Masterarbeit

Automatisierte Testverfahren für web-basierte Anwendungen

Verfasser
Oliver Fischer
Studiengang eBusiness, Matrikelnr. 2307197

Erstgutachterin
Frau Prof. Dr.-Ing. Monika Heiner
Lehrstuhlinhaberin DSSZ (BTU Cottbus)

Zweitgutachter
Herr Dr.-Ing. Steffen Jurk
Technischer Geschäftsführer der epion GmbH

Eingereicht am 06.10.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANT	Another Neat Tool
ASP.NET	Active Server Pages .NET
AUP	Agile Unified Process
CMMI	Capability Maturity Model Integration
COM	Component Object Model
CSV	Comma Separated Values
CUT	Class Under Test
DOM	Document Object Model
DSDM	Dynamic System Development Method
FDD	Feature Driven Development
FF	Firefox
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikationsnummer
IDE	Integrated Development Environment
IE	Internet Explorer
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JDK	Java Development Kit
JS	JavaScript
JSP	Java Server Pages
KW	Kalenderwoche
MS	Meilenstein
OLE	Object Linking and Embedding
PDA	Personal Digital Assistant
PHP	PHP Hypertext Preprocessor
PSP	Personal Software Process
PUT	Program Under Test
RC	Remote Control
RUP	Rational Unified Process
SPPED	Snoopy Petrinetz Editor
SUT	System Under Test
SVG	Scalable Vector Graphics
TCL	Tool Command Language
TF	Testfall
TQM	Total Quality Management
TSP	Team Software Process
Üdt	Überdeckungstest
UI	User Interface
URL	Uniform Resource Locator
VBS	Visual Basic Script
WWW	World Wide Web
XML	eXtensible Markup Language
XP	Extreme Programming
XSD	eXtensible Schema Definition
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformations
XT	Extreme Tailoring

1 Einleitung

Es gibt keine fehlerfreie Software, wie uns die Praxis immer wieder zeigt. Beinahe täglich berichten Fachzeitschriften und Webportale über Fehlverhalten und Sicherheitslücken großer Softwaresysteme. Es existiert also immer ein gewisses Fehlerpotenzial, das mit der steigenden Komplexität des zu errichtenden Systems wächst. Die Konsequenzen sind weitreichend, wie die Vergangenheit uns lehrte. So war bspw. im September 1983 ein Softwarefehler die Ursache dafür, dass das russische Frühwarnsystem fälschlicherweise den Abschuss von fünf nuklearen Raketen meldete [Long07]. Da zu dieser Zeit die internationale Lage ziemlich angespannt war (es herrschte der Kalte Krieg zwischen den USA und der Sowjetunion), hätte diese fehlerhafte Programmausgabe durchaus auch gravierende Folgen haben können.

Sicherlich ist dies ein extremes Beispiel und die Konsequenzen von Softwarefehlern bedrohen nur selten das Wohl unserer Gesellschaft. Doch sind diese sowohl für den Hersteller, als auch für den Endnutzer stets mit Kosten verbunden. So musste die Fluggesellschaft British Airways in den ersten fünf Tagen nach Eröffnung ihres neuen Terminals am Flughafen Heathrow einen Verlust von ca. 16 Millionen Pfund verzeichnen. Ein einfacher Programmfilter, der nach dem Testprozess nicht entfernt wurde, blockierte das Gepäcksystem und zwang die Passagiere zu Alternativrouten [BA08]. Dieses Beispiel zeigt zudem, dass selbst der Testprozess fehlerbehaftet sein kann und ebenfalls getestet werden sollte.

Gegenstand dieser Arbeit sind jedoch weder sicherheitskritische Frühwarnsysteme, noch hochsensible Transportmechanismen, sondern Webanwendungen. Diese sind in der Regel auf Webserver abgelegt, welche über ein Netzwerk (bspw. das Internet) mit dem Rechner des Nutzers verbunden sind. Der Nutzer benötigt zur Bedienung der Software lediglich einen Webbrowser, der bei den gängigsten Betriebssystemen bereits vorinstalliert ist. Typische Webanwendungen sind z.B. Onlineshops, Suchmaschinen, Social Networks und Onlinebanking-Systeme, die in unserem Alltag immer mehr an Bedeutung gewinnen und gerade deshalb eine hohe Softwarequalität ausweisen sollten.

Durch das reine Testen kann man die Fehlerfreiheit eines Programms allerdings nicht nachweisen, sondern lediglich die Präsenz der Fehler aufzeigen. Das bedeutet, je mehr und je effektiver wir testen, desto größer ist die Wahrscheinlichkeit fehlerärmere Software entwickeln zu können. Nun ist das Testen von Software nicht gerade die attrak-

tivste Tätigkeit eines Entwicklers und schon gar nicht, wenn es sich dabei um das eigene Erzeugnis handelt. Daher stellt sich die Frage, ob man das natürliche, menschliche Fehlerpotenzial durch verstärkte Nutzung von Testwerkzeugen einschränken und damit im Endeffekt die Qualität des Produktes steigern kann? Diese werden allerdings immer noch von Menschen bedient und können demnach nur so effizient eingesetzt werden, wie ihr Benutzer es zulässt. Auch den Kostenfaktor sollte man dabei nicht außer Acht lassen. Denn die Kosten für die Anschaffung und Einrichtung des Werkzeugs sowie die Schulung der Mitarbeiter können den Nutzen um ein Vielfaches übersteigen. Jedoch wie sollte man sonst solche komplexen Programme ausreichend testen, wenn nicht mit Unterstützung von automatisierbaren Testwerkzeugen? Wie lange würde also ein Team benötigen, um bspw. 90% aller Funktionalitäten der Onlineauktionsplattform eBay manuell (d.h. ohne Unterstützung von Testwerkzeugen) zu testen? Gerade wenn man bedenkt, dass das System alle zwei Wochen um mehr als 100.000 Codezeilen erweitert wird [Cone06]. Diesen und weiteren Fragen versuche ich in der vorliegenden Arbeit nachzugehen.

2 Aufgabenstellung

In diesem Kapitel werden die Ziele, der Aufbau, die Realisierung und die Betreuung der Arbeit thematisiert.

2.1 Zielsetzung

Die vorliegende Arbeit behandelt zwei Schwerpunkte. Der erste Schwerpunkt konzentriert sich auf die Darstellung aktueller, automatisierbarer Testwerkzeuge für Webanwendungen. Dabei sollen Möglichkeiten sowie Grenzen aufgezeigt und wesentliche Alleinstellungsmerkmale herausgearbeitet werden. Insbesondere ist der Fragestellung nachzugehen, inwieweit sich kommerzielle Testwerkzeuge von quelloffenen Werkzeugen unterscheiden. Im zweiten Schwerpunkt sollen die gewonnenen Erkenntnisse an praktischen Beispielen Anwendung finden. Die epion GmbH entwickelte eine Webanwendung zur Antragstellung von Jugendleiter/Innen-Cards, die das bisherige Papierverfahren ablöste[1]. Anhand dieses Projekts soll eine Aufwandsanalyse durchgeführt werden, mit der die wirtschaftlichen Aspekte der Testautomatisierung näher betrachtet werden können. Bei der zweiten zu betrachtenden Webanwendung handelt es sich um einen grafischen Petrinetzsimulator, der vom Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit (BTU Cottbus) entwickelt wurde[2]. Dieser ermöglicht es, die Interaktion der Netzelemente animiert darzustellen. Anhand einer Machbarkeitsstudie soll geklärt werden, welche Voraussetzungen geschaffen werden müssen, damit ein automatisiertes Testsystem zum Einsatz kommen kann.

2.2 Aufbau der Arbeit

Nach einer Einleitung und der Aufgabenstellung, werden in Kapitel 3 zunächst die Grundlagen des zu bearbeitenden Themas behandelt. Dabei wird geklärt, wo sich das Testen im Softwareentwicklungsprozess eingliedert. Weiterhin wird eine Klassifizierung der bekanntesten Testmethoden vorgenommen und Aspekte der Testautomatisierung aufgegriffen. Eine Übersicht der zahlreichen Webtechnologien schließt dieses Kapitel ab. Das Kapitel 4 beschäftigt sich ausschließlich mit dem ersten Schwerpunkt der Arbeit. In einem Vergleich werden fünf kommerzielle und fünf quelloffene Testwerkzeuge betrachtet und wesentliche Unterschiede herausgearbeitet. Kapitel 5 stellt die Ergebnisse der zwei bearbeiteten Fallstudien dar und wertet diese aus. Das letzte Kapitel

[1] URL: http://www.juleica.de/224.0.html
[2] URL: http://www-dssz.informatik.tu-cottbus.de/web_animation/pn_demos_flat-nets.html

fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick für weitere Untersuchungen.

2.3 Projektplanung

Im Studium der Fachrichtung eBusiness werden sowohl Grundkenntnisse der Informationstechnik, als auch Grundlagen der Betriebswirtschaft vermittelt. Die Projektplanung spielt dabei stets eine wesentliche Rolle. Deshalb wird in diesem Abschnitt auf die Inhalte der geplanten Meilensteine eingegangen und rückblickend auf deren Realisierung Bezug genommen.

Tabelle 1 stellt die Inhalte der einzelnen Meilensteine (im Folgenden mit MS abgekürzt) dar und grenzt diese zeitlich voneinander ab. Für diese Arbeit standen insgesamt sechs Monate Arbeitszeit zur Verfügung, die in 26 Kalenderwochen (im Folgenden mit KW abgekürzt) aufgeteilt wurden. Die geplanten Tätigkeiten der einzelnen Meilensteine sind als Schwerpunktbereiche aufzufassen, weil parallel dazu des Öfteren Tätigkeiten durchgeführt worden sind, die einem Themenbereich der Arbeit nicht unmittelbar zugeordnet werden konnten, wie z.B. Gliederungsumstrukturierungen oder Design-Arbeiten am Dokument. Das Ziel des ersten MS war es, die Inhalte und nötigen Voraussetzungen der Themenschwerpunkte zu ermitteln und stichpunktartig zu notieren. Im MS 2 beschäftigte ich mich hauptsächlich mit dem Vergleich aktueller Testwerkzeuge. Dieser war von äußeren, organisatorischen Einflüssen weitgehend unabhängig und konnte deshalb frühzeitig fertiggestellt werden. Weiterhin konnten, durch die regelmäßige Anwesenheit bei der epion GmbH, umfangreiche Projektinformationen bezüglich der Juleica-Webanwendung gesammelt werden. Während in MS 3 der Fokus auf die Bearbeitung der Fallstudien gelegt wurde, befasste ich mich in MS 4 mit der Umformulierung der notierten Stichpunkte in leserfreundliche Texte und ansprechende Grafiken. Den Abschluss bildet der MS 5, in dem hauptsächlich kosmetische Tätigkeiten vorgenommen worden sind.

Rückwirkend betrachtet kann man sagen, dass der Meilensteinplan zusammen mit dem Projekt gereift ist, da anfangs nur grobe Vorstellungen verarbeitet werden konnten, die sich im Verlauf der Arbeit festigten. Weiterhin ergab sich während der Bearbeitungszeit ein Arbeitsausfall von etwa zwei Wochen, der aus beruflichen Bewerbungstätigkeiten resultierte. In zukünftigen Projekten sollte daher auf eine etwas defensivere Kalkulierung und auf die Schaffung von kleinen Pufferzeiträumen Wert gelegt werden, um auf äußere Einflüsse flexibler reagieren zu können. Trotzdem konnte man dem auch etwas

Gutes abgewinnen. Denn dadurch besaß man permanent das Gefühl mit der Arbeit nicht rechtzeitig fertig zu werden, was zur Steigerung der Selbstdisziplin beitrug. Des Weiteren sind die teilweise sehr monotonen Formulierungstätigkeiten in Zukunft zeitlich weiter zu streuen, um mehr Abwechslung zu erreichen und die Motivation somit über einen längeren Zeitraum aufrecht zu erhalten. Positiv ist das frühzeitige „Abtasten“ der Schwerpunktbereiche zu bewerten, wodurch Schwachstellen schnell aufgedeckt und Alternativen erarbeitet werden konnten. Alles in allem ist hierbei ein sehr straffer Zeitplan entstanden, der weitgehend eingehalten werden konnte.

Tabelle 1: Meilensteinplan dieser Arbeit

Meilen-stein	Kalender-woche	Geplante Tätigkeiten
MS 1	KW 15-21 (7 Wochen)	• Allgemein: Einleitungsblöcke aller Abschnitte verfassen • Kap. 1 – Einleitung: Inhalt (stichpunktartig) • Kap. 2 – Aufgabenstellung: Inhalt (stichpunktartig) • Kap. 3 – Stand der Technik: Inhalt (stichpunktartig) • Kap. 4 – Testwerkzeuge: Ermittlung der Kandidaten für den Vergleichstest • Kap. 5.1 – Juleica: Einarbeitung in das Testsystem von epion
MS 2	KW 22-24 (3 Wochen)	• Kap. 4.3 – Vergleich: Vergleich der ausgewählten Testwerkzeuge • Kap. 5.1 – Juleica: Projektinformationen (stichpunktartig)
MS 3	KW 25-31 (7 Wochen)	• Kap. 1 – Einleitung: fertigstellen • Kap. 4 – Testwerkzeuge: Detailbetrachtung der zwei ausgewählten Testwerkzeuge, Kapitel fertigstellen • Kap. 5.1 – Juleica: Testfälle und Kostenübersicht erstellen • Kap. 5.2 – Patty: Projektinformationen, Modellierung eines Testsystems, Auswertung der Ergebnisse (stichpunktartig)
MS 4	KW 32-38 (7 Wochen)	• Kap. 2 – Aufgabenstellung: fertigstellen • Kap. 3 – Stand der Technik: fertigstellen • Kap. 5.1 – Juleica: Auswertung der Ergebnisse, Kapitel fertigstellen • Kap. 5.2 – Patty: fertigstellen • Kap. 6 – Zusammenfassung: fertigstellen
MS 5	KW 39-40 (2 Wochen)	• Allgemein: Abschließende Tätigkeiten

2.4 Betreuung

Für die Betreuung der vorliegenden Arbeit wurde ich zum Einen von dem technischen Geschäftsführer der epion GmbH Dr.-Ing Steffen Jurk und zum Anderen von der Lehr-

stuhlinhaberin für Datenstrukturen und Softwarezuverlässigkeit (BTU Cottbus) Frau Prof. Dr.-Ing Monika Heiner tatkräftig unterstützt.

Die epion GmbH wurde mit der Umsetzung des Online-Antragsverfahrens für Jugendleiter/In-Cards beauftragt und konnte sich in einem bundesweiten Ausschreiben gegen 80 Mitbewerber behaupten. Das Unternehmen wurde im Jahr 2000 gegründet, beschäftigt derzeit zehn Mitarbeiter und ist in Cottbus ansässig. Die epion GmbH ist ein Softwaresystemhaus mit den Schwerpunkten Beratung, Projektierung, Konzeption und Entwicklung von webbasierten Anwendungen, Datenbankentwicklung und Serverbetrieb. Zur Anerkennung ihrer Leistungen wurde epion Anfang 2009 der Roland-Berger-Gründerpreis verliehen [LR].

Der Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit ist u.a. auf dem Gebiet der Analyse und Modellierung biochemischer Netzwerke tätig. Mit ihrer Snoopy-Software haben sie ein Werkzeug entwickelt, mit dem Graph-basierte Systembeschreibungen, insbesondere Petrinetze, modelliert und simuliert werden können. Die Patty-Software ermöglicht die web-basierte Simulation solcher Petrinetze durch den Webbrowser. Im Lehrbetrieb der BTU Cottbus werden den Studierenden Möglichkeiten und Grenzen von Testverfahren aufgezeigt sowie Methoden zum Modellieren bzw. Analysieren nebenläufiger Systeme mit Hilfe von Petrinetzen vermittelt.

3 Stand der Technik

Dieses Kapitel soll als Einführung in die verschiedenen Kernthemen dieser Arbeit dienen. Dabei wurde versucht die umfangreichen Themenblöcke auf ein wesentliches Maß zu reduzieren, sauber zu strukturieren und trotzdem Zusammenhänge deutlich darzustellen. Die ersten beiden Abschnitte beschäftigen sich mit der Erstellung und dem Test von allgemeinen Softwareprodukten. Dabei wird anhand verschiedener Prozessmodelle aufgezeigt, wo der Testprozess innerhalb der Softwareentwicklung einzuordnen ist und wie dieser durchgeführt wird. In den darauf folgenden Kapiteln wird das Hauptaugenmerk auf das automatisierte Testen von Webanwendungen gelegt. Nach einer Darstellung spezieller Automatisierungsmöglichkeiten und deren Voraussetzungen, erfolgt eine Betrachtung der Webtechnologien, die zur Erstellung aktueller Webanwendungen verwendet werden.

3.1 Prozessmodelle

Die aktuellen Prozess- und Qualitätsmodelle lassen sich in drei Klassen gliedern, nämlich die Referenzmodelle, die monumentalen Modelle und die agilen Modelle. Während bei den Referenzmodellen in der Regel nur die Ziele der Optimierung vorgegeben werden, die Umsetzung aber völlig offen lassen, beschreiben monumentalen Modelle beide Aspekte sehr ausführlich und bis ins kleinste Detail. Agile Modelle sind dagegen eher leichtgewichtig und konzentrieren sich auf wenige Prozesse und deren Realisierung. Spezielle Referenzmodelle sind bspw. das CMMI-Modell (Capability Maturity Model Integration), das ISO 9000-Modell und das TQM-Modell (Total Quality Management), werden jedoch hier nicht näher betrachtet. Vertreter monumentaler Modelle sind das RUP-Modell (Rational Unified Process), das PSP- bzw. TSP-Modell (Personal bzw. Team Software Process) und das V-Modell in all seinen Versionen [Balzert08: S.516ff]. Letzteres wird im nächsten Abschnitt genauer behandelt. Als Kontrast dazu wird ebenfalls das Konzept des Extreme Programming, als bekanntester Vertreter der agilen Modelle und Anwender der testgetriebenen Entwicklung, etwas genauer betrachtet. Weitere Modelle und Methoden dieser Art sind ergänzungsweise die Dynamic System Development Method (DSDM), das AUP-Modell (Agile Unified Process) und das FDD-Konzept (Feature Driven Development). Zum Abschluss dieses Kapitels erfolgt ein Vergleich der agilen und monumentalen Modelle, um die Grundzüge sowie Vor- und Nachteile noch einmal explizit darzustellen.

3.1.1 V-Modell

Das V-Modell wurde im August 1992 erstmals der Öffentlichkeit vorgestellt und erhielt mit dem V-Modell 97 und dem V-Modell XT später zwei bedeutende Erweiterungen [IABG]. Die Grundzüge des Konzepts sind vom Wasserfallmodell übernommen worden, welches als eines der ältesten Prozessmodelle bereits in den 60er-Jahren des vergangenen Jahrhunderts eingesetzt worden ist [Thaller02: S.21].

Das Wasserfallmodell

Das Wasserfallmodell erhält seinen Namen aufgrund seiner stufenartigen Prozessstruktur, die in Abbildung 1 dargestellt ist. Die Abfolge der Prozessschritte von der Analyse, über den Entwurf, der Implementierung, dem Test, bis zum Betrieb der Software, ist streng sequenziell. Jede Phase muss in der richtigen Reihenfolge und vollständig durchlaufen werden. Die in einer Phase entwickelten Softwaredokumente stellen die Ergebnisse dar und sind als bindende Vorgabe der nächsten Phase anzusehen. Um Korrekturen an diesen Ergebnissen vornehmen zu können, sind Rückschritte zur jeweils vorherigen Phase möglich.

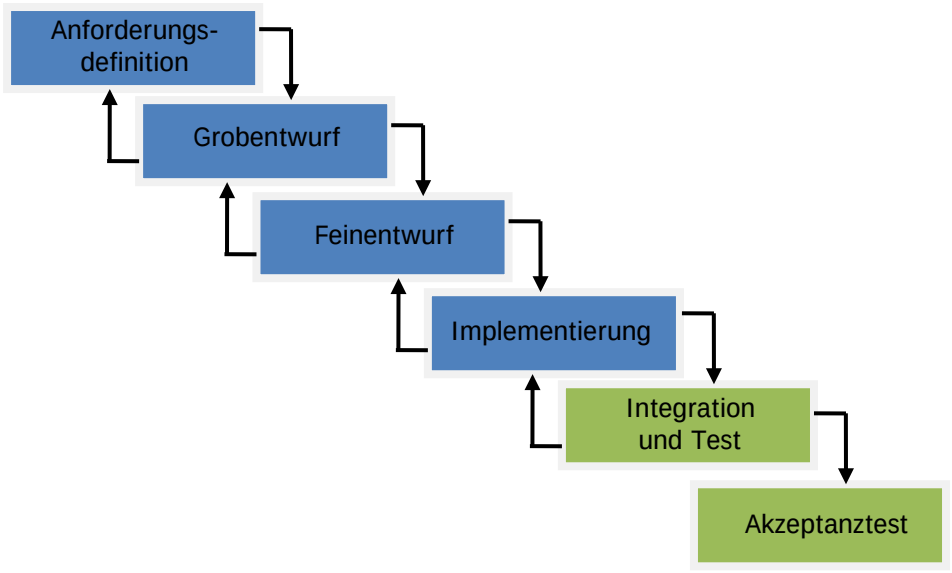


Abbildung 1: Das Wasserfallmodell, vgl. [Thaller02: S.22]

Mit dem Wasserfallmodell werden insbesondere zwei Ziele verfolgt. Zum Einen wird die Verifizierung der Softwaredokumente bereits am Ende einer Phase vorgenommen und nicht erst am Ende des Projekts. Und zum Anderen soll, durch die Gliederung in überschaubare Abschnitte, der Einblick in die Entwicklung, sowohl für den Kunden, als auch für das Management, transparenter gestaltet werden [Thaller02: S.21].

Es ist ein recht einfaches und leicht verständliches Modell, das keinen großen Verwaltungsaufwand besitzt und sich somit für kleine und weniger komplexe Projekte eignet. Es gibt aber auch eine Reihe von Nachteilen, die dazu führten, dass das Wasserfallmodell heutzutage in der Industrie kaum noch Anwendung findet. Z.B. gehen Zeitverzüge in den einzelnen Phasen meistens zu Lasten des Tests und damit zu Lasten des Produkts. Des Weiteren kann der eigentliche Projektfortschritt erst in den späten Phasen (Implementierung und Test) ermittelt werden. Weiterhin müssen die Projektanforderungen, sowie das Budget, die Projektdauer und die zur Verfügung stehenden Ressourcen zu Beginn des Projekts bekannt und vollständig sein, da angenommen wird, dass sich diese Faktoren auch im späteren Verlauf nicht ändern werden. Diese Denkweise macht das Projekt recht unflexibel. Auf Veränderungen muss durch ein effizientes Risikomanagement reagiert werden, was zusätzlichen Verwaltungsaufwand bedeutet. Aus diesem Grund ist das Modell auch kaum für dynamische und komplexe Projekte geeignet [Vigenschow05: S.126ff].

