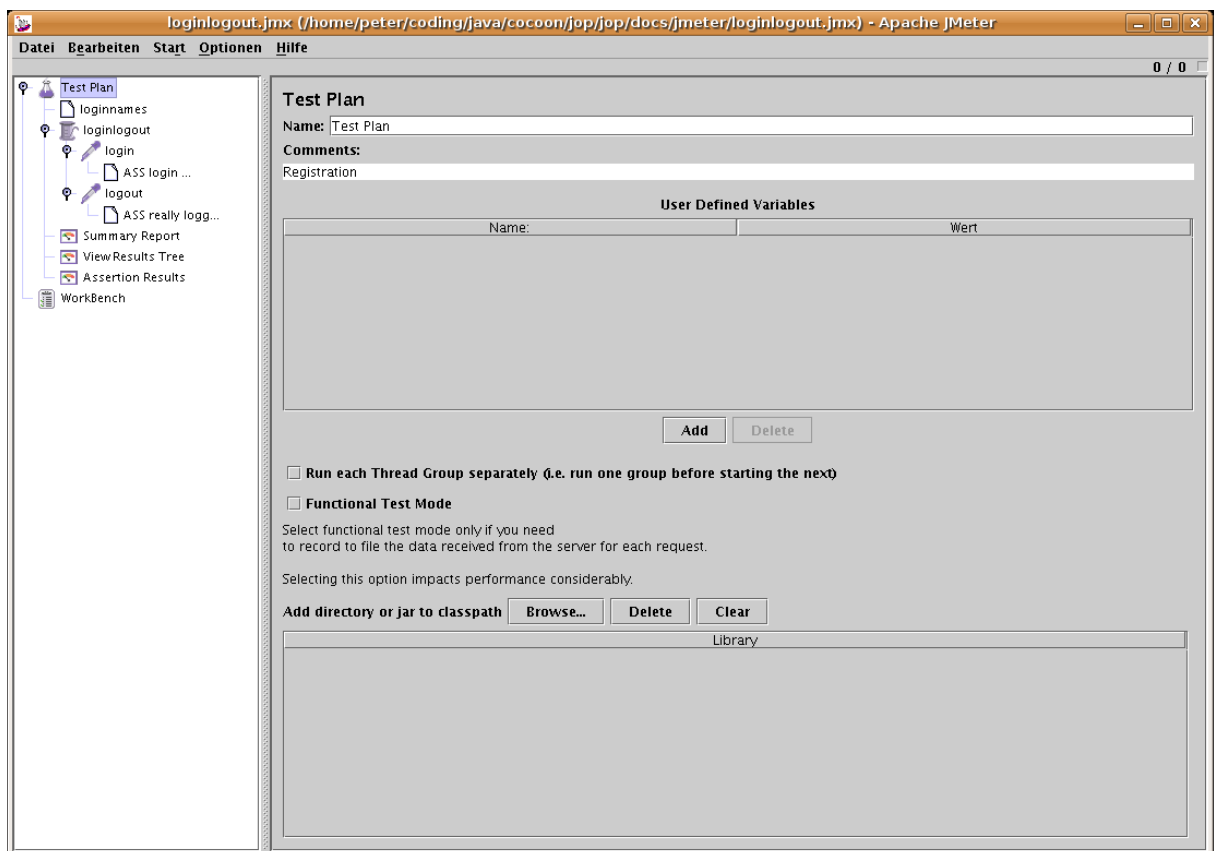
5.4.3 Test 2: Login/Logout

Im nächsten Test wird das Login und das Logout getestet. Der Vorgang entspricht dem von vorher mit zwei Änderungen:

1. Es werden Request Parameter verwendet

2. Innerhalb von JUnit werden Variablen verwendet

Bei diesem Test wird der Login von drei verschiedenen Benutzern getestet. Es werden drei Threads erzeugt, die dann jeweils einen Benutzer einloggen und danach wieder ausloggen. Die Konfiguration von JMeter entspricht der von vorher nur werden jetzt zwei HTTP Request Sampler eingesetzt, die für das Login und das Logout zuständig sind. Zusätzlich wird noch der User Parameters PreProcessor hinzugefügt. Dieser enthält die Logindaten der einzelnen Benutzer. Entsprechend ist die Baumstruktur und die Konfiguration von JMeter in diesem Beispiel etwas aufwendiger:



1. PreProcessor User Parameters: Dieser Preprocessor enthält die Username/Password - Kombinationen, die später als Request Parameter übergeben werden sollen. Das sind zwei Variablen: username und password für drei User: User_1, User_2, User_3

   Konfiguration:

   - Name: loginnames
   - Update Once Per Iteration: ausgewählt
   - Name: username, password
   - User_1: jmeter, jmeter
   - User_2: peter, peter
   - User_3: hiasi, hiasi

   Diese Werte werden vor dem Ausführen der Sampler geladen und in den Variablen username und password gespeichert.