Das V-Modell 97

Den Namen erhält dieses Modell durch die V-förmige Prozessstruktur im Softwareentwicklungsbereich. Neben ihm werden weitere Bereiche für Projektmanagement, Konfigurationsmanagement und Qualitätssicherung definiert. Es ist sehr umfangreich beschrieben und insbesondere für Großprojekte ausgelegt. Im Gegensatz zur Vorgängerversion, ist im V-Modell 97 erstmals die Qualitätssicherung in den Prozessablauf integriert worden (siehe Abbildung 2). Dabei wird zwischen Verifikation und Validation unterschieden. Während bei der Verifikation überprüft wird, ob das Programm so arbeitet, wie es spezifiziert worden ist - *Testen aus Sicht der Entwickler* - wird mit der Validation untersucht, ob das Softwareprodukt für den Einsatzzweck geeignet ist - *Testen aus Sicht der Anwender*. Auf den verschiedenen Ebenen der Definitions- und Entwicklungsstufen werden bereits die Testfälle für die entsprechenden Ebenen der Teststufen formuliert [Vigenschow05: S.129].

Seit Veröffentlichung des V-Modells 97 galt es als Vorgabe für die Durchführung von zivilen und militärischen IT-Projekten des Bundes. In der Industrie wurde es jedoch nicht in dem Maße genutzt, wie es von den Autoren wünschenswert gewesen wäre, was vermutlich auf die Komplexität des Konzepts zurückzuführen ist. Bis 2004 wurde das Modell nicht weiterentwickelt. Zu diesem Zeitpunkt spiegelte es auch nicht mehr den aktuellen Stand der Informationstechnik wider. Neue Methoden und Technologien, wie bspw. die komponentenbasierte Entwicklung oder der Test-First-Ansatz, sind im

97er Modell nur beschränkt möglich. Aus diesem Grund erhielt es eine Überarbeitung und wurde im Februar 2005 unter der Bezeichnung V-Modell XT veröffentlicht [VModellXT: S.7].

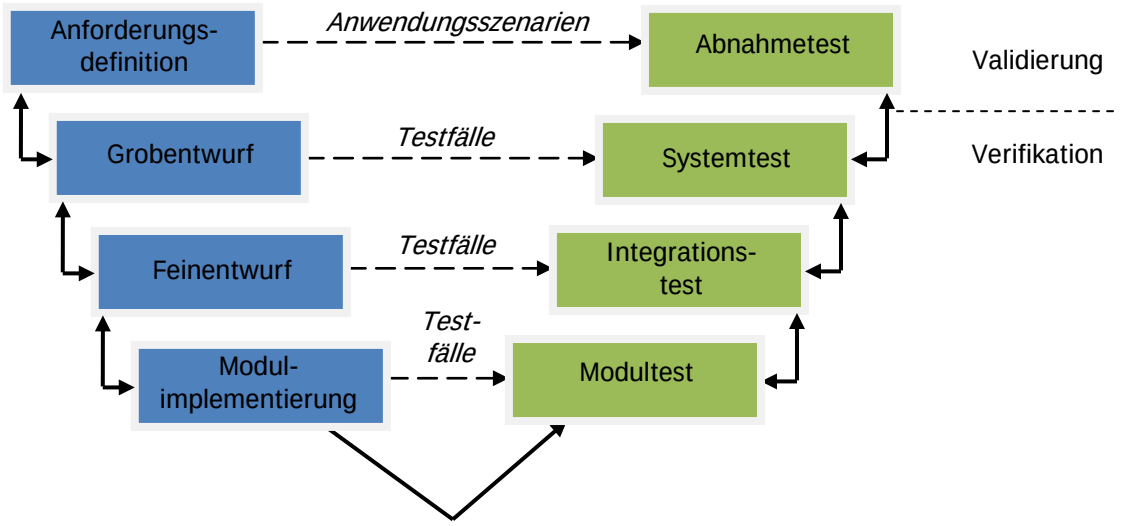


Abbildung 2: Das V-Modell, vgl. [Vigenschow05: S.129]

Das V-Modell XT

Den Forderungen nach leichterer Skalierbarkeit und Anwendbarkeit ist man nachgekommen und die Erfahrungen und Verbesserungsvorschläge wurden im neuen Modell berücksichtigt. Dies führte wiederum zu einer steigenden Akzeptanz in der Industrie und zur Standardisierung für Softwareentwicklungsprojekte der Bundesverwaltung [VModellXT: S.7].

Das V-Modell XT versteht sich als Leitfaden zum systematischen Planen und Durchführen von komplexen und umfangreichen Softwareentwicklungsprojekten. Es wird detailliert festgelegt, welche Aufgaben, von welcher Person (bzw. in welcher Rolle), zu welchem Zeitpunkt bearbeitet werden. Dabei wird die Kooperation zwischen dem Auftragnehmer und dem Auftraggeber ebenfalls genau definiert. Das XT im Namen steht für extreme tailoring und soll andeuten, dass das Modell für verschiedene Projekttypen mit unterschiedlichen Anforderungen angepasst werden kann [Balzert08: S.620]. Dies scheint jedoch, nach Helmut Balzert, nur für umfangreiche Projekte zuzutreffen, da sich der organisatorische Aufwand für Kleinprojekte wirtschaftlich nicht rentiert. Hinzu kommt, dass das Modell extrem dokumentenbasiert ist und dadurch gewisse Entscheidungspunkte, ohne vorherige Abnahme der Dokumente, nicht verlassen werden können. Ein weiterer Aspekt ist der enorme Verwaltungsaufwand, den das Management

zusätzlich zu bewältigen hat, ohne dabei die eigentlichen Aufgaben aus den Augen verlieren zu dürfen. Alles in allem ist das V-Modell XT ein umfassendes, anpassbares Modell für Softwareprojekte mit vielen Dokumentationsvorlagen, welches sich vor allem für umfangreiche und teure Projekte lohnt [Balzert08: S.635ff].

3.1.2 Extreme Programming

Ein differenzierter, fast schon revolutionärer Ansatz wird mit Extreme Programming (XP) verfolgt. Er wurde von Kent Beck (einem Berater und Krisenmanager für Softwareunternehmen) im Jahre 1999 veröffentlicht, als das V-Modell 97 noch damaliger Standard war [Balzert08: S.654]. Nach fünfjährigem Einsatz in der Praxis, wurde der Ansatz aus den gewonnenen Erfahrungen und resultierenden Erkenntnissen erweitert und als XP2 veröffentlicht [Balzert08: S.662]. Im Folgenden beziehe ich mich allerdings auf die erste Version, da ich keine detaillierte Unterscheidung liefern möchte, sondern nur die Grundzüge des Konzepts darstellen möchte.

Im Gegensatz zu monumentalen Prozessmodellen, wie bspw. das V-Modell, ist XP eher als eine Sammlung von Regeln und Management-Techniken zu betrachten, die insbesondere kommunikationsbetonte und teamausgerichtete Vorgehensweisen für kleine bis mittlere Projektgruppen beschreiben. Weiterhin zählt XP zu den agilen Modellen, die auf Änderungen der Projektanforderungen schnell und flexibel reagieren können. Die dabei zusätzlich anfallenden Kosten sollen durch die beschriebenen Methoden ungefähr konstant niedrig gehalten werden, im Gegensatz zur traditionellen Sichtweise mit steigenden Kosten [Vigenschow05: S.136f].

Dieses Vorgehen stellt besondere Ansprüche an das gesamte Team. Sehr kompetente und kommunikativ-starke Mitarbeiter werden dafür vorausgesetzt. Es muss eine vertrauensvolle und offene Zusammenarbeit gewährleistet sein, die Konkurrenzdenken untereinander weitgehend ausschließt. Sie müssen alternativen Lösungsansätzen gegenüber aufgeschlossen sein und eine gewisse Portion Mut aufbringen, diese auch umzusetzen. Durch die kontinuierlich gesammelten Erfahrungen und dem regelmäßigen Feedback des Managements soll eine fortlaufende Verbesserung des Entwicklungsprozesses erfolgen [XP].

Methoden und Techniken

Das Team besteht idealerweise aus zwei bis zwölf Programmierern, dem Management und mindestens einem permanent verfügbaren Ansprechpartner, der die Interessen

der Auftraggeber vertritt. Zur Förderung der Kommunikation arbeiten diese zusammen in einer offenen Arbeitsumgebung (einem gemeinsamen Raum oder einer gemeinsamen Abteilung) und integrieren ein gemeinsames Vokabular, um über das zu entwickelnde System und die Arbeitsweisen effizienter diskutieren zu können. Die Entwicklung erfolgt in kleinen Schritten (den sogenannten Iterationen), im Umfang von ein bis drei Wochen, mit dem Ergebnis eines lauffähigen und getesteten Zwischenprodukts. Anschließend werden in einem Projektmeeting das Ergebnis und die Arbeitsweise der vergangenen Iteration reflektiert und im Problemfall Verbesserungsvorschläge für die folgenden Iterationen erarbeitet. Weiterhin finden täglich sogenannte Stand-up-Meetings[3] statt. Dabei berichtet jedes Teammitglied kurz und knapp, welche Aufgaben derjenige am vorherigen Tag bearbeitet hat und welche Aufgaben er an diesem Tag bearbeiten wird. Probleme werden lediglich erwähnt, jedoch nicht innerhalb des Meetings gelöst. Die Programmierung der Software erfolgt jeweils in Paaren (Pair Programming). Dabei sitzen zwei Entwickler an einem Computer und arbeiten kooperativ an einer Lösung. Während der Eine den Quellcode eintippt (Blick fürs Detail), verfolgt der Zweite den Vorgang am Monitor, ohne dabei das Hauptziel aus den Augen zu verlieren (Blick für das Gesamtkonzept). Dies soll vor allem, durch eine geringere Fehlerrate, zu einer höheren Codequalität und einer steigenden Produktivität führen. Die Rollen werden minütlich getauscht und die Partner stündlich gewechselt. Damit soll das Wissen an alle Entwickler gleichermaßen verteilt werden, damit bei Ausfall einer Person der gesamte Betrieb nicht zum Stillstand kommt. Es gibt keine Spezialgebiete bestimmter Personen, an denen nur diese arbeiten. Der Programmcode gehört allen und jeder kann (und sollte bei Bedarf) diesen verbessern. Die eigentliche Entwicklung erfolgt testgetrieben, worauf ich im nächsten Abschnitt noch einmal genauer eingehen möchte. Das Management hat im Speziellen die Aufgabe, dem Team den Fortschritt zu präsentieren und Probleme frühzeitig aufzuzeigen. Die zur Problemlösung nötigen Entscheidungen werden jedoch von den Entwicklern und den Kunden getroffen. Überstunden sollten generell vermieden werden, um die Kreativität und die Konzentration der Entwickler über einen langen Zeitraum hin aufrecht zu erhalten. Der Kunde, bzw. der Ansprechpartner auf Kundenseite, spielt eine zentrale Rolle in diesem Modell. Er befindet sich die meiste Projektzeit über vor Ort und steht den Entwicklern dabei als direkter Ansprechpartner zur Verfügung. Er definiert die Anforderungen für das Softwareprodukt und hält diese auf Karteikarten fest. Wird diese Anforderung durch das

[3] Das unbehagliche Stehen vor der Gruppe ist beabsichtigt und soll die Teilnehmer dazu veranlassen, sich kurz zu fassen und nur das Wesentliche zu berichten.

Entwicklerteam bearbeitet, so steht der Ansprechpartner dem Team für Verfeinerungen oder Verhandlungen des Konzepts sofort zur Verfügung. Weiterhin erstellt dieser die Akzeptanztests, die für Abnahme des Produkts relevant sind. Zu Beginn einer Iteration findet stets ein Planungsmeeting statt. Dabei werden die Arbeitsaufwände für die auf den Karteikarten befindlichen Anforderungen geschätzt und der Umfang der nächsten Iteration ermittelt. Hierbei kann man besonders erkennen, wie viel Einfluss die Kunden auf den Projektablauf haben. Nach ein bis drei Monaten wird ein Release angefertigt und dem Kunden als Feedback der bisherigen Arbeit übergeben [XP].

Testen in XP

In XP werden zwei Arten von Tests unterschieden. Zum Einen handelt es sich um die Unittests, die den Entwicklern zur Verifikation ihrer Teilprogramme dienen, und zum Anderen um die Akzeptanztests, die zur Validierung des Softwareprodukts benötigt werden und dem Management als Maß für den Fortschritt des Gesamtprojekts dienen. Beide Tests sollten stets automatisiert durchgeführt werden und müssen zu jedem Zeitpunkt der Integration zu 100% erfüllt sein. Während die Akzeptanztests zusammen vom Management und den Kunden erstellt werden, ist die Formulierung der Unittests für die Programmierer als Teil der testgetriebenen Entwicklung (Test Driven Development) fest verankert [Link05: S.9f].

Diese Programmiertechnik ist darauf ausgelegt, das Design der Software so einfach wie möglich zu gestalten und es damit lesbar, testbar und (kosteneffizient) modifizierbar zu halten. Das Design wird nicht vorab bis ins kleinste Detail spezifiziert, sondern unterliegt während der Programmierung ständigen Änderungen (Refactoring), um es so einfach wie möglich zu gestalten. Die Programmierung selbst erfolgt in sehr kleinen Schritten und folgt einem dreiteiligen Zyklus. Zuerst wird immer der Testfall komplett erstellt und anschließend gestartet. Der Test muss fehlschlagen, da noch keine entsprechende Logik implementiert worden ist, auf die getestet werden kann. Man möchte damit aber absichern, dass der Test nicht schon vorher, durch möglicherweise auftretende Nebeneffekte, erfüllt wird. Als nächstes erfolgt dann die eigentliche Implementierung. Dabei ist darauf zu achten, dass man die gestellten Anforderungen so einfach wie möglich erfüllt (Vermeidung von komplexen und fehleranfälligen Funktionen) und darauf verzichtet, den Code für spätere Erweiterungen vorbeugend zu optimieren. Erstens ist nicht gewiss, ob diese Erweiterungen später wirklich gefordert wird und zweitens wird der Code in den meisten Fällen sowieso wieder umstrukturiert. Nachdem die Programmlogik nun implementiert und durch die Tests abgesichert worden ist, beginnt

die Phase des Refactorings. Dabei wird die Struktur des Programms wieder in die einfache und lesbare Form gebracht, ohne dabei die bestehende Funktionalität zu verändern. Mit Hilfe der zuvor entwickelten Unittests können die Änderungen auch sofort getestet werden. Dieser Zyklus führt sich für die gesamte Programmierung fort [Westphal06: S.2f]. Durch eine Analogie zum Klettersport wird der Fortschritt des testgetriebenen Vorgehens noch einmal genau veranschaulicht.

„Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken. […] Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. […] Der bis zum Haken erkletterte Weg gehört ihm in jedem Fall, selbst wenn ihm ein Fehler unterläuft.“, vgl. [Westphal06: S.4].

Sämtliche Testfälle werden gesammelt, gepflegt und nach jedem Kompilieren komplett durchlaufen, um festzustellen zu können, ob die neuen Programmzeilen Auswirkungen auf das gesamte System haben. Wurde eine Aufgabe fertiggestellt und sind die zugehörigen Tests erfolgreich durchlaufen worden, so ist dies der Zeitpunkt für eine komplette Integration des Systems. Dies stellt sicher, dass stets eine lauffähige Version vorhanden ist und sollte deshalb auch mehrmals täglich automatisiert durchgeführt werden. Denn je früher Fehler entdeckt werden, umso leichter sind sie zu beheben. Entstehen auch hierbei keine Probleme, so kann der Code in das Versionsverwaltungssystem eingepflegt werden [Westphal06: S.3].

Zusammenfassung

Extreme Programming glänzt vor allem durch ein gutes Reaktionsvermögen, um sich auf ändernde Anforderungen schnell und flexibel einstellen zu können. Ebenso bemerkenswert ist die konsequente Ausrichtung aller Praktiken auf die eigentliche Wertschöpfung, nämlich die Software. Allerdings kann dieses Vorgehen nur gelingen, wenn eine ganze Reihe von Voraussetzungen (Teamgröße, Persönlichkeitsprofil, Qualifikation) erfüllt sind. Bspw. müssen die Entwickler neben dem Programmieren und dem Testen, ebenso gut auch das Analysieren und das Entwerfen von Software beherrschen können. Nach Kent Beck sind die beschriebenen Methoden so aufeinander abgestimmt, dass es kaum Sinn macht, das System in irgendeiner Form anzupassen [Balzert08: S.659]. Trotzdem lassen sich einige Techniken sehr gut in anderen Projektentypen übernehmen [Vigenschow05: S.137].

3.1.3 Agile vs. monumentale Modelle

Monumentale Modelle werden als schwergewichtig bezeichnet, da sie ihre Prozesse sehr detailliert und formal beschreiben. Neben der eigentlichen Software werden vor allem viele Dokumente und Teilprodukte umfassend geplant und erstellt. Dies führt oft zu einer gut durchdachten und aufeinander abgestimmten Prozesskette. Allerdings erschwert dies nachträgliche Änderungen an bestehenden Systemen und ist mit einem sehr großen Aufwand verbunden, da die Dokumente nach der Änderung wieder konsistent gehalten werden müssen. Weiterhin wächst die Gefahr der Software-Bürokratisierung, wenn die Dokumente wichtiger werden als das eigentliche Produkt. Agile Modelle hingegen werden aufgrund ihrer geringen Anzahl an Prozessen als leichtgewichtig bezeichnet. Es sollen gerade so viele Prozesse vorhanden sein, dass sich der Aufwand lohnt. Im Extremfall ist der Quellcode das einzige Dokument. Durch die wenigen Dokumente und der iterativen Entwicklung, in denen stabile Pläne für kurze Zeiträume erstellt werden, kann flexibel auf sich ändernde Anforderungen reagiert werden. Des Weiteren ist dieses Vorgehen sehr codeorientiert, wodurch der Kunde in kurzen Zeitabständen einsatzfähige Teilprodukte erhält. Bei diesem Modell wird allerdings von einem bestimmten Team- bzw. Mitarbeiterprofil ausgegangen, das nicht immer vorhanden ist. Weiterhin sind keine Festpreisverträge möglich, da diese stabile Anforderungen und einen vorhersagbaren Prozess erfordern. Alles in allem kann man sagen, dass sowohl monumentale als auch agile Modelle ihre Stärken als auch ihre Beschränkungen haben und damit nur für spezifische Projekttypen in bestimmten Situationen geeignet sind [Balzert08: S.681ff].

3.2 Testverfahren

In diesem Abschnitt werden zunächst einzelne Fachbegriffe definiert und näher erläutert, um u.a. eine konsistente Verwendung dieser Begriffe zu gewährleisten. Anschließend wird eine Klassifizierung der bekanntesten Softwaretestverfahren vorgenommen, die sich an der Gliederung von Helmut Balzert [Balzert99: S.509f] orientiert. Diese Testverfahren dienen hauptsächlich der Funktionalitätsoptimierung, doch es gibt auch weitere Testausprägungen, die zur Erhöhung der Produktqualität beitragen. Mit einer Darstellung Letzterer wird dieses Kapitel abgeschlossen.

3.2.1 Softwarequalität

Der ISO/IEC 9126 (International Organization for Standardization/International Electrotechnical Commission) Standard legt Begriffe aus dem Bereich der Softwareentwick-

lung fest und definiert Softwarequalität wie folgt: „Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.“, vgl. [Vigenschow05: S.15]. Das bedeutet vereinfacht: Qualität ist die erbrachte Leistung im Verhältnis zur erwarteten Leistung. Sehr gute Qualität besitzt demnach jenes Produkt, welche die Erwartungen des Kunden maximal erfüllt. Daraus resultieren sechs Hauptqualitätsmerkmale: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit [Vigenschow05: S.15f].

Unter Funktionalität versteht man das Vorhandensein von Funktionen, die festgelegte und vorausgesetzte Erfordernisse erfüllen (Teilmerkmale: Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit). Die Zuverlässigkeit wird als Fähigkeit einer Software beschrieben, die ihr Leistungsniveau unter festgelegten Bedingungen in einem festgelegten Zeitraum halten kann (Teilmerkmale: Reife, Fehlertoleranz, Wiederherstellbarkeit). Unter Benutzbarkeit versteht man die Fähigkeit der Software, für einem Benutzer unter festgelegten Bedingungen verständlich erlernbar, anwendbar und attraktiv zu sein (Teilmerkmale: Bedienbarkeit, Erlernbarkeit, Verständlichkeit). Die Effizienz ist das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen (Teilmerkmale: Zeitverhalten, Ressourcenverbrauch). Die Änderbarkeit beschreibt den Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen können Korrekturen von Fehlern, Anpassungen an veränderte Anforderungen (oder Systemumgebung) oder die Verbesserung der Wartung sein (Teilmerkmale: Analysierbarkeit, Modifizierbarkeit, Stabilität, Prüfbarkeit). Mit der Übertragbarkeit ist die Leichtigkeit gemeint, mit der eine Software von einer Umgebung in eine andere übertragen werden kann. Eine Umgebung kann hier die organisatorische Umgebung, die Hardwareumgebung oder die Softwareumgebung sein (Teilmerkmale: Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit). Es können nicht alle Qualitätsmerkmale gleich gut erfüllt werden. Teilweise können sie sich sogar gegenseitig behindern. Also muss eine individuelle Priorisierung (vom Auftraggeber) für jedes einzelne Projekt vorgenommen werden [Vigenschow05: S.17].

Aus Fehlern lernen

Als Fehler wird jede Abweichung der tatsächlichen Ausprägung eines Qualitätsmerkmals von der Soll-Ausprägung sowie jede Inkonsistenz zwischen Implementierung und Spezifikation definiert [Balzert99: S.502].

Irren ist menschlich, besagt ein altes deutsches Sprichwort. Oft werden Fehler als eigenes Versagen beurteilt und anstatt diese offen und ehrlich zuzugeben, werden sie vertuscht, um sich nicht unnötig vor den Kollegen blamieren zu müssen. Dabei kann man Fehlern viel abgewinnen. Aus Fehlern kann man lernen, um diese in Zukunft zu vermeiden. Weiterhin dienen uns Fehler als Feedback zu früheren Entscheidungen, zur intensiven Auseinandersetzung mit dem zu behandelnden Thema und zum Finden von alternativen Lösungswegen. Eine Grundvoraussetzung dafür ist die Schaffung einer konstruktiven Fehlerkultur innerhalb des Entwicklungsteams. Es soll nicht versucht werden die Schuld abzustreiten oder sogar einem Mitarbeiter anzuhängen, sondern viel wichtiger ist es Lösungen für dieses Problem zu finden. Dies kann allerdings nur in einer sachlichen und ergebnisorientierten Atmosphäre geschehen, in der sich jeder Mitarbeiter sicher fühlt und nicht darauf bedacht ist sich ständig verteidigen zu müssen. Und vor allem muss Ruhe bewahrt werden. Es sollten keine voreiligen Entscheidungen getroffen werden, die das Problem im Endeffekt noch verstärken. Durch Schaffung einer solchen konstruktiven Fehlerkultur, profitiert nicht nur der Einzelne, sondern das gesamte Team, sofern die Fehlerursachen und -wirkungen diskutiert werden. Man lernt mit Fehlern umzugehen und reduziert damit nicht nur unnötige Ängste, sondern fördert auch Mut und Initiative jedes Einzelnen. Denn in Projekten wird es immer Fehler geben, da immer etwas völlig neu oder unbekannt ist (z.B. beim Einsatz neuer Technologien, einen unerfahrenen Team oder einem neuartigen Vorgehen). Andernfalls wäre es per Definition kein Projekt, sondern Teil des Tagesgeschäfts [Vigenschow05: 213ff].