2. Thread Group: Um nun den Test für drei Benutzer druchführen zu können wird die Anzahl der Threads auf drei erhöht. Ebenso wird auch der Name der Threadgroup angepasst. die anderen Einstellungen bleiben gleich, so wie im vorigen Beispiel:

   Konfiguration:

   - Name: loginlogout
   - Number of Threads: 3
   - Ramp-up Period: 1
   - Loop Count: 1

3. HTTP Request Sampler (login): Der erste HTTP Request Sampler versucht einen Benutzer einzuloggen. Dazu müssen zwei Parameter mittels eines HTTP POST Request an die Webapplikation übergeben werden.

   Konfiguration:

   - Name: login
   - Server Name: localhost
   - Port Number: 8080
   - Path: /cocoon/jop/do-login
   - Method: POST
   - Send Parameters with Request
     - Name: username; Value: ${username}; encode: no; include equals: yes;
     - Name: password; Value: ${password}; encode: no; include equals: yes;

   Hier wird dem Request Parameter *username* die Variable `${username}`, die im PreProcessor definiert wurde, übergeben. Gleiches gilt für den Request Parameter *password*.

4. Assertion Login: Bei der Assertion wird überprüft, ob die Response *nicht* die Zeichenfolge failedënthält. Nach dieser Zeichenfolge wird aus einem besonderen Grund gesucht:

   Gibt ein Benutzer eine falsche Benutzername/Passwort - Kombination ein, so wird ihm Fehlermeldung angezeigt, die dieses Wort enthält.

   Wird nun in der Antwort diese Zeichenfolge gefunden, so kann der Login - Prozess als fehlerhaft angesehen werden und damit der Test als f̈ailedg̈ekennzeichnet werden.

Eine andere Möglichkeit wäre, in der Response nach dem übergebenen Benutzernamen zu suchen, weil dieser nach dem Login angezeigt wird. Findet man diesen in der Response nicht, so kann der Test ebenfalls als failedängesehen werden.

Konfiguration:

- Name: ASS login failed
- Respose Field to Test: Text Response
- Pattern matching rules: contains, not
- Patterns to Test: failed

5. HTTP Request Sampler (logout): Dieser Sampler ist ein herkömlicher HTTP Request Sampler, wie er aus dem ersten Test schon bekannt ist. Er ruft eine URL auf, die den aktuell angemeldeten Benutzer ausloggt.

   Konfiguration

   - Name: logout
   - Server Name: localhost
   - Port Number: 8080
   - Path: /cocoon/jop/do-logout
   - Method: GET

6. Assertion Logout: Bei dieser Assertion wird überprüft, ob das Logout tatsächlich geklappt hat. Dabei wird untersucht ob in der Response der Benutzername noch enthalten ist, da dieser für gewöhnlich nach dem Login angezeigt wird. Der Benutzername wird erneut der Variable `${username}` entnommen.