Softwaretest

Testen ist ein experimentelles Verfahren, wobei die stichpunktartige Ausführung eines Testobjekts zu dessen Überprüfung dienen soll. Dazu müssen die Randbedingungen für die Ausführung des Tests festgelegt sein (bspw. eine abgestimmte Hardware-Software-Umgebung). Ziel des Softwaretests ist es festzustellen, ob das Testobjekt die geforderten Eigenschaften erfüllt. Dabei werden im Allgemeinen zwei Herangehensweisen unterschieden. Zum Einen das demonstrative Testen, bei dem das Testobjekt ausgeführt wird, um aufzuzeigen, dass die geforderte Funktionalität realisiert worden ist. Und zum Anderen das destruktive Testen, bei dem ein Testobjekt ganz bewusst so ausgeführt wird, sodass die Wahrscheinlichkeit am höchsten ist einen Fehlverhalten zu entdecken [Vigenschow05: S.20]. Eine genaue Klassifizierung der Testmethoden wird im Abschnitt 3.2.2 vorgenommen.

Das Testobjekt (auch Prüfling oder Testling genannt) kann eine Klasse, ein Programm oder ein System sein und wird über eine Reihe von Testfällen ausgeführt. In der englischen Literatur wird oft vom CUT, PUT oder SUT gesprochen, was als Abkürzung für Class Under Test, Program Under Test oder System Under Test zu verstehen ist. Jeder Testfall besteht immer aus einer Menge von Eingabedaten und einer zugehörigen Menge von erwarteten Ausgabedaten. Andernfalls wäre ein Vergleich des Ist- und Sollverhaltens nicht zu realisieren. Handelt es sich bei dem Testobjekt um eine einzelne Klasse und sind dafür direkte Ein- und Ausgaben nicht möglich, definieren sogenannte Testtreiber einen Testrahmen, der durch ein interaktives Aufrufen der Operationen die Ein- und Ausgabe ermöglicht. Weiterhin bezieht sich jeder Testfall stets auf eine definierte Hardware/Software-Umgebung, bspw. dem Testsystem oder der realen Einsatzumgebung. Somit sind die Ergebnisse auch nicht allgemeingültig, sondern nur im Bezug zur Umgebung zu werten [Balzert99: S.502].

3.2.2 Klassifizierung von Testverfahren

Das Hauptziel von Testverfahren ist es Fehler im Testobjekt aufzudecken. Die populärsten Methoden lassen sich in statische und dynamische Verfahren gliedern. Zu den Statischen gehören Inspektionen, Code Reviews und Walkthroughs. Dabei wird nicht das Programm ausgeführt, sondern der zugehörige Quellcode auf Fehler hin untersucht [Balzert99: S.509f].

Am Beispiel von Code Reviews wird das Prinzip kurz erläutert. Dabei sind stets zwei Entwickler beteiligt. Während der Autor seinem Review-Partner den Quellcode erklärt, versucht dieser die Vorgänge nachzuvollziehen und auf Probleme aufmerksam zu machen, die der Entwickler eventuell nicht beachtet hat. Diese Methode eignet sich auch besonders gut, um Fehlerursachen zu finden, die der Entwickler bisher allein nicht aufdecken konnte. Denn in den meisten Fällen ist der Grund dafür die Betriebsblindheit (starre Sicht auf Details), weshalb das Problem übersehen wird, während eine externe Person eher auf die Zusammenhänge achtet und alles Bisherige in Frage stellt. Analog zum Pair Programming (Methode des Extreme Programming, vgl. Kapitel 3.1.2) eignen sich Code Reviews ebenso gut zum Verteilen von Wissen unter den Teammitgliedern, indem die Review-Partner ständig gewechselt werden [Vigenschow05: S.57f].

Im Gegensatz zu den statischen, wird bei dynamischen Testverfahren das Testobjekt mit Eingabedaten versehen und ausgeführt. Es wird zwischen Strukturtestverfahren (auch Glass Box-Test oder White Box-Test genannt) und funktionalen Testverfahren

(auch Black Box-Test genannt) unterschieden [Balzert99: S.509]. Um ein optimales Testergebnis zu erreichen, benötigen wir beide Verfahren. Mit Strukturtests wird geprüft, ob der Programmcode richtig funktioniert (Verifizierung: wurde richtig programmiert?), während funktionale Tests sicherstellen, dass auch die Anforderungen exakt umgesetzt wurden (Validierung: wurde das Richtige programmiert?). Eine Kombination aus diesen beiden Formen stellen die Gray Box-Tests dar. Diese finden bspw. in Integrationstest Anwendung, bei denen sowohl prinzipielle Realisierungstechniken, als auch Teilabläufe aus der Spezifikation zur Erstellung der Testfälle genutzt werden [Vigenschow05: S.21].

Strukturtestverfahren

Bei Strukturtestverfahren erstellt der Tester seine Testfälle auf Grundlage des Programmquellcodes. Diese Einsicht gibt ihm die Möglichkeit, die internen Abläufe des Programms zu analysieren und diese Informationen in den Testfall einfließen zu lassen. Das Hilfsmittel der Analyse ist dabei der Kontrollflussgraph. Dies ist ein gerichteter Graph, der aus einer Menge von Knoten und gerichteten Kanten besteht. Jeder Kontrollflussgraph hat einen Startknoten und einen Endknoten. Jeder Knoten repräsentiert eine Anweisung des Programms und jede Kante (auch Zweig genannt) zwischen zwei Knoten repräsentiert den Kontrollfluss. Ein Pfad ist als eine Folge von Knoten und Kanten anzusehen, die mit dem Startknoten beginnt und mit einem Endknoten endet [Balzert99: S.506]. Zur Verdeutlichung ist in Abbildung 3 ein Kontrollflussgraph eines kleinen Programms (zur Bestimmung der Teilbarkeit zweier Zahlen) beispielhaft dargestellt. Liegt dem Tester ein solcher Graph vor, kann er davon leicht entsprechende Überdeckungstests (im folgenden Üdt abgekürzt) ableiten. Man unterscheidet vereinfacht betrachtet drei Formen von Üdt. Der Anweisungs-Üdt zielt auf die Ausführung aller im Programm vorhandenen Anweisungen (entspricht den Knoten im Kontrollflussgraphen) ab. Dies ist in der Praxis einfach zu realisieren und stellt sicher, dass keine Anweisung ungetestet bleibt. Doch damit werden die Abhängigkeiten zwischen den Anweisungen noch nicht betrachtet. Dies übernimmt zusätzlich der Zweig-Üdt und ersetzt damit gleichzeitig den Anweisungs-Üdt. Da bspw. im Falle von Endlosschleifen ein Test aller Zweige nicht möglich ist, wird eine maximale Anzahl von Durchläufen definiert, die repräsentativ für einen endlosen Durchlauf verwendet werden kann. Als höchste Form der Überdeckungstests wird die Pfadüberdeckung angesehen. Diese beinhaltet gleichzeitig sowohl Anweisungs- als auch Zweig-Üdt und stellt somit das umfassendste kontrollflussorientierte Testverfahren dar. Leider ist eine komplette Überde-

ckung dieses Verfahrens aufgrund der hohen Pfadanzahl in der Praxis nicht sinnvoll einsetzbar [Balzert99: S.510]. Deshalb nimmt man als Tester eine Priorisierung bestimmter Pfade vor, die einen bestimmten Teilbereich der Anwendung abdecken können.

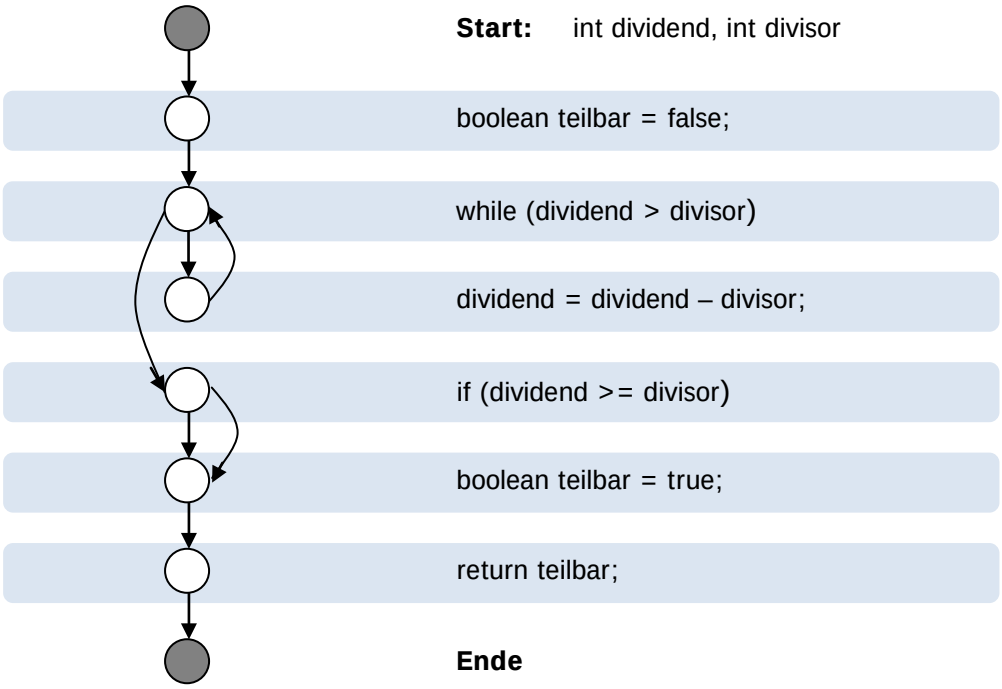


Abbildung 3: Kontrollflussgraph eines einfachen Programms

Funktionale Testverfahren

Die Grundlage zur Erstellung der Testfälle bei funktionalen Testverfahren ist die Spezifikation des Programms. Im Gegensatz zu den Strukturtestverfahren soll der Tester keinen Einblick in die internen Abläufe des Programms erhalten, denn er testet nicht aus Entwicklersicht, sondern aus Anwendersicht. Dabei soll die Prüfung der Funktionalität so umfassend und redundanzarm wie möglich geschehen. Die größte Schwierigkeit liegt dabei in der Definition der Testfälle. Da ein vollständiger Test im Allgemeinen nicht durchführbar ist, sollten die Testfälle so entwickelt werden, dass die Wahrscheinlichkeit am größten ist, auf Fehler zu stoßen. Dazu bedient man sich u.a. der Grenzwertanalyse und der funktionalen Äquivalenzklassenbildung [Balzert99: S.515f].

Um Tests effizient gestalten zu können, benötigen wir Grenzwerte, als Eingabedaten für die vorhandenen Testfälle. Denn Fehler verteilen sich im Allgemeinen nicht gleichmäßig, sondern treten häufig an komplexen Stellen auf, an denen Abläufe implementiert oder Regeln umgesetzt werden. Eine einfache Altersabfrage eines Programms, die

die weitere Fortführung von der Volljährigkeit des Nutzers (Alter >= 18) abhängig macht, dient hierfür als anschauliches Beispiel, was mit Grenzwerten beabsichtigt wird. Bei ganzzahligen Werten sind Grenzwerte die Zahlen, die im Kontext der Anwendung eine andere Eigenschaft annehmen, sobald sich diese Werte um eine Ganzzahl verändern [Vigenschow05: S.60]. Im obigen Beispiel wäre der Grenzwert die Zahl 18 und besitzt damit eine hohe Wahrscheinlich Fehler aufzudecken. Aus programmiertechnischer Sicht sollten ebenfalls die benachbarten Werte des Grenzwertes in den Test mit einbezogen werden. Diese entsprechen im obigen Beispiel den Zahlen 17 und 19. Denn die verschiedenen Vergleichsoperatoren „kleiner“, „größer“, „kleiner-gleich“ und „größer-gleich“ führen bei Programmierern leicht zu Flüchtigkeitsfehlern, die in der Regel nur schwer aufzufinden sind [Thaller02: S.92f]. Handelt es sich bei einem Testobjekt jedoch nicht um Ganzzahlen, werden implizite Reihenfolgen, Abhängigkeiten oder Regeln in Betracht gezogen. Solch einfache Testobjekte, wie in diesem Beispiel, sind jedoch nicht die Regel. Die größte Problematik stellen die Objekte dar, bei denen die entsprechenden Grenzwerte Abhängigkeiten mit anderen Werten besitzen, was zu einer Explosion von Testfällen führt. Äquivalenzklassen versuchen den Aufwand in Grenzen zu halten, damit der Test überschaubar bleibt [Vigenschow05: S.60ff].

Durch das Auswählen möglichst typischer Eingabewerte wird versucht, eine ganze Klasse von Variablen zu testen [Thaller02: S.90]. Die Eingabebereiche sind so zu unterteilen, dass alle Werte innerhalb eines Bereichs gleich gute Testdaten sind, d.h. äquivalent zueinander sind. Die Bildung der Äquivalenzklassen erfolgt heuristisch nach Erfahrung und Intuition. Dabei werden zunächst aus der Spezifikation die Eingabegrößen und die zugehörigen Gültigkeitsbereiche ermittelt. Die Grenzen der Bereiche teilen die Eingabedaten in gültige und ungültige Eingaben. Anschließend wird geprüft, ob in jedem Bereich alle Werte gleich behandelt werden. Falls nicht muss eine weitere Unterteilung vorgenommen werden. Dann werden aus den Bereichen die Eingabedaten für die Testfälle gewählt und erneut geprüft, ob alle diese Daten gleich behandelt werden. Aus ökonomischen Gründen, können mehrere unabhängige Testfälle mit gültigen Eingabedaten zusammengefasst werden. Eine eindeutige Fehlerzuordnung ist dabei trotzdem möglich. Für alle ungültigen Eingabedaten sollte mindestens ein separater Testfall beschrieben werden, da jeweils eine konkrete Fehlersituation getestet wird. Wie effizient und robust Grenzwertanalysen und Äquivalenzklassenbildungen für funktionale Testverfahren sind, ist in erster Linie von deren Testfällen abhängig. Leider ist die zugehörige Ermittlung der entsprechenden Eingabedaten häufig sehr mühsam und manchmal gar nicht möglich [Vigenschow05: S.66f].

3.2.3 Weitere Testausprägungen

Die im vorigen Abschnitt betrachteten Testverfahren dienen insbesondere der Verbesserung des Funktionsumfangs der Software. Man kann aber nebenbei auch noch weitere Wege einschlagen, um die Qualität des Produkts zu erhöhen. So existieren bspw. sogenannte Usability Tests, welche die Handhabbarkeit der Software ermittelt, um dem Endanwender den Umgang mit der Software zu erleichtern. Konfigurationstests sollen klarstellen, ob das entwickelte Produkt ebenfalls auf anderen Computersystemen lauffähig ist und harmonisch mit externen Programmen zusammenarbeitet. Diese Liste ist sicherlich nicht vollständig, doch handelt es sich meiner Meinung nach hierbei um die Bekanntesten.

Volume Test

Jedes Programm hat eine Vielzahl von Grenzen. Das Ziel der Softwareentwicklung ist es u.a., dass diese Grenzen den normalen Betrieb der Software nicht stören. So sollten Endlosschleifen und Programmabstürze nach Möglichkeit abgefangen und der Nutzer gegebenenfalls in Form einer Fehlermeldung informiert werden, damit dieser entsprechend reagieren kann. Im Idealfall sollte der Nutzer seine Arbeit ungehindert fortführen können. Um solche Grenzen ausfindig machen zu können, bedient man sich den Volume Tests. Dabei wird der Programmcode mit einer Masse von Daten konfrontiert, um unter diesen Bedingungen Fehlverhalten der Software hervorzurufen. Selbst wenn der Entwickler sich ziemlich sicher ist und annimmt, dass seine definierten Grenzen niemals erreicht werden können, muss dies nicht den realen Bedingungen der Praxis entsprechen. Ein Beispiel dafür sind die sogenannten Memory Lacks. Falls referenzierter Speicher nach der Verwendung nicht wieder freigegeben wird, würde im Dauerbetrieb der Software irgendwann kein Speicher mehr zur Verfügung stehen und das Programm stürzt ab [Thaller02: S.124f].

Stress Test

Eine dem Volume Test sehr ähnliche Testausprägung ist der Stress Test. Auch hier wird versucht die Leistungsgrenzen zu überschreiten, um das Verhalten der Software in solchen Situationen zu beobachten. Doch im Gegensatz dazu werden bei Stress Tests nur die Belastungsspitzen untersucht, die ein Programm innerhalb kürzester Zeit ausgesetzt sein kann. Um einen sportlichen Vergleich zu ziehen, würde der Volume Test einem Ausdauerlauf entsprechen und ein Stress Test einem 100m-Sprint. Gerade bei Echtzeitsystemen, wie sie bspw. in der Medizintechnik oder der Luftfahrt verwendet

werden, ist der Erfolg eines Systems von der Leistungsfähigkeit in Extremsituationen abhängig [Thaller02: S.128f].

Regressionstest

Nachdem eine Funktion oder ein Modul erfolgreich erstellt und getestet worden ist, kann es vorkommen, dass daran trotzdem noch Änderungen vorgenommen werden müssen. Um sich jedoch abzusichern, dass diese Änderung auch wirksam vollzogen wurde und diese keine negativen Auswirkungen auf andere Teile der Software hat, wird ein sogenannter Regressionstest durchgeführt, bei dem alle vorhandenen Testfälle erneut ausgeführt werden. Voraussetzung dafür ist allerdings, dass dieselben Testfälle bereits vor der Änderung fehlerfrei durchlaufen wurden [Balzert99: S.503].

Akzeptanztest

Akzeptanztests werden auf Grundlage der Spezifikation erstellt und dienen nicht vorrangig der Suche nach Fehlern aus der Programmierung (Verifikation der Software), sondern dienen der Feststellung, ob die Software den Forderungen, Bedürfnissen und Wünschen des Kunden entspricht (Validation der Software). Treten dabei Fehler auf, welche die Entwickler zu verschulden haben, so werden diese in der Regel ohne weitere Kosten behoben. Bei zusätzlichen Anforderungen wird der Auftragnehmer jedoch zur Kasse gebeten. Der Akzeptanztest stellt weiterhin einen Meilenstein im Softwarelebenszyklus dar. Zuvor befindet sich das Produkt in der Entwicklungsphase, wobei die Verantwortung für die Entwicklung beim Auftragnehmer liegt. Mit erfolgreichen Abschluss der Akzeptanztests wechselt die Verantwortung zum Auftraggeber, das Softwareprodukt wird ausgeliefert und befindet sich von diesem Zeitpunkt an in der Pflege- und Wartungsphase [Thaller02: S.162f].

Den Test testen

Der Testprozess ist neben dem Entwurf und der Implementierung ebenfalls ein Teil der Softwareentwicklung und demnach können auch hierbei Fehler auftreten. Eine mögliche Technik zum Testen von Tests stellt die Fehlerinjektion dar. Wie bei allen Tests der Softwareentwicklung, kann auch hierbei die Fehlerfreiheit nicht nachgewiesen werden. Mit Fehlerinjektionen können lediglich Aussagen über die Qualität der Tests getroffen werden. Der Zugriff auf den Code sowie das Konfigurationsumfeld und einer funktionsfähigen Versionskontrollsoftware wird zur Durchführung von Fehlerinjektionen vorausgesetzt. Dabei werden zunächst bewusst Fehler in die Software und deren Umfeld eingebaut. Anschließend wird der normale Testvorgang gestartet. Bei den injizierten Feh-

lern handelt es sich einerseits um typische Programmierfehler, wie bspw. der Austausch bestimmter Vergleichsoperatoren (z.B. „kleiner“ und „kleiner-gleich“) oder der Manipulation von Konstanten, und andererseits um Konfigurationsfehler, wie z.B. das Verweisen der Software auf nicht existierende Dateien. Nach Durchlauf der Tests wird in einer Auswertung die Anzahl der nicht gefundenen Fehler ermittelt, die ein Maß für die Güte der Tests darstellt. Zum Abschluss wird die geänderte Software mit Hilfe der Versionskontrolle auf den Ausgangszustand zurückgesetzt, um die injizierten Fehler wieder zu bereinigen. Für alle Fehler, die nicht gefunden wurden, sollten abschließend entsprechende Testfälle formuliert werden [Vigenschow05: S.197f].

3.3 Testautomatisierung

Das Testen einer Software zählt mit zu den aufwändigsten und teuersten Tätigkeiten in der Softwareentwicklung. Die Kosten dafür werden auf 30% bis 50% der gesamten Projektkosten geschätzt [Thaller02: S.225]. Daher stellt sich irgendwann die Frage, ob diese Kosten durch Automatisierungsvorgänge gesenkt werden können. Die Testfallermittlung und die Testfallautomatisierung sind zwei unterschiedliche Handlungen, die aufeinander aufbauen. Während das oberste Ziel des Testens das Auffinden von Fehlern ist, zielt die Automatisierung der Tests auf eine effiziente und langfristig ökonomische Testdurchführung ab [Vigenschow05: S.159].

Wie bereits im vorherigen Kapitel erläutert wurde, besteht die Testphase im Wesentlichen aus drei Schritten. Zuerst werden die Anforderungen einer Software ermitteln, anschließend der Test durchgeführt und zum Abschluss das Ergebnis mit dem erwarteten Wert verglichen. Der Entwurf eines Testfalls ist eine intellektuell herausfordernde Tätigkeit und deswegen weniger für eine Automatisierung geeignet. Im Gegensatz dazu sollte der Vergleich der Ist- und Soll-Werte generell automatisiert erfolgen, es kann in manchen Situationen jedoch sinnvoller sein dies manuell durchzuführen, nämlich genau dann, wenn der Aufwand den Nutzen übersteigt. Da dieser Prozess allerdings oft mit vielen sich wiederholenden Eingaben verbunden sein kann, die zu viel Zeit des Testers in Anspruch nehmen, eignet sich dieser Schritt ganz besonders für eine Automatisierung [Thaller02: S.231].

3.3.1 Wann lohnt sich die Automatisierung?

Durch die Testautomatisierung soll in erster Linie nicht die Qualität der Software, sondern die Effizienz der Testdurchführung erhöht werden. Man muss sich genau vor Augen halten, dass die Testfallerstellung ebenfalls fehleranfällig ist. So müssen die Test-

fälle möglichst unabhängig von einander durchgeführt werden, damit sie sich nicht gegenseitig beeinflussen und auftretende Fehler somit eine eindeutige Ursache haben. Dieses Vorgehen beschleunigt vor allem die Fehleridentifikation im Debugging-Prozess.

Anforderungen an ein automatisiertes Testsystem

Um dies zu gewährleisten muss vor jedem Test eine konsistente Testumgebung hergestellt und hinterher wieder abgebaut werden. Dies sorgt weiterhin dafür, dass im Fehlerfall das komplette Testskript nicht abgebrochen wird, sondern die restlichen Testfälle weiter durchlaufen können. Zur Identifikation des Fehlers gilt es insbesondere zu prüfen, ob die Ursache dafür wirklich am Testobjekt oder nicht bereits am Testfall zu suchen ist [Vigenschow05: S.160]. Die Erstellung eines automatisierten Testfalls lohnt sich in der Regel nur, wenn das Testobjekt stabil ist und nur wenige Änderungen daran zu erwarten sind. Denn mit jeder Änderung am Testobjekt müssen alle betroffenen Testfälle ebenfalls angepasst werden. Ebenso sollte es sich bei dem Testobjekt nicht um eine Komponente handeln, die nur sehr selten ausgeführt wird. Je mehr Wiederholungen zu erwarten sind, desto mehr lohnt sich ein automatisierter Testfall [Vigenschow05: S.136].

Nutzen durch ein automatisierbares Testsystem

Der größte erkennbare Nutzen, auf den man mit der Testautomatisierung abzielt, liegt in der Zeitersparnis, die man bei langfristigem Einsatz erhält. Durch diesen Zeitgewinn können bspw. zusätzliche Tests abgehandelt werden, die bei manueller Ausführung aus Zeitmangel nicht durchgeführt wurden. Ebenso wäre (allerdings im Extremfall) ein vorzeitiges Release der Software möglich, um das Produkt früher auf den Markt zu bringen als geplant. Neben der Zeitersparnis werden ebenfalls kostbare Ressourcen, wie z.B. die Arbeitskraft des Testers, besser eingesetzt, da sich dieser dann kreativeren Aufgaben widmen kann. Weiterhin können die bereits definierten Testfälle mit kleinen Anpassungen bei ähnlichen Testobjekten wiederverwendet werden. Daraus ergäbe sich ein projektübergreifender Nutzen. Ein weiterer Punkt der für das automatisierte Testen spricht, ist die Möglichkeit umfangreiche Konfigurationstests durchführen zu können, indem die Tests auf unterschiedlichen Computersystemen in verschiedenen Softwareumgebungen zum Einsatz kommen [Thaller02: S.228f].

Stresstests und Lastentests können erst durch die Automatisierung sinnvoll durchgeführt werden. So können Webanwendungen bspw. nur selten unter realistischen Bedingungen auf ihre Belastungsgrenzen hin getestet werden, da eine entsprechende

Anzahl von Nutzern benötigt wird, welche gleichzeitig die Anwendung verwenden. Ebenso ist es, aufgrund der Schnelligkeit heutiger Computersysteme, äußerst unwahrscheinlich eine echte Nebenläufigkeit manuell zu erzeugen. Erst durch die Simulation der benötigten Ressourcen können solche Tests kosteneffizient durchgeführt werden [Vigenschow05: S.196].

Probleme bei der Einführung

Wenn in einem Unternehmen manuelle Tests nur sporadisch durchgeführt werden und es kein festgelegtes Verfahren dafür gibt, dann macht es wenig Sinn bereits mit der Automatisierung zu beginnen. Die Erwartungshaltung, dass mit neuen Methoden und neuen Werkzeugen die Lösung aller Probleme einfach zu erreichen ist, ist meistens ein Irrglaube. Ebenso groß ist die Erwartung, mit der Testautomatisierung viele neue Fehler zu finden. Dies ist jedoch nicht anders als bei manuell durchgeführten Tests. Beim ersten Mal ist die Wahrscheinlichkeit noch hoch auf neue Fehler zu stoßen, während bei jedem weiteren Durchlauf im Allgemeinen nichts Neues gefunden wird. Der eigentliche Wert ist jedoch die Bestätigung dafür, dass sich die Software nach kleinen Anpassungen noch immer so verhält wie gewünscht. Zuerst wird allerdings mindestens ein Verantwortlicher benötigt, der sich dieser neuen Aufgabe voll und ganz widmen kann. Ist dies nicht gegeben, so scheitert das Projekt in den meisten Fällen, da sich für die relevanten Tätigkeiten, wie z.B. die Einrichtung und Wartung des Systems, die Beschreibung der Testfälle und die Schulung der Mitarbeiter, niemand zuständig fühlt. Ein weiteres Problem besteht darin, dass sich die Entwickler in Sicherheit wiegen, solange der automatisierte Test ohne Probleme abläuft. Man könnte glauben, die Software enthielte gar keine Fehler mehr, was jedoch in fast allen Fällen ein Trugschluss ist. Eines der größten Aufwände stellt neben der Definition, die Wartung der Testfälle dar. Denn mit jeder Änderung am Testobjekt müssen die beteiligten Testfälle angepasst werden. In manchen Fällen ist der Aufwand so groß, dass es einfacher ist, bestimmte Tests manuell durchzuführen. Und was man als Testverantwortlicher niemals vergessen sollte ist, dass die zur Testautomatisierung verwendeten Werkzeuge ebenfalls nur Software sind und damit keine Garantie auf Fehlerfreiheit haben [Thaller02: S.230].

3.3.2 Funktionsweise der Web-Testwerkzeuge

Um Webanwendungen automatisiert testen zu können, müssen die Nutzerinteraktionen (Tastatureingaben, Mausklicks) simuliert werden. Dazu gibt es verschiedene Möglichkeiten. Die im Folgenden erläuterten zwei Ansätze habe ich den in Kapitel 4 vorgestell-

ten Testwerkzeugen entnommen. Der Anspruch zur Darstellung aller Automatisierungsverfahren von Webanwendungen ist dabei allerdings nicht gegeben.

Browser-Automatisierung

Diese Testwerkzeuge bringen entweder von Hause aus ihre eigenen Browserimplementierungen mit, oder verwenden die bekannten Standardbrowser. Diese werden durch zusätzlich entwickelte Add-On und Macros gesteuert und ermöglichen die Aufzeichnung von Nutzerinteraktionen (Capture) sowie das automatisierte Abspielen derer (Replay). In den meisten Fällen müssen jedoch nach den Aufnahmen kleinere Anpassungen per Hand vorgenommen werden. Neben der einfachen Testskripterstellung, besteht ein weiterer großer Vorteil darin, dass die Tests genau so durchgeführt werden, wie es der Endnutzer in der späteren Anwendung auch tun würde. Der Tester kann die Aktionen also während der Durchführung live am Bildschirm verfolgen, als würde er mit dem zu testenden Programm interagieren. Dadurch kann das genaue Verhalten der Anwendung auf Clientseite getestet werden [Liquid2]. Der größte Nachteil ist jedoch in der geringen Browserauswahl der Testwerkzeuge zu sehen, da für jeden Browser und jedes größere Release die Erweiterung angepasst werden muss [Sahi]. Diesen enormen Aufwand machen sich viele Werkzeughersteller nicht, sondern beschränken sich dabei auf die Marktführer Firefox und Internet Explorer.

Proxyserver-Ansatz

Einen anderen Ansatz bieten manche Testwerkzeuge mit dem Einsatz eines Proxyservers. Dieser wird (wie in Abbildung 4 zur Verdeutlichung dargestellt ist) zwischen dem Browser und der Webanwendung positioniert und leitet die HTTP-Requests und Responses an den eigentlichen Empfänger weiter. Erhält der Proxy-Server den Response der Webanwendung, so injiziert dieser den entsprechenden JavaScript-Code in den Quellcode der Anwendung und kann somit auf die benötigten HTML-Elemente zugreifen, um die Nutzeraktionen damit aufzeichnen und abspielen zu können. Durch diesen Ansatz ist die Automatisierung völlig unabhängig vom verwendeten Browser. Der Proxy ermöglicht weiterhin die Generierung der Testreports und kann Operationen ausführen, die von Clientseite her nicht zu bewerkstelligen sind, wie z.B. Up- und Downloads [Sahi].

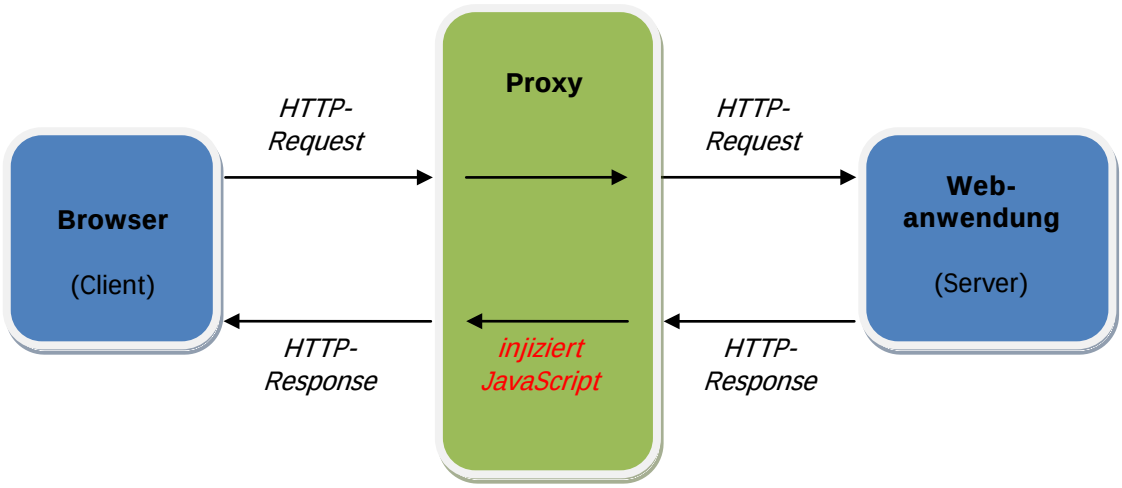


Abbildung 4: Architektur des Proxy-Ansatzes, vgl. [Sahi]

3.3.3 Testskriptarten

Eines haben alle Automatisierungskonzepte gemeinsam: ihre Abläufe werden nämlich durch Skripte gesteuert. Diese Skripte werden einmal definiert und im Idealfall können sie dauerhaft verwendet werden. Doch in den meisten Fällen veralten diese sehr schnell und werden unbrauchbar. Die Wartung solcher automatisierbaren Skripte ist oft so aufwändig, dass diese schon selbst als eigenes Softwareprojekt anzusehen sind. Um die Skripte optimal verwenden und verwalten zu können, sind drei Grundprinzipien zu beachten. Damit die Tester und die Entwickler mit den Skripten arbeiten können, sollten diese stets ausreichend kommentiert sein. Jedes Skript sollte genau eine funktionale Aufgabe beschreiben, damit sie elementar und wiederverwendbar bleiben. Und um eine optimale Lesbarkeit und Wartbarkeit zu gewährleisten, sollte eine geeignete Struktur der Skripte vorhanden sein. Uwe Vigenschow unterscheidet genau fünf Formen von Testskripten, die im Folgenden kurz vorgestellt werden [Vigenschow05: S.192].

Lineare Skripte

Die einfachste Form stellen die Capture and Replay-Skripte dar. Dabei werden die einzelnen Interaktionselemente, wie z.B. Tastenschläge oder Mausklicks, die je nach Umsetzung entweder auf einzelne HTML-Elemente oder absoluten Bildschirmpositionen bezogen sind, aufgezeichnet. Diese Elemente sind dann in einem Skript linear hintereinander aufgelistet. Die Definition solcher Skripte benötigt kaum Vorbereitungszeit und es sind auch keine Programmierkenntnisse nötig. Weil nachträgliche Anpassungen oft sehr aufwändig sind, werden sie kaum gewartet, sondern meistens einfach neu erstellt. Ebenso ist eine umfassende Wiederverwendung der Skripte bei anderen Testobjekten

kaum möglich, da diese Skripte sehr fehleranfällig gegenüber Programm- bzw. Ablaufänderungen sind [Vigenschow05: S.192f].

Strukturierte Skripte

Eine verbesserte Form der linearen Skripte stellen die strukturierten Skripte dar. Diese setzen allerdings Programmierkenntnisse voraus, da die Anweisungen zur Steuerung der Ausführung um Bedingungen, Selektionen und Iterationen erweitert werden. Durch die Schleifen können Abläufe einfacher formuliert und kürzere Skripte erstellt werden, was sich wiederum positiv auf die Wartung auswirkt. Durch die bedingten Anweisungen kann auf diverse Veränderungen zur Laufzeit reagiert werden, was wiederum zu robusten, flexiblen und fehlertoleranten Skripten führt. Ein Nachteil stellt allerdings die feste Integration der Testdaten im Skript dar [Vigenschow05: S.193].

Verteilte Skripte

Verteilte Skripte verlagern die gesamte Funktionalität eines Testfalls in verschiedene Dateien, um die Basisfunktionalitäten wiederverwenden zu können. So kann z.B. eine Trennung zwischen anwendungsspezifischen und anwendungsunabhängigen Teilen vorgenommen werden. Dadurch lassen sich die Tests noch einfacher und schneller erstellen, während der Wartungsaufwand spürbar sinkt. Vorhandene Redundanzen werden entfernt bzw. gar nicht erst programmiert. Die komplexere Dateistruktur stellt allerdings einen Nachteil dar [Vigenschow05: S.193f].

Datengetriebene Skripte

Durch die Trennung der Testdaten von den Skripten, können dieselben Testabläufe mit unterschiedlichen Daten durchgeführt werden, was die Effizienz enorm steigern kann. Die Verwaltung der Daten erfolgt in separaten Dateien oder in einer Datenbank. Damit kann die Pflege der Daten unabhängig von der Programmierung durchgeführt werden, was insbesondere Fachexperten ohne Programmierkenntnisse sehr entgegen kommt. Der initiale Programmieraufwand ist jedoch erheblich und gerade deshalb sollten die Skripte spätestens jetzt wie ein Softwareprojekt organisiert und verfolgt werden [Vigenschow05: S.194].

Schlüsselwortgetriebene Skripte

Um in den datengetriebenen Skripten nun auch die Skriptabläufe ohne Programmierkenntnisse erstellen und warten zu können, werden elementare Anweisungen vorprogrammiert und mit einem Schlüsselwort versehen. Dadurch ist die Anzahl der Testskrip-

te nicht mehr von der Anzahl der Testfälle abhängig, sondern wächst mit der zu testenden Funktionalität der Software, wodurch auch Wartungsaufwände minimiert werden. Der initiale Programmieraufwand ist hierbei maximal. Deshalb stellt die gesamte Testskriptprogrammierung ein echtes Softwareprojekt dar [Vigenschow05: S.194].

3.4 Webtechnologien

In den folgenden Abschnitten soll ein Grundverständnis über Funktion und Aufbau des World Wide Web geliefert werden. Dazu wird u.a. auf die aktuellen Technologien eingegangen und Testmöglichkeiten der daraus entstehenden Dokumente aufgezeigt. Abschließend werden die wichtigsten Unterschiede zwischen Webanwendungen und Desktopanwendungen diskutiert.

3.4.1 World Wide Web

Aus technologischer Sicht basiert das Web komplett auf dem Client-Server-Modell, wobei die Server (zu Deutsch: Diener) Informationen bereitstellen und die Clients (zu Deutsch: Nutzer) auf diese Informationen zugreifen. Im World Wide Web (kurz: WWW) entspricht der Client immer dem Webbrowser, der auf dem Rechner des Nutzers installiert ist, während der Webserver eine Anwendung auf einem entfernten Rechner darstellt. Beide sind zusammen über das Internet miteinander verbunden. Möchte der Nutzer nun auf Informationen zugreifen, so gibt dieser eine eindeutige Adresse des gewünschten Dokuments (URL) in den Webbrowser ein. Daraufhin wird eine spezielle Anfrage (HTTP-Request) generiert und über das Internet abgeschickt. Der Webserver erhält die Anfrage, sucht nach dem HTML-Dokument, das mit der Adresse gekennzeichnet ist und schickt es dem Nutzer als HTTP-Response zurück. Der Browser empfängt die HTML-Datei und versucht diese darzustellen. Das HTTP ist also für die Übertragung zuständig und operiert im Hintergrund, während HTML als Auszeichnungssprache der Dokumente dient, mit der die Texte strukturiert und verlinkt werden können. Dies ist die grundlegende Funktionsweise des WWW. Alle zusätzlichen Technologien sind lediglich als Unterstützung anzusehen [HWM06: S.35f].

Kernstandards

Das W3-Konsortium (World Wide Web-Konsortium, kurz: W3C) ist ein Standardisierungsgremium speziell für Webtechnologien, welches vom Erfinder der HTML-Sprache Tim Berners-Lee geleitet wird. Der aktuelle Webstandard ist jedoch nicht mehr HTML, sondern XHTML. Dabei handelt es sich nach wie vor um HTML, das nun aber XML-

konform geschrieben wird. XML stellt wiederum eine Sprache dar, mit der andere Sprachen (wie bspw. XHTML) definiert werden können. HTML und auch XHTML besitzen eine einfache Grundstruktur und können mit Hilfe eines beliebigen Texteditors erstellt und bearbeitet werden [HWM06: S.84]. Trotz der sehr fehlertoleranten Browser, können die dargestellten Webseiten von Browser zu Browser stark abweichen, was oftmals auf eine Missachtung der Standards zurückzuführen ist. Um sicherzustellen, dass die Webseite den Webstandards folgt, kann diese mit Hilfe eines Validators (bspw. W3C-Validator) gegen eine entsprechende Dokumententypdefinition geprüft werden. Solche Tests lassen sich auch relativ einfach in automatisierte Testsysteme integrieren.

Neben HTML/XHTML stellt CSS (Cascading Style Sheets) eine weitere wichtige Sprache zum Erstellen von Webseiten dar, die ebenfalls vom W3C verwaltet wird. Mit Hilfe dieser Sprache können Layout-Befehle definiert werden, um den Inhalt einer Webseite von deren Darstellung trennen zu können. Dadurch lassen sich Webseiten besser warten und leichter aktualisieren [HWM06: S.112f]. Die syntaktische Korrektheit der CSS-Strukturen kann ebenfalls durch Zuhilfenahme eines Validators abgesichert werden.

JavaScript ist eine clientseitige Skriptsprache, die ursprünglich von der Firma Netscape entwickelt worden ist und heute in allen aktuellen Browsern zum Einsatz kommt. Clientseitig bedeutet, dass die in einer HTML-Datei befindlichen JavaScript-Anweisungen nur vom Browser ausgeführt werden und dabei keine Interaktion mit dem Webserver stattfindet. JavaScript wird häufig dazu genutzt, dem Nutzer bei der Interaktion mit der Webseite bzw. Webanwendung zu unterstützen. Z.Bsp. indem Zusatzfenster ohne Navigationsleisten (sogenannte Pop-Ups) geöffnet werden können oder Formulare vor dem Absenden auf Vollständigkeit geprüft werden. Die Realisierung von Kernfunktionen mittels JavaScript ist nicht zu empfehlen, da dem Nutzer aus Sicherheitsgründen die Möglichkeit geboten wird, diese Technologie im Browser abzuschalten [HWM06: S.240f].

JavaScript kann auch mit dem Server interagieren. Diese Technik ist unter dem Schlagwort AJAX bekannt geworden, das eine Abkürzung für „Asynchrous JavaScript + XML“ darstellt. Ist eine Webseite vollständig geladen, so kann durch Auslösen eines beliebigen JavaScript-Ereignisses eine Anfrage an ein serverseitiges Skript gestellt werden, das entweder in einer beliebigen serverseitigen Technologie geschrieben ist oder auch ein XML-Dokument sein kann. Daraufhin erhält das JavaScript die Antwort vom serverseitigen Skript und kann diese Informationen im Browser darstellen, ohne dass die aktuelle Seite neu geladen werden muss. Die Asynchronität ist darin begründet,

dass das JavaScript weiterhin ausgeführt werden kann, während der Webserver noch antwortet. Durch Nutzung dieser Technologie konnte die Qualität von Webservices verbessert werden und sie führte zu einer steigenden Entwicklung interaktiver Benutzerschnittstellen auf Webseiten [HWM06: S.331f].

Zusatzstandards

Flash ist eine clientseitige Technologie, die es erlaubt, Vektorgrafiken und Vektoranimationen im Webbrowser darzustellen. Zusätzlich bietet der Flash-Player eine sehr gute Unterstützung für das Abspielen von Videos sowie Sounds und verfügt über eine eigene Skriptsprache, dem Actionscript, die zur Steuerung der Elemente dient. Ein Flash-Film wird in eine HTML-Seite eingebettet und vom Webserver an den Webbrowser übertragen. Zur Darstellung des Flash-Films muss der Browser über ein Plug-In verfügen können, nämlich dem Flash-Player. Dieser ist für die wichtigsten Plattformen verfügbar (Windows, Linux, Mac) und ist in einigen Browsern bereits integriert. Dementsprechend weit verbreitet ist diese Technologie [HWM06: S.44]. Zum automatischen Testen dieser Technologie bietet bspw. Selenium RC mittels einfacher JavaScript-Anweisungen die Möglichkeit, Buttons zu aktivieren und Variablenwerte auszulesen. Dazu muss der Tester allerdings über den Quellcode verfügen können (anders als bei HTML-Seiten kann dieser nicht über den Browser angezeigt werden), da Zusatzcode in das Flash-Objekt integriert werden muss [Selenium2].

Java Applets sind kleine Java Programme die in einem Webbrowser lauffähig sind. Sie sind u.a. der Grund dafür, dass Java heute so weit verbreitet ist. Trotzdem sind solche Anwendungen nur noch selten auf Webseiten zu finden. Dies liegt vor allem daran, dass heutzutage viele Aufgaben mit den Kernstandards realisierbar sind, die damals nur mit Hilfe von JavaApplets ermöglicht werden konnten. Ein weiter Grund dafür sind die begrenzten grafischen Möglichkeiten, die einem Webdesigner eher zu alternativen Technologien greifen lässt [JavaApp]. Da diese Anwendungen lediglich in einfache HTML-Seiten eingebettet werden, bieten sich zum Test der Anwendungen die javaüblichen Testwerkzeuge, wie bspw. JUnit, an. Zusätzlich können Web-Testwerkzeuge verwendet werden, welche diese Technologie unterstützten. Einige davon werden im Kapitel 4.3.2 (Vergleich aktueller Testwerkzeuge) detaillierter betrachtet.

Eine den JavaApplets sehr ähnliche Technologie stellt ActiveX dar. Dies sind Programmiersprachen-unabhängige Softwarekomponenten, die allerdings ausschließlich für den Internet Explorer unter Microsoft Windows lauffähig sind. Diese Softwarekomponenten

(auch ActiveX Controls genannt) können mit dem Betriebssystem viel enger zusammenarbeiten als bspw. JavaApplets, was einerseits eine Vielzahl neuer Möglichkeiten bietet, andererseits aber auch ein hohes Sicherheitsrisiko darstellt [ActiveX]. Analog zu JavaApplets, können ActiveX Controls zum Einen mit Hilfe von programmiersprachenspezifischen Testwerkzeugen getestet werden und zum Anderen durch Einsatz von geeigneten Web-Testwerkzeugen, die ActiveX unterstützen.

Die eXtensible Markup Language (XML) ermöglicht eine Vielfalt von innovativen Web-Technologien und ist im Grunde eine Metasprache, mit der weitere Sprachen definiert werden können. Oft wird sie jedoch lediglich zu Datenhaltung verwendet. Jedes XML-Dokument besteht aus Befehlen (sogenannten Tags), welche Attribute besitzen können. Weiterhin existiert immer ein Wurzelelement, von dem das gesamte Dokument aus in Form einer Baumstruktur aufgebaut ist. XSD (eXtensible Schema Definition) ist z.Bsp. eine XML-Sprache, mit der die Struktur von XML-Dokumenten definiert werden kann und die Gültigkeit des Dokuments geprüft wird. Eine weitere XML-basierte Technologie stellt XSL (eXtensible Stylesheet Language) dar. Damit können XML-Dokumente in andere Dokumente transformiert werden. SVG (Scalable Vector Graphics) stellt eine konkurrenzfähige Alternative zur Macromedias Flash-Technologie dar, welche sich allerdings im Massenmarkt bisher nicht durchsetzen konnte [HWM06: S.46f].