   Konfiguration

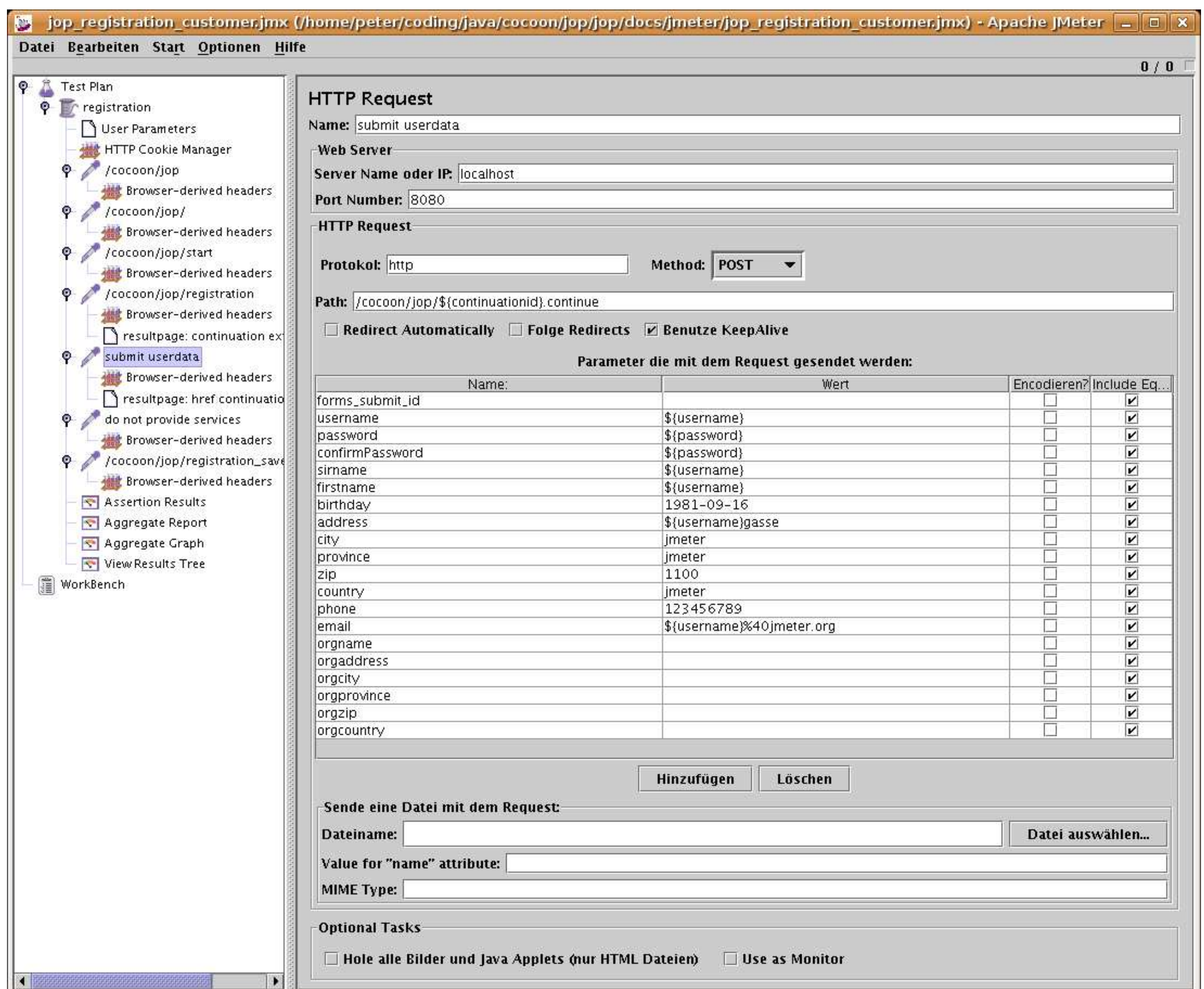
   - Name: ASS really logged out
   - Respose Field to Test: Text Response
   - Pattern matching rules: contains, not
   - Patterns to Test: ${username}

Hier sieht man schön, dass Variablen nicht nur Request Parametern übergeben werden können, sondern in JMeter an unterschiedlichsten Stellen verwendet werden können.

5.4.4 Test 3: Registrierung

Der folgende Test führt mehrere Registrierungen durch. Um dies durchführen zu können müssen mehrere Formulare ausgefüllt werden. Beim Testen muss zwischen den einzelnen Seiten der Registrierung die Session gehalten werden und ebenso Cocoon Continuations berücksichtigt werden. Die Session wird mit Hilfe von Cookies gehalten. Dafür wird ein Jmeter dann eine eigene Komponente verwendet: der *HTTP Cookie Manager*.

Das Continuation Handling bei Jmeter ist ein wenig aufwendiger. Dafür wird die *Regular Expression Extractor PostProcessor* Komponente von JMeter verwendet. Aus diesen Gründen ist auch der Aufbau des Tests etwas kmplexer als zuvor.



Im folgenden Teil werden die einzelnen Teile beschrieben und ihr Zweck erläutert.

- registration: Thread Group: Hier wird allgemein ein Thread definiert, der einen Benutzer anlegt. Alle Aktionen die innerhalb dieser Thread Group liegen, werden nach dem Start innerhalb eines Threads nacheinander ausgeführt.

  Diese ThreadGroup erzeugt drei gleiche Threads, die demnach drei Benutzer anlegen. Da die Benutzer unterscheidliche Benutzernamen besitzen müssen, werden drei verschiedene Benutzer angelegt. Die Parameter der Einzelnen Benutzer werden im nächsten Punkt (User Parameters) beschrieben.

  Parameter:

  – Name: registration
  – Number of Threads: 3
  – Ramp-up Period: 1
  – Loop Count: 1

- User Parameters: Hier werden die Benutzernamen und Passwörter für die einzelnen User, die angelgt werden sollen gespeichert. Nach einem erfolgreichen Testlauf müssen sich dann Benutzer ganz normal mit diesen Credentials einloggen können.