Skriptsprachen und kompilierte Sprachen

Mit den bisher vorgestellten Technologien können lediglich statische Webseiten erstellt werden, bei denen jede einzelne Seite in der Regel einem eigenen HTML-Dokument entspricht. Dynamische Webseiten werden hingegen über serverseitige Technologien zur Laufzeit generiert und ermöglichen neben einer verbesserten Datenhaltung, vor allem die Nutzung der üblichen Programmiertechniken. Es handelt sich dabei dennoch um HTML-Dokumente, bei denen mit Hilfe von geeigneten Web-Testwerkzeugen definierte Funktionstests automatisiert durchgeführt werden können. Da serverseitige Technologien generell clientunabhängig sind, gibt es sehr viele Alternativen auf diesem Gebiet. Zu den wichtigsten zählen PHP, Java Server Pages (JSP), Active Server Pages .NET (ASP.NET) und Ruby on Rails. Für nahezu alle kompilierbaren Sprachen bzw. Skriptsprachen existieren entsprechende Unittest-Frameworks, mit denen die erstellten Programme getestet werden können. Ruby on Rails ist in dieser Hinsicht besonders interessant, da das Framework Mechanismen für eine komplett-testgetriebene Entwickelung bereitstellt.

3.4.2 Webanwendungen vs. Desktopanwendungen

Im Vergleich zu klassischen Desktopanwendungen grenzen sich Webanwendungen deutlich von diesen ab. Während Desktopanwendungen stets auf dem Rechner des Nutzers installiert werden müssen, werden Webanwendungen auf Webservern betrieben, die zur Interaktion mit dem Nutzer ausschließlich einen Webbrowser benötigt. Da dieser auf den meisten Systemen bereits vorhanden ist, muss keine zusätzliche Software installiert werden (mit Ausnahme bestimmter Plug-Ins, wie bspw. dem Flash Player). Dies ermöglicht den Webanwendungen eine weitestgehende Plattformunabhängigkeit, sofern diese keine plattformabhängigen Technologien, wie z.Bsp. ActiveX, verwenden. Ein weiterer Vorteil gegenüber Desktopprogrammen ist darin begründet, dass Änderungen an einem zentralen Punkt der Software vorgenommen werden, was sich wiederum positiv auf die Wartungskosten auswirken kann. Denn bei Desktopanwendungen werden Anpassungen in Form von Updates vertrieben, die vom Nutzer bezogen und installiert werden müssen. Da dies allerdings oft ein optionaler Vorgang ist, müssen solche Systeme einen reibungslosen Ablauf mit unterschiedlichen Programmversionen ermöglichen. Ein weiterer Pluspunkt ist die stetige Browserentwicklung für Mobiltelefone oder PDAs, da Webanwendungen ohne zusätzlichen Entwicklungsaufwand auf diesen Endgeräten betrieben werden könnten. Die möglichen Darstellungsunterschiede der verschiedenen Browser sind jedoch allgemein als nachteilig zu bewerten, da ein nicht unerheblicher Aufwand darin investiert werden muss, die Anwendungen über verschiedene Browser und sogar Browserversionen konsistent zu halten. Auch die Performanz stellt bei Webanwendungen einen kritischen Faktor dar, welche insbesondere von der Bandbreite der Verbindung abhängig ist. Wenn man jedoch als Beispiel die web-basierten Office-Anwendungen[4] der Google Inc. betrachtet, ist ein deutlicher Trend zu erkennen, bei dem immer mehr klassische Desktopanwendungen als Webanwendung realisiert werden.

[4] URL: http://docs.google.com

4 Testwerkzeuge

In diesem Kapitel werden zehn verschiedene Werkzeuge zum automatisierten Testen von Webanwendungen vorgestellt und miteinander verglichen. Im ersten Unterkapitel wird zunächst geklärt, warum gerade diese Testwerkzeuge für die Untersuchungen ausgewählt wurden. Im darauf folgenden Unterkapitel werden diese kurz vorgestellt, wobei Canoo Webtest und Selenium einer detaillierteren Betrachtung unterzogen werden. Im letzten Unterkapitel findet dann der eigentliche Vergleich der kommerziellen und quelloffenen Testwerkzeuge statt. Dieser stützt sich auf elf Kriterien, welche insbesondere bei der Beschaffung eines Testwerkzeugs maßgeblich sind.

4.1 Auswahlkriterien

Für den Vergleich in Abschnitt 4.3.2 habe ich mich speziell auf fünf kommerzielle und fünf quelloffene Systeme beschränkt, um u.a. signifikante Unterschiede der beiden Arten aufdecken zu können. Bei meiner Recherche - nach geeigneten kommerziellen Testwerkzeugen - nutzte ich vor allem Suchmaschinen und folgte Werbebannern auf fachspezifischen Webseiten. Für die Auswahl quelloffener Testwerkzeuge nutzte ich zusätzlich die Werkzeugsammlungen von SoftwareQATest[5] und opensourcetesting[6].

In meinen Untersuchungen konzentrierte ich mich im Wesentlichen auf reine Testwerkzeuge für Webanwendungen. Die „großen“ Testprogramme, wie z.B. TestComplete (von der Firma AutomatedQA Corporation) oder SilkTest (von der Firma Borland Software Corporation) sind vorrangig für das Testen umfangreicher Desktop- und Systemprogramme ausgelegt. Diese stellen ebenfalls Funktionen zum Testen von Webanwendungen bereit, welche allerdings nicht den Fokus des Produkts bilden und somit (vermutlich) kaum signifikante Unterschiede zu web-spezialisierten Testwerkzeugen zeigen. Des Weiteren kamen ausschließlich solche Testwerkzeuge in Betracht, welche die Durchführung von funktionalen Tests (auch GUI-Tests genannt) ermöglichen. Werkzeuge zur automatisierten Durchführung von Performance-Tests dienen nicht in erster Linie dem Aufspüren von Programmfehlern, sondern sollen vor allem die Leistungsfähigkeit des Softwaresystems ermitteln. Mit sogenannten Linkchecker, die zum Aufspüren von defekten Verlinkungen dienen, können ebenfalls Programmfehler in Webanwendungen aufgespürt werden, allerdings immer nur diesen einen Fehlertyp. Und da

[5] http://www.softwareqatest.com
[6] http://www.opensourcetesting.org

solche Prüfungen ebenfalls mit funktionalen Testwerkzeugen durchführbar sind, werden Linkchecker nicht näher betrachtet.

In dieser Arbeit werden diese kostenlosen Werkzeuge weder frei noch als Open Source bezeichnet, sondern zusammenfassend als quelloffen, da sich diese beiden Begriffe zwar sehr ähnlich sind, aber nicht dasselbe meinen. Open Source wurde 1998 erstmals als Marketingbegriff für freie Software eingeführt, da das Wort „frei“ ein Mehrdeutigkeitsproblem besitzt. Richard Stallman (Initiator und Verfechter der freien-Software-Bewegung) meint nicht frei im Sinne von kostenlos, was besonders zu Akzeptanzproblemen in der Geschäftswelt führte, sondern im Sinne von frei über die Software verfügen zu können. Um diesen Gedanken besser in die Geschäftswelt portieren zu können, entstand der Open Source-Begriff [GNU1]. Das interessanteste Merkmal von freier und Open Source-Software ist die Möglichkeit und das Recht zur Einsicht sowie Änderung des Quelltexts. Gerade Unternehmen könnten davon profitieren, indem sie die Software auf ihre spezifischen Anforderungen zuschneiden. Ein weiterer Vorteil besteht darin, dass man durch den Besitz des Quellcodes weitestgehend unabhängig vom Hersteller der Software ist. [GNU2].

4.2 Aktuelle Testwerkzeuge

4.2.1 Kommerzielle Testwerkzeuge

iMacros (iOpus Software, Walldorf/Heidelberg – Deutschland)

iMacros gibt es in drei Business-Editionen mit jeweils unterschiedlichem Funktionsumfang. Die Probiervariante ist 30 Tage gültig und kann uneingeschränkt genutzt werden. Für den Internet Explorer und den Firefox Browser existiert ein kostenloses Plug-In, das die Automatisierung von einfachen HTML-Seiten unterstützt. Für dynamische Webseiten, wie bspw. Flash- oder Java-Applets, steht dem Anwender eine Bilderkennungsfunktion zur Verfügung [iMacros].

LiquidTest (JadeLiquid Software Pty Ltd, Hobart – Tasmanien)

Ebenso wie iMacros nutzt auch LiquidTest den Ansatz der Browser-Automatisierung um Webseiten automatisiert testbar zu machen. Die Developer-Edition ist speziell auf die Bedürfnisse der Entwickler von Webanwendungen zugeschnitten und bietet den größten Funktionsumfang für die Erstellung von Skripten. Die Tester Edition hingegen legt den Fokus speziell auf Testteams und ermöglicht vor allem technisch-unerfahrenen

Anwendern einen leichten Umgang mit der Software. Die 30-tägige Probeversion kann auch hier in vollem Umfang genutzt werden [Liquid1].

QEngine (ZOHO Corporation, Pleasanton – USA)

QEngine wird noch immer unter dem ursprünglichen Firmennamen AdventNet Inc. vertrieben, der aufgrund einer Umstrukturierung in ZOHO Copr. geändert worden ist [QEngine]. Zum Produktumfang gehört neben dem Web-Funktional-Testing- auch das Web-Performance-Testing-Paket, die aber beide separat bezogen werden können. Um sich aber vorab schon einmal einen Eindruck von der Software zu verschaffen, steht die 15-tägige Probierversion, die Livedemo (leider nur unter veralteten Browserversionen lauffähig) sowie ein Präsentationsvideo zur Verfügung, das dem Nutzer mit der Bedienung des Programms vertraut macht.

Squish for Web (froglogic GmbH, Hamburg – Deutschland)

Froglogic bietet mit Squish eine breite Palette von Testwerkzeugen für Anwendungen verschiedenster Programmiersprachen an, wie z.B. für Java- (Squish for Java) oder Qt[7]-basierte Anwendungen (Squish for Qt). Squish for Web beschränkt sich auf das Testen von Webanwendungen und lässt sich leicht in Mercury Quality Center und anderen Testframeworks integrieren. Durch die verwendeten Skriptsprachen sind die Testskripte sowohl Browser- als auch Plattform-unabhängig, womit bspw. die unter Safari[8] erstellten Skripte auch auf dem Firefox unter einem Windows-System lauffähig wären [Vat-Squi1].

Web2Test (itCampus Software- und Systemhaus GmbH, Leipzig – Deutschland)

Für die Erstellung und Pflege der Testskripte in Web2Test sind keinerlei Programmierkenntnisse notwendig. Die Testskripte werden über den integrierten Browser aufgenommen, welcher auf der Mozilla-Technologie basiert. Höchste Flexibilität wird durch das datengetriebene Vorgehen ermöglicht, womit Testdaten aus Excel-Dateien, CSV-Dateien oder Datenbanken in die Testskripte integrieren werden können [VaT-Web23].

[7] Qt ist eine plattformübergreifendes Anwendungs- und UI-Framework (User Interface). Qt-basierte Programme können ohne zusätzliche Änderungen auf mehreren Plattformen portiert werden.
[8] Apple's Standardwebbrowser für das Mac OS X Betriebssystem

4.2.2 Quelloffene Testwerkzeuge

Sahi (Sahi Software, Bangalore – Indien)

Im Grunde startet Sahi einen Proxy Server, über den der verwendete Browser die Anfragen zum Internet stellt. Sahi manipuliert mittels JavaScript den Verkehr zwischen dem Browser und der Webanwendung, sodass auf die relevanten Elemente der Webseite zugegriffen und die jeweilige Nutzerinteraktion simuliert werden kann [Sahi]. Der größte Vorteil dabei ist die Browserunabhängigkeit, die sowohl den Entwicklern (es muss keine browserspezifische Wartung vorgenommen werden), als auch den Endnutzern (durch freie Browserwahl) zu Gute kommt. Über die Flashdemo auf der Webseite kann ein erster Eindruck des Testwerkzeugs gewonnen werden.

Selenium (ThoughtWorks, Chicago – USA)

Selenium ist eines der weitverbreitesten Testwerkzeuge und genießt hohes Ansehen unter den Webentwicklern. Nach fünf-jähriger Entwicklungsphase konnte im Mai 2009 die Betaphase beendet und das 1.0-Release veröffentlicht werden [VaT-Sele2]. Nicht zuletzt ist dies auch ein Grund dafür, das Programm im nächsten Abschnitt (Kapitel 4.2.4) genauer unter die Lupe zu nehmen.

Watir (Hauptentwickler: Bret Pettichord und Paul Rogers)

Watir ist ein Akronym für „Web Application Testing in Ruby“ und wird wie das englische Wort „water“ ausgesprochen. Wie es der Name schon vermuten lässt, basiert das Werkzeug auf der Programmiersprache Ruby. Diese nutzt die OLE-Schnittstelle des Internet Explorers, um auf den DOM und deren Elemente zuzugreifen [VaT-Wati2]. Watir ist ursprünglich nur für den Internet Explorer entwickelt worden, seit Sommer 2008 zählt aber FireWatir (Watir für den Firefox Browser) zum festen Bestandteil des Projekts [VaT-Wati1].

WebInject (entwickelt von Corey Goldberg)

Dieses Werkzeug ist in der Programmiersprache Perl entwickelt worden und kann damit als plattformunabhängig betrachtet werden, da es auf jeder Plattform lauffähig ist, auf der ein Perl-Interpreter vorinstalliert wurde. Zur Beschreibung der Testfälle wird XML verwendet [VaT-WebI1]. Leider liegt das letzte Release über 3 Jahre (Januar 2006) zurück, sodass man vermuten könnte, die Entwicklung sei fertiggestellt oder eingestellt worden.

WebTest (Canoo Engineering AG, Basel – Schweiz)

Ebenso plattformunabhängig wie WebInject ist das auf Java-basierende WebTest der Firma Canoo, das zur Ausführung lediglich das Java Development Kit (JDK) benötigt. Die Testfälle können sowohl in XML, als auch in der objektorientierten Skriptsprache Groovy formuliert werden [VaT-WebT1]. Ebenso wie Selenium wird auch Canoo WebTest im Folgenden einer detaillierten Betrachtung unterzogen.

4.2.3 Detailbetrachtung – Canoo WebTest

Da ich zur Bearbeitung der Juleica-Fallstudie (Kapitel 5.1.) viele Erfahrungen mit Canoo WebTest sammeln konnte, möchte ich in diesem Abschnitt den Umfang und die Möglichkeiten des Testwerkzeugs etwas detaillierter darstellen.

Architektur

Der Kern von Canoo WebTest besteht aus HTMLUnit, dem Kommandozeilen-basierten Browser für Java-Programme. Dieser modelliert HTML Dokumente und verfügt über eine Programmierschnittstelle, mit deren Hilfe man die Nutzerinteraktionen, wie bspw. das Klicken von Links oder das Ausfüllen von Formularen, auf der HTTP-Ebene simulieren kann [HTMLUnit]. Die Testfälle können in XML (Extensible Markup Language) oder alternativ in der an Java angelehnten Skriptsprache Groovy verfasst werden. XSLT (Extensible Stylesheet Language Transformations) wird für die Darstellung der Testergebnisse - dem Reporting - verwendet. Für das optimale Zusammenspiel der Komponenten sorgt das plattformunabhängige Buildtool Apache ANT. Für die Installation und Inbetriebnahme des Testwerkzeugs wird lediglich das JDK in der Version 5 oder höher benötigt, alles weitere bringt das Werkzeug mit sich. Für die Testfallentwicklung empfiehlt es sich zusätzlich den WebTest-Recorder zu installieren. Dieser zeichnet die Aktionen des Nutzers auf und wandelt diese in die gewünschte Skriptsprache um. Leider wird das Plug-In nur für den Firefox bereitgestellt [WebTest].

Testablauf

Nachdem das Testwerkzeug erfolgreich eingerichtet worden ist und anhand des integrierten Demoprojekts getestet wurde, kann man mit dem Erstellen der eigenen Testfälle beginnen. Wählt man dabei XML als Beschreibungssprache, so wäre ein XML-Editor vorteilhaft, jedoch würde ein einfacher Texteditor ebenfalls ausreichen. Abbildung 5 stellt den Quellcode eines einfachen Testfalls in XML dar. Jede Anweisung zwischen den step-Tags entspricht in diesem Beispiel genau einem Testschritt. Zunächst wird mit

dem invoke-Befehl die zu testende Webseite aufgerufen. VerifyText prüft den HTTP-Response auf die gewünschte Zeichenfolge. Wurde diese nicht gefunden, so bricht der Test an der Stelle sofort ab. Die Anweisung setInputField nimmt Eingaben in Formularfeldern vor. In diesem Beispiel sollen der Benutzername und das Passwort angegeben werden. Die clickButton-Anweisung schickt das Formular bzw. die Loginanfrage ab und das abschließende verifyText-Element überprüft, ob der Zugang gestattet wurde.

```
01:    <webtest name="Automatischer Login">
02:        <steps>
03:            <invoke url="http://www.juleica-antrag.de"/>
04:            <verifyText text="Hier kannst Du Deine Juleica beantragen"/>
05:            <setInputField name="user" value="AntragstellerXY"/>
06:            <setInputField name="pass" value="geheim"/>
07:            <clickButton label="Login"/>
08:            <verifyText text="Herzlich Willkommen AntragstellerXY!"/>
09:        </steps>
10:    </webtest>
```

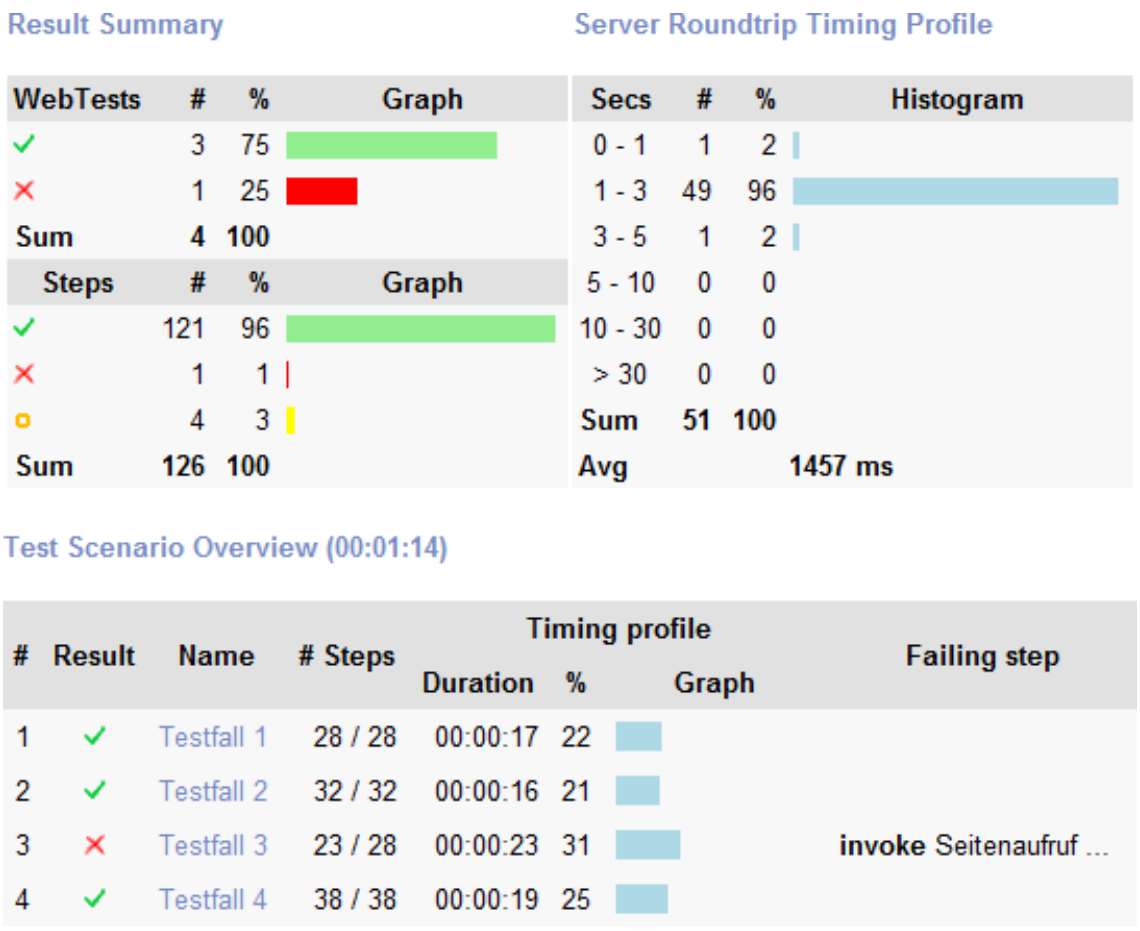
Abbildung 5: Canoo WebTest – Ein einfacher Testfall in XML

Um Fehler in der Entwicklung schon frühzeitig zu erkennen, sollten die Testfälle bereits nach kleinen Änderungen komplett durchlaufen werden. Dies kann, abhängig vom Umfang des Testfalls, in wenigen Sekunden erledigt sein. Die programminternen Abläufe werden in einer Log-Datei abgelegt und das Testergebnis detailliert in HTML dargestellt (siehe Abbildung 6). Dieser Testreport ist in drei Abschnitte gegliedert. Der erste Teil liefert eine Zusammenfassung des Testergebnisses, indem die erfolgreichen, die fehlgeschlagenen und die nicht durchgeführten Tests, welche aus den fehlgeschlagenen Tests resultieren, prozentual und als Grafik angegeben werden. Der zweite Teil stellt die Zugriffszeiten der aufgerufenen Webseiten dar, um Performanzprobleme ausfindig zu machen. Der dritte Teil stellt den gesamten Testablauf strukturiert dar. Der Ursachen von fehlgeschlagenen Testschritten kann somit schnell nachgegangen werden.

Funktionsumfang

WebTest simuliert die Aktionen des Nutzers (bzw. des Testers) mit der Webanwendung. Dazu gehören u.a. das Aufrufen von Webseiten, das Klicken von Links und Buttons sowie das Ausfüllen von Formularfeldern, wie bspw. Checkboxen, Auswahlboxen oder Eingabefelder. Auch versteckte Felder, sogenannte „hidden input fields“, können automatisiert mit Werten versehen werden. Weiterhin unterstützt WebTest den Um-

gang mit mehreren Fenstern, Frames und Cookies[9]. Neben dem Setzen von Werten ist es vor allem auch wichtig zu prüfen, ob bestimmte Elemente vorhanden sind oder ein gewisses Ereignis eingetreten ist. Dafür wird der HTTP-Response eines Webservers (der zurückgelieferte HTML-Quellcode) auf die zu testenden Werte hin untersucht. Für JavaScript-Elemente, wie bspw. den Dialogboxen oder Maus-Events, stehen ebenfalls entsprechende Anweisungen zum Auslösen und Prüfen bereit. Auch HTML-übergreifende Inhalte können automatisiert getestet werden. So können auf einzelne Bestandteile einer E-Mail zugegriffen, Inhalte von PDF-Dokumenten ausgelesen und Datensätze von Excel-Tabellen ausgewählt werden. Durch Zusammenspiel mit SQLUnit können sogar Datenbanken getestet werden. Zum Einen ließen sich damit SQL-Befehle intern testen und zum Anderen könnte man bspw. überprüfen, ob nach der Nutzerregistrierung auf einer Webseite tatsächlich ein Eintrag in der Datenbank erfolgte [Web-Test].



Result Summary

WebTests	#	%	Graph
✓	3	75	
✗	1	25	
Sum	4	100	

Steps	#	%	Graph
✓	121	96	
✗	1	1	
○	4	3	
Sum	126	100	

Server Roundtrip Timing Profile

Secs	#	%	Histogram
0 - 1	1	2	
1 - 3	49	96	
3 - 5	1	2	
5 - 10	0	0	
10 - 30	0	0	
> 30	0	0	
Sum	51	100	
Avg		1457 ms	

Test Scenario Overview (00:01:14)

#	Result	Name	# Steps	Timing profile			Failing step
				Duration	%	Graph	
1	✓	Testfall 1	28 / 28	00:00:17	22		
2	✓	Testfall 2	32 / 32	00:00:16	21		
3	✗	Testfall 3	23 / 28	00:00:23	31		**invoke** Seitenaufruf ...
4	✓	Testfall 4	38 / 38	00:00:19	25		

Abbildung 6: Canoo WebTest – Der Testreport

[9] Cookies werden von Webanwendungen auf den lokalen Rechner des Nutzers abgelegt und enthalten Informationen, die für die Anwendung relevant sein können. Dies können z.B. getätigte Nutzereinstellungen, der Sitzungsschlüssel oder Warenkorbinformationen eines Onlineshops sein.

Eigene Erfahrungen

Das Installation des Testwerkzeugs empfand ich als unkompliziert und sollte selbst Ungeübten in wenigen Minuten gelingen. Das Demoprojekt wird durch einen einzigen Befehl erzeugt und kann als Vorlage für die eigenen Testfälle genutzt werden. Ein Testfallentwickler sollte sich im Quelltext einer Webseite zurechtfinden können, da leider nicht alle HTML-Elemente mit dem WebTest Recorder ausgewählt werden können. Aus diesem Grund gibt es die Möglichkeit HTML-Elemente mittels XPath[10] anzugeben. Hierfür kann das XPather Plug-In für den Firefox sehr hilfreich sein, das die benötigten Pfade u.a. ganz unkompliziert per Mausklick erzeugen kann, jedoch ohne Grundkenntnisse in XPath nur schwer zu bedienen ist. Zusätzlich können Erfahrungen im Umgang mit Java und Apache ANT zum Erstellen der Testfälle hilfreich sein.

Trotz der umfangreichen Dokumentation ist es mir jedoch nicht gelungen alle Probleme und Unklarheiten zu beseitigen. So gab es z.B. Unstimmigkeiten im Umgang mit JavaScript. Das Auslösen eines onClick-Events funktionierte auf einigen Webseiten nicht. Kurioserweise führte dann der Aufruf des hierarchisch darüber liegenden HTML-Elements zum Erfolg, obwohl dieses kein onClick-Event besaß. Die Ursache dafür habe ich bisher nicht ausfindig machen können. Ebenso ist es mir nicht gelungen einen zufriedenstellenden E-Mail-Test zu erstellen. Die Funktionsweise dafür ist in der Anleitung eindeutig beschrieben, doch in der Umsetzung blieben die gewünschten Ergebnisse außen vor. Trotzdem habe ich aber den Eindruck gewonnen, dass es sich bei Canoo WebTest um ein mächtiges und solides automatisierbares Testwerkzeug handelt, dessen Einsatz sich langfristig lohnen kann.

4.2.4 Detailbetrachtung – Selenium

Selenium zählt zu den bekanntesten und mächtigsten, quelloffenen Testwerkzeugen und verfügt über eine große Entwicklergemeinde. Es gliedert sich in drei Unterprojekte, welche jeweils verschiedene Aspekte des automatisierten Testens von Webanwendungen abdecken. Selenium IDE (IDE steht für Integrated Development Environment) ist sowohl ein Recorder als auch ein Editor zur Erstellung bzw. Bearbeitung Testskripten und ermöglicht die vereinfachte Ausführung derer. Selenium RC (Remote Control) unterstützt vor allem die browserübergreifende Ausführung der Testskripte und bietet maximale Flexibilität, durch eine große Auswahl an verschiedenen Beschreibungssprachen. Das dritte Unterprojekt heißt Selenium Grid und sorgt für eine Performanzsteige-

[10] Mit XPath können Knoten/Elemente in XML-Dokumenten ausgewählt werden.

rung, indem es die parallele Ausführung der Testfälle ermöglicht. Auf diese drei Konzepte möchte ich im Folgenden eingehen und deren Zusammenhänge verdeutlichen.

Selenium IDE

Da Selenium IDE als Plug-In für den Mozilla Firefox entwickelt worden ist, könnte die Installation nicht einfacher sein. Ein Klick auf den Downloadlink genügt und den Rest übernimmt der Browser. Anschließend können mit dem Werkzeug (siehe Abbildung 7) die ersten Testskripte erstellt werden, indem (ähnlich wie im Recorder von Canoo WebTest) die Nutzeraktionen beim Navigieren durch die zu testende Webanwendung in entsprechende Skriptbefehle umgewandelt werden.

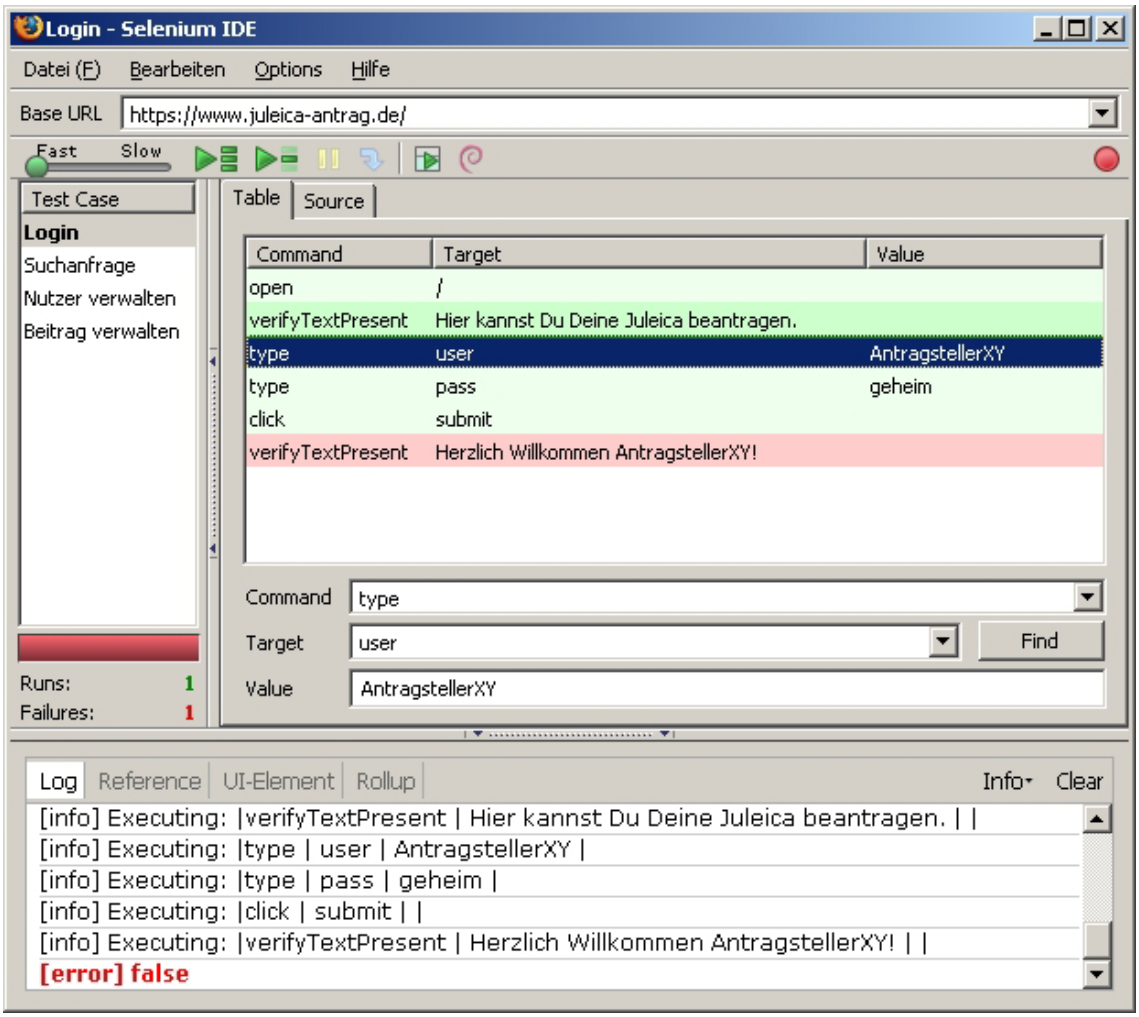


Abbildung 7: Selenium IDE – Die grafische Benutzeroberfläche

Diese Befehle werden tabellarisch aufgelistet und zu einem Testfall zusammengefasst. Da es sich aber um eine integrierte Entwicklungsumgebung handelt, können diese zusätzlich im Browserfenster editiert, ausgeführt und, dank einer umfassenden Logdatei

sowie dem Setzen von Breakpoints[11], debuggt werden. Das Abspielen dieser Testfälle erfolgt über den Browser, wobei man die Testgeschwindigkeit seinem Wünschen entsprechend anpassen kann. Dabei werden nicht nur statische Seiten inklusive sämtlicher Links überprüft, sondern auch dynamische Inhalte, wie z.B. JavaScript- und AJAX-Funktionen. Trotz des großen Funktionsumfangs unterliegt das Werkzeug jedoch noch kleinen Einschränkungen. Zum Einen können die Tests lediglich auf dem Firefox Browser gestartet werden und zum Anderen ermöglicht die Beschreibung der Testfälle weder Bedingungen noch Schleifen. Diesen Missstand behebt Selenium RC.

Selenium Remote Control

Die Architektur von Selenium RC besteht im Grunde aus drei Komponenten. Die Erste ist der Client-Treiber, der die auszuführenden Testschritte besitzt und diese an den Selenium Server (die zweite Architekturkomponente) weiterleitet. Der Server fungiert als Proxy für HTTP-Requests und wandelt diese Testschritte in JavaScript-Code um. Dieser wird letztlich an den Selenium Core (die dritte Architekturkomponente) weitergeleitet, welcher Komponenten des Internet Explorer, Safari und Firefox-Browsers enthält. Diese interpretieren wiederum den JavaScript-Code und führen damit den Test durch [Selenium1].

Der Server kann sofort nach dem Download mit einer Java-Anweisung gestartet werden, während der jeweilige programmiersprachenspezifische Client-Treiber vergleichsweise mühsam in eine entsprechende Entwicklungsumgebung integriert werden muss. Erst dann können die Skripte gestartet und die Tests durchlaufen werden. Hat man mit der Selenium IDE bereits Testfälle erstellt, so kann man diese für den Betrieb mit Selenium RC exportieren und anschließend mit einem normalen Editor bearbeiten bzw. verfeinern. Dabei kann der Nutzer sich zwischen Java, C#, Perl, PHP, Ruby oder Python, als Beschreibungssprache entscheiden. Durch Nutzung der üblichen Programmiertechniken (Iterationen und Bedingungen) können die Testfälle effizienter beschrieben werden und sind durch den Einsatz von Kommentaren besser lesbar als in der Tabellenstruktur der Selenium IDE. Weiterhin werden Datenbanktests, Exception Handling und benutzerdefinierte Testreports ermöglicht. Kurz gesagt: mit Selenium RC ist theoretisch alles möglich, was die verwendete Skriptsprache hergibt.

[11] Der Testdurchlauf kann durch das vorherige Markieren einzelner Testschritte an diesen Stellen pausiert werden, um bspw. den Status der Anwendung und einzelne Variablen kontrollieren zu können. Diese Markierungen nennt man Breakpoints.

Für die Ausführung sehr umfangreicher Tests kann die Ausführungsdauer zu einem Knackpunkt werden. Die Verbindung zwischen dem Selenium Server und dem Browser stellt den Flaschenhals des Systems dar. Eine parallele Ausführung mehrerer Testskripte ist auch nur beschränkt möglich. Um die Performanz des Systems auf ein höheres Level zu heben, wurde Selenium Grid ins Leben gerufen. Dieses steuert über den sogenannten Selenium Hub mehrere Selenium RCs und ermöglicht somit die Parallelität.

Eigene Erfahrungen

Eine umfassende Installationsbeschreibung des Testwerkzeugs ist online verfügbar und ist in wenigen Minuten umsetzbar. Durch Verwendung der Selenium IDE können einfache Testszenarien ohne Voraussetzungen von Programmierkenntnissen schnell durchgeführt werden, die für einen effizienten Einsatz mit Selenium RC allerdings notwendig werden. Wie auch beim Canoo WebTest sind Erfahrungen im Umgang mit XPath für die Beschreibung der Interaktionselemente einer Webanwendung von Vorteil. Während der Testphasen traten Probleme nur sehr selten auf, weshalb das Testwerkzeug einen sehr stabilen Eindruck hinterlässt.

4.3 Vergleich

Der folgende Vergleich hat genau zwei Motivationen. Zum Einen soll die Funktionsvielfalt der aktuellen Web-Testwerkzeuge dargestellt werden. Zum Anderen soll die Frage geklärt werden, ob (und falls ja, inwiefern) die quelloffenen Werkzeuge den kommerziellen Werkzeugen funktionstechnisch nachstehen. Auf eine Platzierung wird bewusst verzichtet, weil dafür ein Vergleich der Features allein nicht ausreichend Informationen liefern kann, sondern umfangreiche Erfahrungen in der Anwendung der teilweise sehr komplexen Testwerkzeuge nötig wären.

4.3.1 Vergleichskriterien

Ich habe mich zur Aufstellung dieser Kriterien auf die Punkte konzentriert, die man bei der Beschaffung eines automatisierbaren Testwerkzeugs für Webanwendungen in Betracht ziehen würde. Das bedeutet allerdings nicht, dass damit alle wesentlichen Faktoren abgedeckt sind. Sie sind eher als eine subjektive Auswahl anzusehen. Die Tabelle 2 stellt nur eine kurze Übersicht der verwendeten Kriterien dar. Eine detailliertere Erklärung erfolgt im nächsten Abschnitt.

Tabelle 2: Kriterienüberblick für den Testwerkzeugvergleich

Nr.	Vergleichskriterium	Fragestellung
1	Capture and Replay	Wird eine GUI-basierte, vereinfachte Form der Skripterstellung angeboten?
2	Testfallbeschreibung	Welche Skriptsprachen werden unterstützt?
3	Internettechnologien	Welche Technologien können getestet werden?
4	Browserunterstützung	Welche Browser werden unterstützt?
5	Keyword Driven Testing	Ist eine Schlüsselwort-getriebene Strukturierung der Testfälle möglich?
6	Data Driven Testing	Findet eine Trennung der Testdaten von den Testabläufen statt?
7	Berichterstattung	Wie erfolgt die Benachrichtigung über das Testergebnis?
8	Multitasking	Können mehrere Testfälle parallel durchgeführt werden?
9	Aktualität	Wird das Werkzeug weiterentwickelt?
10	Preis/Lizenz	Wie hoch sind die Anschaffungskosten bei kommerziellen Testwerkzeugen? / Unter welchen Bedingungen darf ich Veränderungen an quelloffenen Testwerkzeugen vornehmen?
11	Support	Welche Hilfestellungen werden vom Hersteller geboten?

4.3.2 Vergleich aktueller Testwerkzeuge

Aus Gründen der Übersichtlichkeit wird in diesem Abschnitt nicht für jedes einzelne Feature ein extra Verweis zur Onlinequelle angegeben. Die für diesen Vergleich genutzten Quellen sind im Literaturverzeichnis speziell markiert. So bezieht sich z.B. die Quellenangabe mit dem Kürzel [VaT-Sele1] auf die genutzte Online-Quelle, wobei VaT (Vergleich aktueller Testwerkzeuge) für diesen Abschnitt und Sele für das betrachtete Web-Testwerkzeug (hier Selenium) steht. Dies schließt jedoch nicht aus, dass diese Quellen auch in anderen Abschnitten der Arbeit genutzt werden können. Die jeweiligen

Tabellen mit den entsprechenden Vergleichskriterien stellen die Ergebnisse der Untersuchungen übersichtlich dar.

1. Capture and Replay

„Capture and Replay“ beschreibt eine vereinfachte Form zur Erstellung der Testskripte. Dabei navigiert der Nutzer wie gewohnt durch die zu testende Webanwendung, während das Testwerkzeug alle relevanten Nutzeraktionen, wie z.B. Mausklicks oder Tastatureingaben, abspeichert und diese in ein Testskript umwandelt. Im Idealfall sind keine Programmierkenntnisse notwendig, häufig müssen jedoch am Skript Nachbesserungen per Hand vorgenommen werden.

Tabelle 3: Vergleichskriterium – Capture and Replay

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
+	+	+	+	+	+	+	–	–	+

Bis auf Watir und WebInject verfügen alle Testwerkzeugen über diese bequeme Möglichkeit der Testfallerstellung. Paul Rogers (einer der Watir-Hauptentwickler) begründet diesen Missstand als Anreiz für Testentwickler, um herauszufinden wie Watir mit Webanwendungen interagiert. Zitat: *„If you think a script recorder is going to do the testing for you, please think again. And again“*, vgl. [VaT-Wati2].

2. Testfallbeschreibung

Die meisten Testwerkzeuge nutzen XML oder Skriptsprachen wie bspw. Groovy, JavaScript oder Perl zur Beschreibung der Testfälle. Zusätzlich werden häufig baumartige Strukturen oder tabellenförmige Darstellungen der Abläufe angeboten. Beide Varianten haben ihre Vor- und Nachteile. Während Skriptsprachen im Allgemeinen viel mehr Freiheiten zur Gestaltung der Testfälle bieten, setzen Abläufe in Tabellenform keine Programmierkenntnisse voraus. Im schlimmsten Fall allerdings bietet ein Testwerkzeughersteller nur seine eigene proprietäre Skriptsprache, die sich der Tester erst einmal aneignen muss.

Tabelle 4: Vergleichskriterium – Testfallbeschreibung[12]

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
ASP.NET, C#, C++, Excel, Java, JS, Perl, PHP, Python, Ruby, TCL, VBS	Java, Ruby, Groovy, C#	Jython	Python, JS, Perl, TCL	Jython, Groovy	JS	C#, Java, Perl, PHP, Python, Ruby, HTML	Java, Ruby	XML	Groovy, XML

Alle Testwerkzeuge verwenden zur Beschreibung der Testfälle aktuelle, nicht-proprietäre Programmiersprachen, jedoch bieten manche dem Nutzer nur wenig oder gar keine Auswahl. Die größte Auswahl bietet iMacros. Es kann laut Hersteller jede Sprache genutzt werden, welche auch Microsofts COM-Schnittstelle unterstützt. Und da dies fast alle aktuellen Programmiersprachen tun, bleibt dem Nutzer nur noch die Qual der Wahl.

3. Internettechnologien

Bereits bei der Auswahl wurde darauf geachtet, dass die Werkzeuge das Testen der Standardtechnologien HTML und JavaScript ermöglichen. Welche darüber hinaus Möglichkeiten zum Testen erweiterter Technologien (bspw. Java Applets oder Flash) bieten, wird in der folgenden Tabelle übersichtlich dargestellt.

Tabelle 5: Vergleichskriterium – Internettechnologien

iMac	Liqu	QEng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
ActiveX*, Flash, Java Applets	Java Applets	ActiveX, Flash, Java Applets	–	–	–	Flash	Flash	–	–

* Es handelt sich um eine Annahme.

Hier triumphieren ganz klar die kommerziellen Testwerkzeuge. iOpus (Hersteller von iMacros) bewirbt sein Produkt damit, alles, was im Browser sichtbar und klickbar ist, automatisieren zu können. Dies konnte jedoch am Beispiel der Patty-Webanwendung (vgl. Kapitel 5.2) wiederlegt werden, da die Anwendung auf der SVG-Technologie basiert, welche der mitgelieferte Browser leider nicht unterstützt. Mit der Selenium- und

[12] Abkürzungen: ASP.NET (Active Server Pages .NET), JS (JavaScript), TCL (Tool Command Language), VBS (Visual Basic Script)

Watir-Grundversion ist das Testen von Flash-Anwendungen nicht möglich. Erst durch das Nachinstallieren spezieller Erweiterungen kann diese Webtechnologie getestet werden.

4. Browserunterstützung

Jeder professionelle Webentwickler hat stets mehrere Browser auf seinem Entwicklungssystem installiert, um mögliche Abweichungen in der Darstellung der Webanwendungen aufzuspüren und zu beseitigen. Ebenso sollte jedes Testwerkzeug die gängigsten Webbrowser unterstützen.

Tabelle 6: Vergleichskriterium – Browserunterstützung[13]

iMac	Liqu	QEng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
FF2, FF3, IE6, IE7, IE8	FF2, IE6, IE7, IE8	FF2, IE6, IE7, Mozilla 1.7	+ *	FF2, FF3, IE6, IE7, IE8, Mozilla 1.8	+ *	Chrome, FF2, FF3, IE7, IE8, Opera8, Opera9, Safari2, Safari3	Chrome, FF2, FF3, IE6, IE7, Safari	+ *	FF2, FF3, IE6, IE7

* Diese Testwerkzeuge arbeiten browserunabhängig.

Internet Explorer und Firefox, die beiden Marktführer im Webbrowserbereich, werden von allen Kandidaten unterstützt. Die browserunabhängigen Testwerkzeuge operieren auf der HTTP-Protokoll-Ebene. Dies hat den Vorteil, dass nicht für jeden Browser extra eine Erweiterung entwickelt und gewartet werden muss. Der Nachteil liegt jedoch darin begründet, dass dadurch das exakte Nutzerverhalten nicht abgebildet werden kann. Dies können nur GUI-basierte Testwerkzeuge realisieren.

5. Keyword Driven Testing

Die Idee des „Keyword Driven Testing“ besteht darin, elementare Testschritte mit einem Schlagwort zu versehen und diese, zusammen mit anderen Schlagworten, zu einem Testfall höherer Ebene zu definieren. Dies ermöglicht die Wiederverwendung der Testskripte, erleichtert die Wartung und kann sogar für manuelles Testen eingesetzt werden. Der größte Vorteil resultiert jedoch aus der vereinfachten Testfallerstellung. Sind die elementarsten Schritte erst einmal mit einem Schlagwort markiert, können auch Mitarbeiter ohne Programmierkenntnisse komplizierte Testszenarien formulieren.

[13] Abkürzungen: FF (Firefox), IE (Internet Explorer)

Tabelle 7: Vergleichskriterium – Keyword Driven Testing

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
– *	+	+	+	– *	– *	– *	+ *	–	– *

* Es handelt sich um eine Annahme. Kriterium konnte weder bestätigt, noch widerlegt werden.

Da die meisten Werkzeuge nicht explizit mit diesem Feature beworben wurden, musste ich mehrere Annahmen treffen. Ich achtete dabei besonders auf die Möglichkeiten der verwendeten Skriptsprachen und die technische Umsetzung der Testwerkzeuge, sofern dies möglich war. Somit sind die mit einem Stern markierten Symbole als subjektive Einschätzung zu bewerten.

6. Data Driven Testing

Ein Testfall kann durch eine externe Datenquelle mit Eingabedaten versorgt werden, sodass derselbe Test mit unterschiedlichen Daten durchlaufen werden kann, ohne dabei das Skript anpassen zu müssen. Diese Trennung der Testdaten vom Testablauf ermöglicht insbesondere eine unabhängige Pflege der Testdaten, bei der keine Programmierkenntnisse erforderlich sind.

Tabelle 8: Vergleichskriterium – Data Driven Testing

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
+	+	+	+	+	+	+	+	–	+

Bis auf WebInject verfügen alle untersuchten Web-Testwerkzeuge über dieses Feature.