  Parameter:

Name	User_1	User_2	User_3
username	jmeter10	jmeter11	jmeter12
password	jmeter10	jmeter11	jmeter12

  Hier werden die Parameter für Benutzername und Passwort definiert. Nach einem erfolgreichen Test, ist ein neuer Benutzer angelegt und er kann sich mit dem Benutzernamen *jmeter10* und dem Passwort *jmeter10* anmelden. Das gleiche gilt für die zwei anderen Benutzer *jmeter11* und *jmeter12*, die hier konfiguriert wurden.

- HTTP Cookie Manger: Da Cocoon Session Informationen in Cookies speichert ist es nötig einen HTTP Cookie Manger einzufügen. Dieser HTTP Cookie Manager ermöglich das Speichern und Senden von Cookies, wie es sonst ein Browser machen würde.

  Parameter:

  – Name: HTTP Cookie Manager
  – Clear cookies each iteration: unchecked

- Cookie Policy: compatibility
- Cookies gespeichert im Cookie Manger: Keine eigenen Cookies laden.

- /cocoon/jop/registration: HTTP Request: Dieses Request ruft die erste Seite der Registrierung auf und stößt damit den Registrierungsprozess an. Als Response wird eine Seite geschickt, zwei wichtige Elemente enthält:

  1. Ein großes Formular Element wo der Benutzer Anageben zur Person machen muss.
  2. Die aktuelle Continuation Id als form action:

```
<form
    xmlns:fi="http://apache.org/cocoon/forms/1.0#instance"
    action="8d89693f72554a11001f742617717840603974l5.continue"
    method="POST"
    onsubmit="forms_onsubmit(); "
>
```

  Der Inhalt dieser Action muss mit dem nächsten Request mitgeschickt werden, da sonst das cocoon Continuation Handling fehlschlägt und cocoon den Registrierungsvorgang mit einem Fehler abbricht.

  Zum Extrahieren dieser Daten wird der *Regular Expression Extractor PostProcessor* verwendet. Mit ihm wird die Response auf eine Zeichenfolge durchsucht, die mit *form=¨* beginnt und mit *.continue¨* endet. Das Ergebnis wird in der Variable `continuationid` gespeichert.

  Die Grundidee dieser Form des Handlings von Cocoon Continuations stammt aus dem Cocoon Wiki[6]. Und wurde von dem User CODonovan entwickelt.

- Regular Expression Extractor Konfiguration:

  - Name: resultpage: continuation extractor;
  - Response Field to check: Body; Der Body soll mit Regular Expression Matching druchsucht werden.
  - Reference Name: continuationid; Hier wird der Name der Variable gesetzt, wo die aktuelle Contnuation ID gespeichert wird.

[6] http://wiki.apache.org/cocoon/LoadTestingFlowWithJMeter

- Regular Expression:

  ```
  action=\"([^"]*)\.continue\"
  ```

  Regular Expression nach das Matching stattfindet

- Template: 1; Hier wird definiert wie die gefundenen Matchings in Variablen gespeichert werden sollen. 1 bedeutet, dass die erste Gruppe, die gematcht hat genommen wird. 2 stünde für die zweie Gruppe und so weiter. Wollte man alles was matchen würde in der Variablen speichern, so wäre das Template dazu: 0
- Match No.: 1; Das bedeutet, dass nur das erste Matching genommen wird. 0 würe bedeuten, dass ein zufälltiges ausgewähltes matching in der Variable continuationid gespeichert würde
- Default Value: error; Dieser Ausdruck wird übergeben, wenn kein matching gefunden werden konnte

• submit userdata: HTTP Request: In diesem HTTP POST Request werden alle persönlichen Daten mittels HTTP Post übergeben. Die einzelnen Daten werden als HTTP Parameter übergeben, für jedes Feld ein eigener Parameter. Um das Beispiel einfach zu halten und um die angelegten Datensätze in der Datenbank leichter identifizieren zu können, wird an so vielen Stellen wie möglich die Variable texttt${username} übergeben.

  Besonders hervorzuheben ist, dass auch im Pfad des HTTP Requests eine Variable übergeben wird. Diese Variable heißt ${continuationid} und enthält die Cocoon Continuation ID, die aus der letzten Response extrahiert wurde.

  Konfiguration:

  - Name: submit userdata
  - Server Name or IP: localhost
  - Port Number: 8080
  - Protocol: http
  - Method: POST
  - Path: /cocoon/jop/${continuationid}.continue
  - Use KeepAlive: ja

- Send Parameters with the Request: Diese Parameter werden mit dem POST mitgeschickt und enthalten die Daten des Benutzers:

Name	Value	Encode	Include Equals
forms_submit_id			x
username	${username}		x
password	${password}		x
confirmPassword	${password}		x
firstname	${username}		x
sirname	${username}		x
birthday	1981-09-16		x
address	${username}gasse		x
city	jmeter		x
province	jmeter		x
zip	1100		x
country	jmeter		x
phone	123456		x
email	${username}%40jmeter.org		x
orgname			x
orgaddress			x
orgcity			x
orgprovince			x
orgzip			x
orgcountry			x

- resultpage: href continuationid extractor: Regular Expression Extractor Die Response auf das letzte HTTP POST Request ist wiederum eine HTML Seite, die eine Continuation ID enthält. Auch diese muss für das nächste Request extrahiert werden. Dies geschieht nach dem gleichen Schema wie zuvor. Lediglich die Contination ID ist dieses Mal an einer anderen Stelle gespeichert und entsprechend muss die Regular Expression leicht angepasst werden. Alle anderen Einstellungen sind wie zuvor.

  Konfiguration:

  - Name: resultpage: href continuationid extractor
  - Regular Expression:

    ```
    href=\"([^"]*)\.continue.*\"
    ```

  - Template 1
  - Match No.: 1

  - Default Value: error

- do not provide services: HTTP Request Hier wird mittels eines einfachen HTTP GET Requests mitgeteilt, dass der Benutzer keine Services anbietet. Dies geschieht indem dem HTTP Parameter des GET Requests names `provider` der Wert `no` übergeben wird. Alle anderen Parameter dieses JMeter HTTP Requests sind wie gewohnt:

  Konfiguration:

  - Name: do not provide services
  - Server Name or IP: localhost
  - Port Number: 8080
  - Protocol: http
  - Method: GET
  - Path: /cocoon/jop/${continuationid}.continue
  - Use KeepAlive: ja
  - Send Parameters with the Request: Dieser Parameter wird mit dem GET übermittelt und bestimmt, dass der Benutzer keine Services bei JOP anbieten wird:

Name	Value	Encode	Include Equals
provider	no		x

- /cocoon/jop/registration_save: HTTP Request Die Response des letzten Requests ist eine HTML Seite, die sämtliche Angaben, die der Benutzer in den Schritten zuvor gemacht hat noch einmal anzeigt. Hier kann der Benutzer noch einmal seine Angaben verifizieren und falls nötig zurück gehen um Änderungen vorzunehemen. Ist er von der Richtigkeit seiner Angaben überzeugt, klickt er auf *Save*.

  In diesem letzten HTTP Reqest wird genau dieser Schritt, also das klicken auf Save imitiert.

  Konfiguration:

  - Name: /cocoon/jop/registration_save
  - Server Name or IP: localhost
  - Port Number: 8080
  - Protocol: http
  - Method: GET

– Path: /cocoon/jop/registration_save
– Use KeepAlive: ja

Alle diese Bestandteile und Konfigurationen sind nötig um mittels JMeter einen neuen Benutzer in JOP zu registrieren.

5.5 JMeter Tests ausführen

Nachdem jetzt eingehend geschildert wurde wie unterschiedliche Tests für JMeter aufgebaut und konfiguriert werden können soll nun natürlich auch beschrieben werden, die diese Tests ausgeführt werden können.

Es gibt drei Möglichkeiten JMeter Tests ausführen zu lassen.

1. **Im JMeter GUI**

   Die einfachste Methode JMeter Tests durchzuführen, ist sie direkt über das GUI zu starten. Dazu muss der Benutzer lediglich in der Menüleiste das Menü *Start* wählen und diesem Menü den Eintrag *Start*. Alternativ können die Tests auch durch *Strg + R* gestartet werden. Sobald der Testlauf gestartet ist, zählt der Zähler in der rechten oberen Ecke die offenen Threads und ein grünes Quadrat leuchtet so lange bis der Testlauf zu Ende ist.

2. **Von der Kommandozeile**

   Für gewisse Anwendungen kann JMeter auch von der Kommandozeile gestartet werden. Dafür wird zumindest folgende Syntax benötigt:

   ```
   java -jar bin/ApacheJMeter.jar -n -t <testplan>
   ```

   Der Parameter -n bestimmt, dass JMeter ohne GUI gestartet wird. Selbstverständlich können noch weitere Parameter übergeben werden um Einstellungen für Proxyserver, oder Logfiles zu definieren. Hier die komplette Liste aus der JMeter Hilfe Aufruf:

   ```
   java -jar ApacheJMeter.jar -n
   ```

   - -h, –help print usage information and exit
   - -v, –version print the version information and exit
   - -p, –propfile [argument] the jmeter property file to use
   - -q, –addprop [argument] additional JMeter property file(s)
   - -t, –testfile [argument] the jmeter test(.jmx) file to run