7. Berichterstattung

Die Übersichtlichkeit der Berichte ist hierbei von größter Bedeutung. Einfache Log-Ausgaben eignen sich im Allgemeinen nicht zur Berichterstattung. Sie beinhalten nicht nur für den Tester unwichtige Details, sondern sind auch kaum optisch aufbereitet, sodass sich das Lesen eher als anstrengend erweist. Idealerweise bestehen die Berichte aus übersichtlich-gestalteten HTML-Dateien, die man sich mühelos abspeichern und archivieren kann. Eine Datenexportunterstützung über CSV-Dateien oder einer Datenbankschnittstelle ließen für statistische Zwecke keine Wünsche mehr offen.

Tabelle 9: Vergleichskriterium – Berichterstattung

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
Log	Log (Junit)	HTML	HTML, Log	HTML, Log	Log	HTML	Log	HTML, XML	HTML

HTML-basierte Berichte sind die weitverbreitetste Form der Ergebnispräsentation. WebInject bietet eine XML-basierte Möglichkeit zum Export der Testergebnisse an.

8. Multitasking

Da sich im Laufe der Zeit (je nach Umfang der zu testenden Webanwendung) eine Vielzahl von Testfällen anhäufen, kann ein kompletter, sequentieller Durchlauf sogar mehrere Stunden in Anspruch nehmen. Falls das verwendete Werkzeug die Möglichkeit bietet, Testskripte parallel abzuarbeiten, könnte daraus eine enorme Zeitersparnis resultieren.

Tabelle 10: Vergleichskriterium – Multitasking

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
– *	+	– *	– *	+	+	+	+	– *	+

* Es handelt sich um eine Annahme. Kriterium konnte weder bestätigt, noch widerlegt werden.

Bei den markierten Werkzeugen ist es mir nicht möglich gewesen zu entscheiden, ob diese parallele Testabläufe unterstützen. Daher habe das Feature bei diesen Testwerkzeugen vorsichtshalber als nicht vorhanden markiert.

9. Aktualität

Von besonderem Interesse ist ebenfalls die Aktualität des Produkts. Eine Software die nicht mehr gepflegt bzw. weiterentwickelt wird stirbt und findet weder Akzeptanz, noch Käufer. Dazu wurden die Releasedaten der letzten Softwareupdates und die zugehörige Versionsnummer betrachtet, um entscheiden zu können, ob das Produkt bereits die Marktreife erreicht hat.

Tabelle 11: Vergleichskriterium – Aktualität[14]

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
10. Jun 2009 V 6.60	04. Jun 2009 V 1.0.8	Keine An-gabe V 6.9	15. Apr 2009 V 3.4.3	06. Mai 2009 V 1.2.1	21. Mai 2009 V 2.0	10. Jun 2009 V 1.0.1	06. Nov 2008 V 1.6.2	04. Jan 2006 V 1.41	05. Mrz 2009 V 3.0

Ganz besonders sticht hier WebInject hervor, deren letztes Update über drei Jahre zurückliegt. Man könnte vermuten, dass die Entwickler das Projekt erfolgreich abgeschlossen haben. Viel wahrscheinlicher ist jedoch, dass die Weiterentwicklung eingestellt worden ist. Der Zeitpunkt für die Veröffentlichung der QEngine-6.9-Version konnte nicht festgestellt werden. Weder in den Release Notes, noch im News-Bereich der offiziellen Produktwebseite konnten Informationen darüber gefunden werden.

10. Preis/Lizenz

Eines der Hauptfaktoren zur Beschaffung kommerzieller Software ist neben dem Funktionsumfang ohne Frage der Preis. Da aber nicht alle untersuchten Werkzeuge dieselben Bedingungen haben, soll zumindest die Größenordnung der Preise dargestellt werden, in der diese gehandelt werden. Die angegebenen Kosten entsprechen einer Einzelnutzerlizenz der umfangreichsten Produktedition. Für Open Source Produkte spielt dies keine Rolle. Hier ist u.a. der Lizenztyp interessant, der z.B. angibt, unter welchen Bedingungen die Software geändert werden darf.

Tabelle 12: Vergleichskriterium – Preis/Lizenz

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
355,- EUR* (499,- USD)	708,- EUR* (995,- USD)	1710,- EUR* (2394,- USD)	2400,- EUR	1595,- EUR	Apache License 2.0	Apache License 2.0	BSD-License	GNU General Public License	Apache License 2.0

* Die Werte sind gerundet und entsprechen dem offiziellen Wechselkurs vom 11.06.2009.

Angesichts der Preisunterschiede (bis zu 2000,- EUR pro Lizenz) sollte man sich viel Zeit zum Studieren der eigenen Anforderungen nehmen, um möglichst genau abschätzen zu können, wie umfangreich das kostenpflichtige Support-Packet ausfallen muss.

[14] Stand der Untersuchung: 12.06.2009

Bei den quelloffenen Testwerkzeugen kann man in diesem Punkt kaum etwas falsch machen. Alle erlauben die veränderte Weitergabe der Software und mit Ausnahme der GNU General Public License können diese Produkte unter jeder beliebigen Drittlizenz verbreitet werden.

11. Support

Zum Abschluss wird geprüft, ob die untersuchten Programme über eine umfangreiche Online-Hilfe, wie bspw. einem Handbuch, einer Dokumentation oder einem Wiki, verfügen, sodass sich der Nutzer bei Unklarheiten möglichst schnell und unkompliziert selbst behelfen kann.

Tabelle 13: Vergleichskriterium – Support

iMac	Liqu	Qeng	Squi	Web2	Sahi	Sele	Wati	WebI	WebT
+	+	+	+	+	+	+	+	+	+

Dieses Kriterium erfüllen alle Kandidaten, was auch darauf zurückzuführen ist, dass bei der Auswahl der Kandidaten auf dieses Kriterium besonders geachtet wurde. Zusätzlich bieten die Hersteller (und Drittanbieter für quelloffene Testwerkzeuge) kostenpflichtige Schulungen, Installationsservices, 24-Stunden Hotlines oder Nutzerforen an, um Supportfragen zwischen Entwicklern und Community zu klären.

4.3.3 Fazit

Die kommerziellen Testwerkzeuge heben sich im Speziellen durch eine größere Auswahl der unterstützten Internettechnologien ab. Weiterhin bieten die Hersteller dieser Werkzeuge eine größere Vielfalt an Supportmöglichkeiten an, welche allerdings in den meisten Fällen mit zusätzlichen Kosten verbunden sind. Dennoch stehen ihnen die quelloffenen Werkzeuge funktionstechnisch kaum nach, da bspw. hinter den Testwerkzeugen Watir und Selenium große Entwicklergemeinden stehen, die für eine kontinuierliche Weiterentwicklung sorgen. Deshalb sehe ich die Anschaffungskosten für QEngine, Web2Test und insbesondere Squish for Web als wenig rentabel an. Dass es bei gleichem Produktumfang auch preiswerter geht, hat iMacros gezeigt. Nach meinen Recherchen hatte ich den Eindruck, dass Sahi und Canoo's WebTest in der Praxis kaum wahrgenommen und von Selenium sowie Watir eher überschattet werden. Dabei könnten diese aufgrund ihrer verwendeten Technologien insbesondere für Java-Entwickler

sehr interessant sein. Von WebInject würde ich eher abraten, vor allem weil es die heutigen Anforderungen nicht mehr erfüllen kann und die Weiterentwicklung anscheinend eingestellt wurde.

Wenn man die Weiterentwicklung des Testens bei klassischen Programmiersprachen betrachtet (vom manuellen Testfall bis zur testgetriebenen Entwicklung mit Werkzeugunterstützung), könnte man davon einen Trend für die Entwicklung zukünftiger Web-Testwerkzeuge ableiten, bei dem diese viel stärker mit dem eigentlichen Entwicklungsprozess (der Programmierung) verzahnt werden.

5 Fallstudien

In diesem Kapitel werden die Ergebnisse der bearbeiteten Fallstudien dargestellt und diskutiert. Dabei handelt es sich zum Einen um eine Aufwandsanalyse des Juleica-Testsystems, das von der epion GmbH entwickelt worden ist und zum Anderen um eine Machbarkeitsstudie zur Erstellung eines automatisierbaren Testsystems für das Webprojekt Patty vom Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit.

5.1 Das Juleica-Datenbank Projekt

Sowohl zur Einführung, als auch zur Ermittlung benötigter Daten wurde der komplette Einrichtungsprozess des Juleica-Autotestsystems durchlaufen und die dafür benötigten Aufwände in Zeiteinheiten gemessen. Weiterhin konnten durch die Erstellung der Testskripte wichtige Erfahrungen im Umgang mit Testwerkzeugen gewonnen werden, die zur Bearbeitung der Fallstudien hilfreich waren. Zunächst wird eine kleine Einführung in die Webanwendung und das Testsystem vorgenommen. Anschließend werden die Aufwände eines manuellen Testprozesses mit denen eines automatisierten Testprozesses verglichen und kritische Kostenfaktoren genauer betrachtet.

5.1.1 Projektbeschreibung

Die Jugendleiter/In-Card (Kurzform: Juleica) ist ein bundesweit-einheitlicher Ausweis für Personen, die ehrenamtlich in der Jugendarbeit tätig sind. Dies umfasst bspw. sportliche Aktivitäten, Feriengestaltung, Weiterbildung oder die Vorbereitung auf gesellschaftliche Verantwortung. Da jeder Karteninhaber eine Ausbildung nach festgeschriebenen Standards absolviert hat, dient die Karte sowohl als Qualifikationsnachweis zur Betreuung von Kindern und Jugendlichen, als auch zur Legitimation gegenüber öffentlichen Stellen, wie z.B. Jugendeinrichtungen, Konsulaten oder der Polizei. Zusätzlich soll diese die gesellschaftliche Anerkennung für das ehrenamtliche Engagement zum Ausdruck bringen. Deshalb erhalten die Kartenbesitzer oftmals Vergünstigungen, wie z.B. Preisnachlässe im Kino oder freien Eintritt ins Schwimmbad. Für den Erwerb der Karte gibt es allerdings einige Voraussetzungen. So muss ein Antragsteller mindestens 16 Jahre alt sein, über Erste-Hilfe-Kenntnisse verfügen und eine Ausbildung nach bestimmten Richtlinien absolviert haben. Weiterhin muss ein dauerhaftes Engagement bei einem Träger der Jugendarbeit (z.B. Jugendverband, Jugendring, Jugendamt) bestehen [Juleica1].

Sind diese Voraussetzungen erfüllt, so kann der Antrag gestellt werden, der allerdings alle zwei Jahre erneuert werden muss. Dazu musste das Formular bisher bei einem Träger bezogen und handschriftlich-ausgefüllt dem Jugendamt zuschicken werden. Zwischen der Antragstellung und dem Erhalt der Karte war eine Wartezeit von mehreren Wochen durchaus üblich, welche auf die langen Informationswege zurückzuführen sind. Der Deutsche Bundesjugendring stellte im Juni 2007 finanzielle Mittel für das Projekt „Weiterentwicklung Juleica“ bereit, welches u.a. zur Attraktivitätssteigerung der Jugendleiter/In-Card führen soll. So wurde neben einem neuen Kartendesign (Abbildung 8, rechts) vor allem die medienbruchfreie Antragstellung über das Internet realisiert, das im Vergleich zum bisherigen Verfahren eine kostengünstigere und schnellere Abwicklung ermöglicht.

Seit 1999 Seit 2009

Abbildung 8: Die Jugendleiter/In-Card, vgl. [Juleica2]

Das Online-Verfahren der Juleica-Antragstellung

Die Webanwendung ist seit April 2009 vorerst als Pilotprojekt für die Regionen Berlin und Niedersachsen im Einsatz. Ab Juli 2009 soll das Verfahren dann planmäßig jedem Bundesland zur Verfügung stehen. Der Zugriff auf das Online-Antragsverfahren kann entweder über die offizielle Juleica-Webseite (siehe Abbildung 9) oder direkt unter www.juleica-antrag.de erfolgen. Um einen Onlineantrag stellen zu können, muss sich zunächst, unter Verwendung einer gültigen E-Mail-Adresse, registriert werden. Mit dem zugeschickten Passwort kann der/die Antragsteller/In sich anschließend in das System einloggen und das Antragsformular online ausfüllen. Neben der Angabe der persönlichen Daten und der träger-bezogenen Daten, wird ebenfalls ein digitales Portraitbild benötigt. Zusätzlich können einige Fragen für statistische Zwecke beantwortet werden, deren Auswertung anonym erfolgt. Ist das Formular abgeschickt worden, so wird der/die Antragsteller/In über den Bearbeitungsstatus per E-Mail informiert. Nun kann

Abbildung 9: Die Antragsverfahren der Jugendleiter/In-Card, vgl. [Juleica3]

der Antrag von der zuständigen Prüfungsinstanz entweder abgelehnt (bzw. unter Auflagen vorerst abgelehnt) oder bestätigt werden. Letzteres veranlasst die Beauftragung der Druckerei zur Erstellung der Karte, welche dem/der Antragsteller/In letztendlich zugesandt wird. Das vorhandene Benutzerkonto kann für spätere Kartenverlängerungsanträge wiederverwendet werden.

Vorstellung des Testsystems

Als Werkzeug dient WebTest von der Firma Canoo, das im Kapitel 4.2.3 detailliert betrachtet wurde. Die zu testende Anwendung entspricht jedoch nicht der eigentlichen Anwendung. Es sind nämlich kleine Veränderungen daran vorgenommen worden, die das automatisierte Testen erst ermöglicht haben. So musste bspw. die Captcha-Abfrage, die ein automatisiertes Erstellen von Nutzerkonten ursprünglich verhindern sollte, deaktiviert werden, um genau dieses Verhalten zulassen zu können. Weiterhin werden Test-relevante Daten, die dem Nutzer im Allgemeinen verborgen wären, an das Ende einer jeden HTML-Seite aufgeführt. Den Startpunkt des Testsystems stellt die Entwicklungsumgebung Eclipse dar. Damit können die Testfälle komfortabel erstellt und anschließend ausgeführt werden. Der Testprozess kann aber ebenso über einen Kommandozeilenbefehl erfolgen. Um Abhängigkeiten unter den Testfällen zu vermeiden, ist die Webanwendung nach Bedarf auf den Ausgangszustand zurückzusetzen. Dazu genügt eine einfache XML-Anweisung, die an der entsprechenden Stelle im Testfall angegeben werden kann. Des Weiteren wird innerhalb eines jeden Testfalls eine entsprechende Beschreibung des jeweiligen Testschritts angegeben, die zur Ausführungszeit in die Log-Datei geschrieben wird. Extrahiert man diese Abschnitte aus der Log-Datei, so erhält man eine exakte Dokumentation der durchgeführten Tests, die auch von Personen interpretiert werden können, die nicht über Programmierkenntnisse verfügen. Das Testsystem ist somit fertig eingerichtet und voll funktionsfähig. Zur Weiterentwicklung könnten lediglich Code-Optimierungen vorgenommen werden, um bspw. redundante Anweisungen zu vermeiden oder kürzere Einarbeitungszeiten zu erreichen. Der größte Entwicklungsbedarf besteht jedoch in der Definition ausreichender Testfälle, die mit dem Testsystem abgearbeitet werden können [epion].

5.1.2 Aufwandsanalyse für Juleica-Autotest

Anhand der Juleica-Webanwendung werden die entsprechenden Aufwände zwischen dem manuellen und dem automatisierten Testprozess verglichen. Ein Großteil der Daten konnte praktisch ermittelt werden. Da eine Betrachtung des gesamten Testprozesses im Rahmen einer Masterarbeit zu umfangreich wäre, wird sich nur auf einen Teil der Testprozesse bezogen. Mit der Auswertung der Ergebnisse wird diese Fallstudie abgeschlossen.

Da die Testprozesse ebenfalls von der Organisationsform eines Entwicklerteams abhängig sind, wird sich für folgende Betrachtungen ausschließlich auf die Organisations-

form bezogen, bei der Entwickler und Tester zwei verschiedene Personen sind. Speziell in kleineren Unternehmen werden diese Aufgabenbereiche oftmals von derselben Person übernommen, was in den meisten Fällen auf ein stark begrenztes Personal zurückzuführen ist. Dies ist im Allgemeinen nicht zu empfehlen, da der Entwickler während der Programmierung ein bestimmtes Schema für seinen Code entwickelt, das er auch beim Testen im Gedächtnis behält und somit gewisse Nebenbedingungen und Abhängigkeiten einfach übersieht, die ein separater Tester mühelos erkannt hätte. Der Entwickler geht mit der Absicht heran die Korrektheit seines Programms beweisen zu wollen, während die Tester die Absicht haben Programmfehler zu provozieren. Weiterhin besteht bei Entwicklern eine psychische Blockade, da diese stets bemüht sind fehlerfreie Programme zu schreiben und deshalb durch das Testen ihrer Erzeugnisse nur ungern die eigenen Fehler aufdecken.

Darstellung des manuellen und automatisierten Testprozesses

Die einzelnen Prozessschritte sind in Abbildung 10 zusammenfassend dargestellt. Der manuelle Testprozess beginnt mit der Einarbeitung in die zu testende Webanwendung, in dem das Pflichtenheft gesichtet und Funktionen der Anwendung ausprobiert werden. Hat man die Grundzüge der Anwendung (bzw. des Moduls) verstanden, so entwickelt man im Allgemeinen ein bestimmtes Testverständnis, das für den nächsten Schritt – die Erstellung der Testfälle – notwendig ist. Dabei werden die Testfälle als textuelle Ablaufbeschreibungen (inkl. dem Anfangszustand, den Eingaben, die zu erwartenden Ausgaben und dem Endzustand) formuliert, die jeder Tester verstehen und in einem Testdurchlauf umsetzen kann. Den letzten Prozessschritt bildet die Testfallnachbereitung. Ist im Testdurchlauf ein Fehler erkannt worden, so muss zunächst geprüft wer-

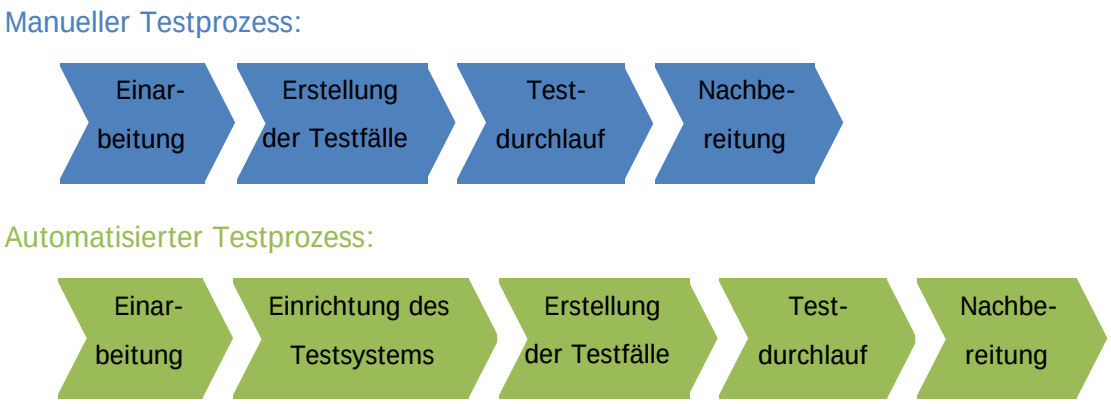


Abbildung 10: Darstellung der Testprozesse

den, ob sich der Fehler wirklich in der Anwendung befindet oder auf eine fehlerhafte Testfallbeschreibung zurückzuführen ist. Dies kann durch die Sichtung des Pflichtenhefts geprüft werden. Erst wenn die Ursache des Fehlers im Testfall auszuschließen ist, kann dieser dem zuständigen Entwickler in Form eines übersichtlichen Protokolls übermittelt werden. Das Fehlerprotokoll sollte die betreffende Funktion, deren getätigten Eingaben, die zu erwartenden Ausgaben sowie die tatsächlichen Ausgaben beinhalten. Sobald Änderungen am Programm vorgenommen wurden, die eine Anpassung der Dokumentation bzw. des Pflichtenhefts erfordern, müssen ebenfalls die betroffenen Testfälle angepasst werden.

Der automatisierte Testprozess unterscheidet sich vom manuellen Testprozess grundlegend nur in der zusätzlichen Einrichtung des Testsystems, welches aus der zu testenden Webanwendung und dem Testwerkzeug besteht. Die Testfälle werden statt einer textuellen Beschreibung in der für das Testwerkzeug relevanten Syntax formuliert. Zusätzlich müssen den Anweisungen Kommentare hinzugefügt werden, um anderen Mitarbeitern die Einarbeitung zu erleichtern.

Darstellung der ermittelten Aufwände

Eine Auflistung der praktisch ermittelten Aufwände ist in Tabelle 14 dargestellt. Für die Projekteinarbeitung wurden 3,25 Stunden benötigt. Dieser Aufwand ist sowohl für den manuellen, als auch für den automatisierten Testprozess identisch. Für die Einrichtung des Testsystems ergab sich ein Gesamtaufwand von 28,25 Stunden. Dieser beinhaltet die Kosten für die Einarbeitung in das Testwerkzeug und die eigentliche Einrichtung anhand einer vorhandenen Installationsanleitung, die von den Entwicklern der Firma epion erstellt worden war. Mehr als die Hälfte des Aufwands wurde in die experimentelle Erstellung von Testskripten investiert. Da es hierbei zu einigen Problemen mit der Testfallbeschreibungssyntax kam, mussten weiterführende Recherchen betrieben werden. Für die Erstellung von vier manuellen Testfällen wurde ein Gesamtaufwand von genau zwei Stunden benötigt. Dieser entspricht ungefähr einem Fünftel des Aufwands zur Erstellung der entsprechenden, automatisierten Testskripte (ca. 9,5 Stunden). Um sich eine Vorstellung vom Umfang der Testskripte machen zu können, wurden zwei Testfälle im Anhang der Arbeit angefügt. Der Beginn der Testskripterstellung erfolgte nach einer ausgiebigen Einarbeitung, um Lerneffekte zu nutzen und somit konsistentere Messdaten zu erhalten. Da eine Testfallabdeckung der gesamten Juleica-Webanwendung selbst im Rahmen einer Masterarbeit zu umfangreich wäre, habe ich mich auf die Erstellung von vier Testfällen beschränkt, die eine repräsentative Grund-

komplexität darstellen. Damit kann eine Hochrechnung vorgenommen werden, die sich allerdings auf zwei Annahmen stützt. Zum Einen wird angenommen, dass bei einer Anzahl von 30 Testfällen der Funktionsumfang der Juleica-Webanwendung weitestgehend abgedeckt werden kann und zum Anderen die Komplexität dieser 30 Testfälle gleichverteilt ist. Diese Annahmen konnten nach Rücksprache mit den Mitarbeitern der Firma epion bekräftigt werden. Für eine manuelle Durchführung dieser vier Testfälle konnte ein Gesamtaufwand von 12,5 Minuten gemessen werden. Dieser Vorgang beinhaltet die Sichtung der Testfallbeschreibung und die anschließende Durchführung dieser Anweisungen. Pro Testfall wurden im Durchschnitt etwa 3 Minuten benötigt. Bei der automatisierten Testdurchführung liegt der Durchschnitt etwa bei 21 Sekunden. Für jeden Durchlauf sind ca. 30 Sekunden zu addieren, die für die Initialisierung des Testwerkzeugs und die Erstellung des Testberichts aufgewendet werden. Dies führt bei einer Anzahl von vier Testfällen zu einer Durchlaufzeit von 1,93 Minuten. Da eine Datenerhebung für die Aufwände der Nachbereitungsphase (inkl. einer möglichen sich anschließenden Testfallanpassung) zu komplex und zu stark von auftretenden Fehlern abhängig ist, welche man vorher weder bestimmen, noch abschätzen kann, entfällt dieser Prozessschritt für die vorliegende Analyse.