- -l, –logfile [argument] the file to log samples to
- -n, –nongui run JMeter in nongui mode
- -s, –server run the JMeter server
- -H, –proxyHost [argument] Set a proxy server for JMeter to use
- -P, –proxyPort [argument] Set proxy server port for JMeter to use
- -N, –nonProxyHosts [argument] Set nonproxy host list (e.g. *.apache.org—localhost)
- -u, –username [argument] Set username for proxy server that JMeter is to use
- -a, –password [argument] Set password for proxy server that JMeter is to use
- -J, –jmeterproperty [argument]=[value] Define additional JMeter properties
- -D, –systemproperty [argument]=[value] Define additional system properties
- -S, –systemPropertyFile [argument] additional system property file(s)
- -L, –loglevel [argument¿]=[value] [category=]level e.g. jorphan=INFO or jmeter.util=DEBUG
- -r, –runremote Start remote servers from non-gui mode
- -d, –homedir [argument] the jmeter home directory to use

3. **Als Ant - Task**

   Für alle, die für ihre Java Entwicklungen Apache Ant[7] verwenden könnte sich der JMeter Ant Task[8] als äußerst nützlich erweisen. JMeter wird bereits mit diesem Task ausgeliefert. Die nötigen Bilbliotheken und Dateien sind im Ordner *extras* zu finden.

   Eingebunden wird der Task folgendermaßen: Zuerst muss natürlich das jar-File *ant-jmeter.jar* im Classpath von Ant eingetragen sein. Danach muss man den Task in seinem Build - File definieren:

```
<taskdef
    name="jmeter"
    classname="org.programmerplanet.ant.taskdefs.jmeter.JMeterTask"/>
```

[7]http://ant.apache.org/
[8]http://www.programmerplanet.org/pages/projects/jmeter-ant-task.php

Danach kann man seinen Testplan mit folgendem Aufruf laden und starten:

```
<jmeter
    jmeterhome="c:\jakarta-jmeter-1.8.1"
    testplan="${basedir}/loadtests/JMeterLoadTest.jmx"
    resultlog="${basedir}/loadtests/JMeterResults.jtl"/>
```

Es ist natürlich auf möglich mehrere Testpläne nacheinander auszuführen, oder JMeter Properties zu setzen. Die Details sind auf der Homepage des JMeter Ant Tasks zu finden.

5.5.1 Kommandozeilen Parameter übergeben

Ein sehr praktisches Feature von JMeter ist, dass man beim Start von der Kommandozeile JMeter Parameter übergeben kann die überall in den Testplänen verwendet werden können. Die Übergabe der Parameter auf der Kommandozeile werdem mit dem Parameter `-J` ermöglicht. Der Aufruf an der Kommandozeile sieht dann folgendermaßen aus:

```
java -jar bin\ApacheJMeter.jar -n -Jparam1=setting1
```

Innerhalb eines Testplans wird dann mittels forgender Syntax der Parameter *param1* referenziert:

```
${__property(param1)}
```

5.5.2 Tests Aufzeichnen

Das praktischste Feature von JMeter ist *HTTP Proxy Server*. Mit diesem HTTP Proxy Server können Testfälle aufgezeichnet werden, was eine sehr große Erleichterung beim Erstellen von Testfällen ist.

Die Funktionsweise des HTTP Proxy Servers ist sehr simpel. Er wird zwischen Browser und JMeter geschaltet. Damit ist es ihm möglich die Requests und Responses, die ein Benutzer beim Bedienen einer WebApplikation erzeugt aufzuzeichnen.

Die Anwendung ist des HTTP Proxy Servers ist ein einem PDF - Dokuemnt, das sich auf der JMeter Homepage findet sehr gut erklärt [9]

Hier sollen noch einmal kurz die wichtigsten Schritte für die Aufzeichnung von Testsfällen beschrieben werden:

[9] http://jakarta.apache.org/jmeter/usermanual/jmeter_proxy_step_by_step.pdf

1. Thread Group erzeugen und eindeutigen Name geben: z.B.: Login
2. In Workbench bereich den HTTP Proxy Server hinzufügen (Add - Non-Test Element - HTTP Proxy Server)
3. Port wählen, z.B.: 9090
4. Target Controller: die Zuvor erstellte Thread Group auswählen. (hier: Login) Mit dieser Einstellung werden alle aufgezeichneten Requests als Kinder dieser Thread Group angelegt.
5. Im Browser die Proxy Einstellungen zu *localhost* und Port *9090* verändern
6. In JMeter bei den HTTP Proxy Server Settings start klicken.
7. Nun kann die zu Testende Applikation verwendet werden und alle Requests die dabei entstehen werden von JMeter gespeichert

Um den Output des HTTP Proxy Servers gering zu halten und auf die notwendigen Requests zu beschränken können Muster um ein- und ausschließen angegeben werden. So kann ein Benutzer zum Beispiel alle Bilder ausschließen. Dazu würde er im Feld *Patterns to Exclude*

```
.*\.gif, .*\.jpg, .*\.jpeg, .*\.png
```

als Patterns hinzufügen.

Genauso können auch Patterns, die inkludiert werden sollen definiert werden. Falls zum Beispiel alles inkludiert werden soll, so ist das passende Muster dafür: `.*`.

Nach der Aufzeichnung des Testfalls, oder der Testfälle müssen diese sogut wie immer manuell nachbearbeitet werden. Trotzdem bringt diese Vorgehensweise eine sehr große Zeitersparnis mit sich.

6 Verwandte Literatur

In diesem Bereich sollen einige andere Arbeiten zum Thema Testen von Web Applikationen betrachtet werden.

6.1 Analysis and Testing of Web Applications

Diese Arbeit[2] stammt von Filippo Ricca and Paolo Tonella vom Centro per la Ricerca Scientiica e Tecnologica in Trento, Italien. Die beiden versuchen mittels eines modellbasierten Ansatzes automatisches Testen von Web Applikationen möglich zu machen. Dabei gehen sie wie folgt vor:

Zu allererst erstellen sie ein theoretisches Modell für die Benutzerschnittstelle der Web Applikation. Dabei werden die Websites, die einem Benutzer gezeigt werden in verschiedene Module zerlegt, wie zum Beispiel Forms, Links, statische Pages, Frames und ähnliches. Danach werden diese Module miteinander in Beziehung gesetzt, so wie es die Funktionalität der WebApplikation vorsieht. Daraus ergibt sich dann ein graphentheoretisches Modell.

Die einzelnen Testfälle werden dann anhand der Pfade in diesem Graphen ermittelt und diese dienen dann auch als Grundlage für alle weiteren Tests.

Zum Erstellen dieser Modelle und zum Testen setzen die beiden Authoren zwei Tools ein: ReWeb und TestWeb.

ReWeb ist ein graphisches Werkzeug mit dem Websites analysiert werden können und das ein Modell, so wie oben beschrieben, erstellt. Dieses Modell ist der Input für TestWeb, das anhand der Pfade in dem Modell die Testfälle erstellt und auch durchführen kann.

Zur Überprüfung haben die beiden Authoren dieser Arbeit ihrere Theorien an zwei öffentlichen Websites ausprobiert und kamen dabei zu folgendem Ergebnis: Vor allem ReWeb eignet sich hervorragend zur Analyse von Web Applikationen, da es die Zusammenhänge der einzelnen Teile sehr anschaulich darstellt. Auch TestWeb liefert bereits recht gute Ergebnisse, jedoch ist hier der manuelle Zusatzaufwand, der nötig ist um die Tests laufen zu lassen noch sehr groß.

6.2 Improving Web Application Testing with User Session Data

Die Arbeit *Improving Web Application Testing with User Session Data[1]* stammt von Sebastian Elbaumt, Srikanth Karret, Gregg Rothermel vom Department of Computer Science and Engineering der University of Nebraska - Lincoln und dem Department of Computer Science der Oregon State University in Corvallis, Oregon. Die Authoren untersuchen wie weit sich Whitebox

Testfälle durch das hinzufügen von aufgezeichneten User Session Daten verbessern lassen. Dazu gingen sie wie folgt vor:

Zuerst erstellten sie Testsfälle basieren auf der oben beschriebenen Arbeit von Ricca und Tonella. Danach versuchten sie einen grundlegend anderen Ansatz. Sie speicherten einfach sämtlichen Requests und User Session Daten ab. Aus den gesammtelten Daten generierten sie Testfälle und konnten so wiederum Tests durchführen. Als dritte Überprüfung versuchen sie die Testfälle die mit der ersten Methode erstellt wurden mit den Daten, die sie in mit der zeiten Methode gesammelt hatten zu verbessern.

Alle diese Ansätze probierten sie an praktischen Beispielen und verglichen dann die Ergebnisse. Es zeigte sich folgendes: Keines der drei Verfahren ist den anderen weit überlegen, jedoch ist klar ersichtlich, dass die Abdeckung der Web Applikation durch Testfälle bei der dritten Methode am größten ist, weil die beiden ersten Methoden unterschiedliche Teile der Webapplikation abdecken.

6.3 Automated Replay and Failure Detection for Web Applications

Die Arbeit *Automated Replay and Failure Detection for Web Applications[3]* von Sara Sprenkle, Emily Gibson, Sreedevi Sampath und Lori Pollock von der Abteilung Computer and Information Sciences der University of Delaware beschäftigt sich mit einem ähnlichen Ansatz wie die Arbeit von Elbaumt, Karrent und Rothermel. Auch sie versuchen aus aufgezeichnetem Benutzerverhalten Tests möglich zu machen.

Die Authorinnen dieser Arbeit gingen aber noch einen Schritt weiter. Sie implementieren gleich ein ganzes Framework mit dem solche automatisierten Tests möglich werden sollen. Diese Framework kann als Zwischenschicht, zwischen Web Applikation und Benutzer gesehen werden. Hier wird jede Aktion des Benutzer aufgezeichnet und in Testfälle umgewandelt. Diese werden in einer zentralen Datenbank abgelegt, wobei Rücksicht genommen wird, dass nur unterschiedliche Testfälle in der Datenbank verzeichnet werden.

Die auf den aufgezeichneten Useraktionen basierenden Testfällen können dann wieder an die Webapplikation gesendet werden. Die Ergebnisse werden erneut analysiert und mit erwarteten Anfragen verglichen. Basierend auf solchen Ergebnissen kann dann ein Reporting passieren.

Basierend auf ihren eingenen Tests empfehlen die Authorinnen, dass Anwender ihres Frameworks die Test sowohl stateless, als auch stateful ablaufen lassen, da sich so die größte Abdeckung mit Testfällen ergibt. Außerdem solle ein etwaiger Anwender großes Augenmerk auch die Analyse und den Ver-

gleich mit den Erwartenten Ergebnissen richten um zu viele falsche Positive zu vermeiden.

6.4 Testing Web Applications focusing on their Specialties

Die Arbeit *Testing Web Applications focusing on their Specialties[4]* stammt von Lei Xu, Baowen Xu und Jixiang Jiang vom Department of Computer Science and Engineering der Southeast University in Nanjing, China. In ihrer Paper arbeiten sie heraus, warum eine Web Applikation sehr spezielle Software ist und wie sich das auf das Testen dieser auswirkt.

Dabei beschrieben sie einen einfaches Vorgehensmodell wie Testfälle für Web Applikationen erstellt werden können. Ihre herangehensweise ist sehr pragmatisch und einfach: reverse engineering. Die Basis dieses reverse engineerings ist der HTML Code und die Session Information, die der Benutzer eine Web Applikation zu sehen bekommt und außerdem die Erfahrung, die er beim Benutzen der Webapplikation sammelt. Ihre Grundidee umfasst dabei folgende Schritte:

1. Analyse der Web Applikation und Aufbau einse Modells
2. Definieren von Anforderungen für die folgenden Tests
3. Aus den Anforderungen und dem Modell Anhand der Pfade in dem Modell Testfälle generieren
4. Die Anzahl der Testfälle so weit wie möglich minimieren.

Dieses Vorgehensmodell beschränken die Authoren keineswegs ausschließlich auf funktionale Tests. Auch für Performance Tests, Usability Tests oder Regressions Tests dient diese Art der Testfallerstellung als Basis.

Um diese Art des Testens und der Testfallerstellung möglich zu machen entwerfen sie ein abstraktes Framework, das sehr stark an jenes von Sprecle, Gibson et al. erinnert.

7 Fazit

In dieser Arbeit wurde deutlich, dass es möglich ist Web Applikationen strukturiert und automatisiert zu testen. Keines der untersuchten open source Werkzeuge bietet zwar eine generelle Lösung an, die sich auf alle Web Applikationen anwenden ließe. Dies ist vermutlich auch garnicht möglich, da jedes Web Applikation Framework gewisse Eigenheiten hat (vgl. Cocoon Contunations) auf die speziell Rücksicht genommen werden muss. Trotzdem bieten die untersuchten Vertreter durchaus die Möglichkeit sinnvolle Tests durchzuführen. Besonders aufgefallen ist das Tool JMeter, da es sehr flexibel ist und sich auch für das Cocoon Framework anpassen ließ. Vor allem der Ausblick mit JMeter ohne Aufwand aus den funktionalen Tests Lasttests machen zu können macht JMeter - auch als funktionales Testtool - sehr interessant.

Literatur

[1] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[3] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[4] Lei Xu, Baowen Xu, and Jixiang Jiang. Testing web applications focusing on their specialties. *SIGSOFT Softw. Eng. Notes*, 30(1):10, 2005.