Tabelle 14: Juleica-Autotest – Die ermittelten Testprozessaufwände

Prozessschritt	Aufwand (min.)	
	Manuell	Automatisch
Projekteinarbeitung	195	195
Einrichtung des Testsystems	---	1695
Erstellung von **vier** Testfällen	120	571
Testdurchlauf der **vier** Testfälle	12,5	1,9
Nachbereitung	Datenerhebung nicht möglich	

Fehlerbetrachtung

Generell können die ermittelten Aufwände aller Prozessschritte (mit Ausnahme der automatisierten Testdurchführung) aufgrund unterschiedlicher Vorkenntnisse und persönlicher Eigenschaften des jeweiligen Testers variieren. Weiterhin wäre bei einer Neukonzipierung des Testsystems mit weitaus höheren Aufwänden zu rechnen. Die ermittelten Daten beziehen sich auf ein existierendes Testsystem, das anhand geleisteter Vorarbeit (Leitfaden zur Einrichtung des Testsystems) selbstständig eingerichtet worden ist. Bei der Erstellung der Testfälle können sich zusätzlich Lerneffekte ergeben, die

zu einer effizienteren Umsetzung führen können. Des Weiteren ist unklar, ob sich die getroffenen Annahmen (Anzahl von 30 Testfällen benötigt, Gleichverteilung der Komplexität dieser Testfälle) in der Realität bestätigen lassen. Der Aufwand der Nachbereitung ist schwierig abzugrenzen. Generell ist beim Auftreten von Fehlern mit Folgeaufwänden zu rechnen, die durch den Ausschluss eines Fehlers im Testfall bzw. der Anpassung des Testfalls und einem erneuten Durchlauf entstehen.

Berechnung des Break-Even-Punktes

Der Break-Even-Punkt entspricht der Anzahl von Testdurchführungen, bei der die Gesamtaufwände der beiden Prozesse ausgeglichen sind. In Abbildung 11 kann man sehr deutlich erkennen, dass der manuelle Testprozess einen geringen Initialaufwand besitzt, der allerdings mit zunehmenden Testdurchführungen drastisch steigt. Die automatisierten Testprozesse besitzen durch die zusätzliche Erstellung des Testsystems einen sehr hohen Initialaufwand, der jedoch im Vergleich zum manuellen Testprozess bei zunehmenden Testdurchführungen nur noch minimal ansteigt. Der Break-Even-Punkt stellt sich auf Grundlage der gemessenen Daten sowie der getroffenen Annahmen und unter ohne Berücksichtigung der Testfallanpassungsaufwände bei 66 Testdurchführungen ein. Der orange-schraffierte Bereich stellt das Einsparungspotenzial jeder zusätzlichen Testdurchführung dar. Beim 76. Testdurchlauf ergibt sich somit eine Aufwandsersparnis von 10,2% und im 96. Testdurchlauf sogar 25,3%.

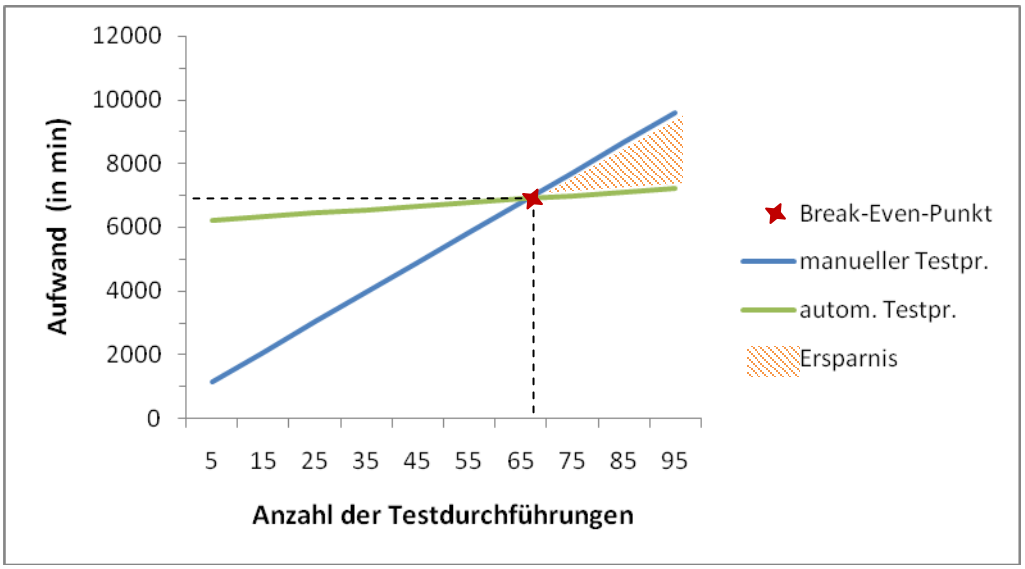


Abbildung 11: Juleica-Autotest – Darstellung des Break-Even-Punktes

Auswertung der Ergebnisse

Ein wesentlicher Kostenfaktor des automatisierten Testprozesses liegt in der Konzipierung und Umsetzung des Testsystems begründet, was von der Wahl des Werkzeugs, bis hin zur endgültigen Fertigstellung der Testumgebung mehrere Wochen Arbeitszeit in Anspruch nehmen kann. Der Konzipierungsaufwand könnte durch eine projektübergreifende Nutzung für ähnliche Webanwendungen stark reduziert werden bzw. komplett entfallen.

Die Definition der Testskripte stellt einen weiteren kritischen Kostenfaktor dar. Zum Einen sollte darauf geachtet werden redundante Beschreibungen zu vermeiden, da diese nicht nur erhöhten Entwicklungsaufwand bedeuten, sondern auch die Ausführungszeit der Testskripte unnötig verlängern. Eine Schlüsselwort-getriebene (keyword driven) Organisation der Testfälle wäre in dieser Hinsicht ein guter Ansatz, doch stellt dabei die dazu nötige Definition atomarer Testschritte erfahrungsgemäß die größte Schwierigkeit dar. Zum Anderen sollte man darauf achten, dass nicht zu detailliert getestet wird. Bspw. ist es für die Antragstellung der Juleica-Webanwendungn wenig hilfreich automatisierte Testfälle zu schreiben, bei denen das Vorhandensein der einzelnen Formularelemente geprüft wird, da die Anwendung keine Verhaltensmuster aufweist, bei denen Formularelemente verschwinden könnten. Es sollte also versucht werden Testfälle für die Teilbereiche zu definieren, bei denen man auch Fehlverhalten erwarten könnte. Weiterhin könnte es wirtschaftlicher sein, ausschließlich für die Kernfunktionen der Webanwendung Testskripte zu erstellen und die seltener ausgeführten Nebenfunktionen bei Bedarf manuell zu testen. Die Hauptfunktionen könnten entweder über bereits vorhandene Erfahrungswerte, Nutzerverhaltensstatistiken oder mit Hilfe des Pflichtenhefts ermittelt werden. Daraus würde für die Erstellung der Testfälle eine erhebliche Aufwandseinsparung resultieren.

In der Testdurchführung wird der eigentliche Vorteil des automatisierten Testprozesses ausgespielt. Da in dieser Testphase bereits sehr geringe Ablaufzeiten vorhanden sind, ist es schwierig diese weiterhin zu optimieren. Das größte Einsparungspotenzial besteht darin, eine minimale Anzahl von Aufruf- und Wiederherstellungsoperationen zu erreichen. Denn jeder Seitenwechsel und jedes Zurücksetzen der Datenbank kann (abhängig von den Eigenschaften der verfügbaren Ressourcen) mehrere Sekunden in Anspruch nehmen. Dies könnte durch eine geschickte Strukturierung der Testfälle und Anordnung der Ablaufreihenfolge realisiert werden. Es muss jedoch bezweifelt werden, dass der Realisierungsaufwand dieses Vorgehens dem Nutzen langfristig gerecht wer-

den kann. Außerdem würde dies einen tiefen Einschnitt in eine flexible Testfallorganisation bedeuten. Alternativ könnte die Webanwendung mit dem Testsystem lokal auf einem einzelnen Rechner installiert werden. Dadurch könnten einerseits sehr schnelle Anfrage- und Antwortzeiten erreicht werden, andererseits befindet sich die Webanwendung bei diesem Vorgehen nicht mehr in ihrer realen Umgebung und sollte deshalb kritisch betrachtet werden.

Das automatisierte Testsystem kann sowohl in der Programmierphase, als auch in der Wartungs- und Pflegephase des Softwareentwicklungsprozesses dem Entwickler bzw. dem Tester assistieren. Für den ersten Fall müssen die Testskripte parallel zum Programmcode erstellt werden. Dies kann sich allerdings negativ auf den Gesamtaufwand auswirken, da zu diesem Zeitpunkt im Allgemeinen häufig mit Programmänderungen zu rechnen ist und die vorhandenen Testfälle dann ebenfalls angepasst werden müssten.

Ein großes Problem stellen mögliche Überlappungen von Testfällen dar. Denn je höher der Überlappungsgrad ist, desto größer ist der Aufwand für die Testdurchführung und insbesondere für die Wartung dieser Testfälle. Somit sollte man bestrebt sein überlappungsarme Testskripte zu erstellen. Bei der Durchführung einer Programmänderung ergibt sich ein weiteres Problem im Zusammenhang mit überlappten Testskripten. Es stellt sich die Frage, wie man die Testskripte finden kann, die ebenfalls anzupassen sind.

Der große Nutzen eines automatisierten Testsystems besteht darin, dass diese Tests wirtschaftlich, nahezu unendlich durchführbar sind, da diese nicht personengebunden sind. Wo Routinetests beim manuellen Testprozess aus Zeitdruck, fehlender Arbeitskraft oder mangelnder Motivation eher entfallen, genügt beim automatisierten Testprozess im Idealfall ein Knopfdruck. Ein weiterer positiver Nebeneffekt ergibt sich durch das höhere Vertrauen in den bereits realisierten Programmcode. Dadurch ist man Programmänderungen gegenüber eher aufgeschlossen. Durch eine kontinuierliche Erhebung der Testaufwände können weiterhin die Aufwände für entsprechende Folgeprojekte zu Planungszwecken besser abgeschätzt werden.

5.2 Der Petrinetzsimulator

Analog zum Aufbau der vorherigen Fallstudie, werde ich zunächst das zu behandelnde Projekt vorstellen und anschließend auf die Ziele der Fallstudie eingehen und deren Umsetzung darstellen und auswerten. Das Ziel dieser Fallstudie ist die Ausarbeitung

eines Konzepts, das ein automatisiertes Testen des web-basierten Petrinetzsimulators ermöglicht.

5.2.1 Projektbeschreibung

Der Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit der BTU Cottbus hat 1997 ein Projekt zur Erstellung einer Modellierungs- und Simulationssoftware für Graph-basierte Systembeschreibungen ins Leben gerufen. Diese Software trägt den Namen Snoopy, kann von der Webseite des Lehrstuhls bezogen und in vollem Umfang verwendet werden. Neben Erreichbarkeitsgraphen (engl. reachability graph) und Fehlerbäumen (engl. fault tree), können damit auch Petrinetze erstellt und deren Abläufe animiert werden [Schulz08: S.10].

Gegenstand der Untersuchungen ist allerdings nicht Snoopy selbst, sondern das web-basierte Animationswerkzeug Patty (in der Version 1.0), welches die von Snoopy erstellten Petrinetze verwendet. Zunächst wird jedoch eine kleine Einführung in Petrinetze vorgenommen, um ein besseres Verständnis im Umgang mit der Software zu erhalten.

Eine kleine Einführung in Petrinetze

Ein Petrinetz kann als bipartiter[15] und gerichteter Graph aufgefasst werden. In Abbildung 12 ist u.a. ein solcher Graph zum Vergleich dargestellt. Die Knoten des Graphen werden in Plätze (Kreissymbole) und Transitionen (Rechteckssymbole) unterschieden. Die Plätze können als Lager für Marken (auch Token genannt) betrachtet werden, deren jeweilige Anzahl an Marken oft mit Punkten oder Zahlen gekennzeichnet wird. Die Markenanzahl pro Platz beschreibt den Systemzustand, die Plätze beschreiben Systembedingungen und die Transitionen stellen die elementaren Aktivitäten des Systems dar. Systemereignisse werden durch das Schalten (auch als Feuern bezeichnet) der jeweiligen Transition ausgelöst. Damit es zum Schalten einer Transition kommen kann, müssen deren Vorbedingungen erfüllt sein. Als Vorbedingung wird ein Platz bezeichnet, der direkt über eine Kante mit der zu schaltenden Transition verbunden ist. Je nach Gewichtung muss eine Mindestanzahl von Marken auf diesem Platz vorhanden sein, um die Vorbedingung zu erfüllen. Äquivalent zur Vorbedingung gibt es auch eine Nachbe-

[15] Ein einfacher Graph, der aus einer Menge von Knoten und einer Menge von Kanten besteht, heißt bipartit, falls sich seine Knoten in zwei disjunkte Teilmengen A und B aufteilen lassen, sodass innerhalb dieser Teilmengen zwischen den Knoten keine Kanten verlaufen. In Petrinetzen entsprechen diese beiden Knotenteilmengen den Plätzen und Transitionen. Somit können zwei Plätze (oder zwei Transitionen) nicht direkt durch eine Kante verbunden sein. Es muss sich dazwischen genau ein Knoten der anderen Teilmenge befinden.

dingung. Somit ist jeder Platz, der über eine Kante verfügt, die von der zu schaltenden Transition zu ihm führt, als Nachbedingung zu betrachten. Die Nachbedingung wird erfüllt, indem die Anzahl der Marken entsprechend der Gewichtung erhöht wird. Sollte es keine Transitionen geben, deren Vorbedingungen erfüllt sind, befindet sich das System in einem toten Zustand, da keine Transition und damit kein Systemereignis ausgelöst werden kann [Schulz08: S.7f].

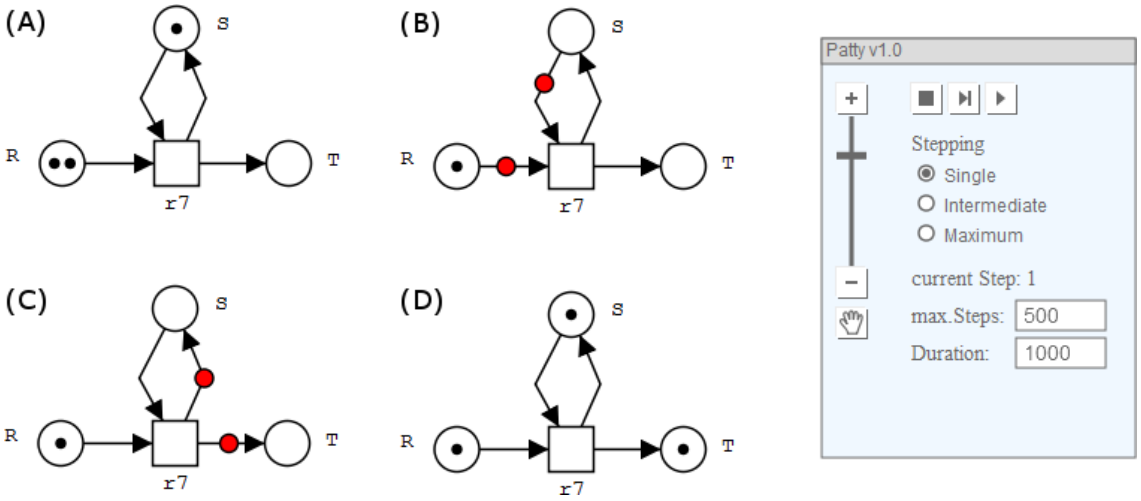


Abbildung 12: Patty – Die grafische Benutzeroberfläche

Neben diesen Grundelementen verfügen Petrinetze ebenfalls über logische Knoten, Makroknoten und spezielle Kanten. Letztere spielen für unsere Betrachtungen allerdings keine Rolle. Logische Knoten dienen der Übersichtlichkeit in Petrinetzen. Muss bspw. ein Platz A sowohl mit Transition X, als auch Transition Y verbunden werden und die komplexe Struktur des vorliegenden Netzes ließe keine überschneidungsfreie Verbindung zu, so kann es zwei identische Plätze A und A‘ geben, welche in ihrem Status und Verhalten absolut identisch sind. Makroknoten (auch als Hierarchien von Petrinetzen bezeichnet) sind ebenfalls Strukturierungselemente für Petrinetze. An jedem Makroknoten ist ein Unternetz definiert, das dieselben Schnittstellen besitzt und somit die Funktionalität des Unternetzes vom darüber liegenden Netz kapselt. Die Analogie zu Prozeduren in Programmen, sollte das Prinzip verdeutlichen.

Das web-basierte Animationswerkzeug für Petrinetze

Die oben beschriebenen Netze können mit dem Editor der Snoopy-Software erstellt und sowohl in Snoopy, als auch in Patty animiert werden. Patty findet derzeit vor allem als installationsfreie Demoversion von Snoopy auf der Webseite des Lehrstuhls Anwen-

dung. Abbildung 12 stellt die grafische Benutzeroberfläche der Webanwendung dar und deutet die Animation der Netze an.

Der Nutzer kann sowohl mit den Plätzen (Kreissymbole), als auch mit den Transistoren (Rechtecksymbole) der Petrinetze interagieren. Während das Klicken auf einen Platz die Anzahl der vorliegenden Marken erhöht bzw. durch zusätzliches Halten der Shift-Taste verringert, löst das Klicken auf eine Transition die entsprechende Animation aus, sofern alle Vorbedingungen erfüllt sind. Da in diesem Beispiel keine Gewichtung der Bedingungen angegeben ist, muss jeder Platz, der zur schaltenden Transition führt, mindestens eine Marke besitzen. Abschnitt (A) zeigt den initialen Zustand des Graphen mit zwei Marken auf dem Platz R und einer Marke auf dem Platz S. Dadurch sind alle Vorbedingungen für das Schalten der Transition r7 erfüllt und die Animation beginnt. Um den Ablauf der Animation steuern zu können, bietet das Panel drei Ablaufmodi an, die mit Hilfe der Buttons gestartet werden können. Durch den Single-Modus wird genau eine zufällig gewählte, schaltfähige Transition pro Animationsschritt ausgelöst. Der Intermediate-Modus führt eine zufällig ermittelte Menge von schaltfähigen Transitionen pro Schritt aus, während der Maximum-Modus eine maximale Menge von konfliktfreien Transitionen in einem Schritt durchläuft. Die Dauer eines Animationsschritts sowie die maximale Anzahl der Schritte kann ebenfalls im Panel angegeben werden. Mit dem Schieberegler kann zusätzlich die Größe und mit dem Handsymbol die Position des Graphen verändert werden. Beinhaltet das zu animierende Netz einen oder mehrere Makroknoten, so stellt das Panel dem Nutzer eine zusätzliche Auswahlliste der Unternetze zur Verfügung, um diese isoliert animieren zu können. Bei den Makroknoten kann ebenfalls zwischen Plätzen und Transitionen unterschieden werden, welche allerdings eine besondere Kennzeichnung aufweisen. Solche Plätze sind innerhalb ihrer Kreissymbole mit einem weiteren Kreis markiert, während die Transitionen innerhalb ihrer Rechtecksymbole ein weiteres Rechteck besitzen. Logische Knoten werden grau hinterlegt dargestellt. Als besonderes Feature können bestimmte Teile eines Netzes eingefärbt werden, was allerdings keinen Einfluss auf die Funktionalität der Elemente hat. Die Färbung dient lediglich der Übersichtlichkeit und ermöglicht den Nutzern eine verbesserte Kommunikation.

Architektur der Patty-Webanwendung

Um einen besseren Einblick in die Funktionsweise des Programms zu erhalten, wird im Folgenden auf die Architektur des Animationswerkzeugs eingegangen. Dieses besteht im Wesentlichen aus drei Komponenten bzw. Dateien, deren Zusammenhang in Abbil-

dung 13 skizziert wurde. Ausgangspunkt ist eine SPPED-Datei, die von der Snoopy-Software erstellt und abgespeichert werden kann. Diese beschreibt alle Anforderungen und Elemente eines Petrinetzes in XML-Notation. Das Ziel ist die Umwandlung dieser Beschreibung in das XML-basierte SVG-Format (Scalable Vector Graphics), welches die Visualisierung der Petrinetzelemente in zweidimensionale Vektorgrafiken ermöglicht. Um dies realisieren zu können, wird eine Transformationsvorschrift (XSL-Datei) verwendet, in der festgelegt wird, welche Elemente der SPPED-Datei, wie transformiert werden. Die eigentliche Umsetzung wird von einem XSLT-Prozessor durchgeführt, der Transformationsanweisungen interpretieren und ausführen kann. Solche Prozessoren sind in den gängigsten Webbrowsern standardmäßig integriert. Um die grafisch-dargestellten Netze animieren zu können, bietet die SVG-Technologie mehrere Möglichkeiten, welche allerdings browserabhängige Probleme mit sich bringen. Deshalb wird zu diesem Zweck auf entsprechende JavaScript-Funktionen zurückgegriffen [Schulz08: S.33ff].

Abbildung 13: Patty – Die drei Architekturkomponenten, vgl. [Schulz08: S.38]

5.2.2 Konzept für ein Testsystem

Zu Beginn dieses Abschnitts wird zunächst geklärt, welche Eigenschaften der Patty-Webanwendung unter Verwendung eines automatisierten Testsystems getestet werden können. Weiterhin werden zwei Konzepte zur Realisierung von Funktionstest vorgestellt. Das Eine ermöglicht einem Entwickler die Definition und Durchführung von Regressionstests, während das Andere einen vollautomatisierten Testprozess darstellt, der Testfälle generieren und ausführen kann. Die Betrachtungen erfolgen allerdings nur auf konzeptioneller Ebene, da eine prototypische Umsetzung den Rahmen dieser Arbeit sprengen würde.

Festlegung des Testkriteriums

Unter Verwendung eines automatisierten Testsystems könnten die Elemente der Petrinetz-Animationssteuerung getestet werden. Damit das Testwerkzeug die entsprechen-

den Zustandsänderungen am Netz verifizieren kann, muss sowohl der Zustand des Netzes, als auch der konkrete Skalierungswert auf der Oberfläche der Webanwendung erscheinen und mit jeder Änderung aktualisiert werden. Beim Test des Play- bzw. Stop-Buttons ergibt sich allerdings ein Problem. Der Play-Button nimmt genau soviele Animationsschritte vor, wie in dem zugehörigen Eingabefeld angegeben ist. Da die Petrinetze aber aufgrund der integrierten Zufallskomponenten im Allgemeinen nicht deterministisch sind, kann das Testwerkzeug bei identischen Testdurchläufen auf verschiedene Endzustände des Netzes stoßen. Deshalb müsste in einem Testsystem die Webanwendung so angepasst werden, dass ein künstlicher Determinismus vorhanden ist, der statt einer zufälligen Auswahl schaltbarer Transitionen, diese stets nach einer festgelegten Abfolge ermittelt. Damit könnten dann ebenfalls komplette Animationsabläufe eines Petrinetzes in einem einzigen Testschritt durchgeführt werden. Unabhängig von dieser Anpassung können einzelne Transitionen sowie das gesamte Netz (in einer Sequenz einzeln-getesteter Transitionen) getestet werden.

Die Funktionalität der Petrinetz-Grundelemente (Plätze und Transitionen) scheint im web-basierten Simulator sehr stabil zu sein. Doch es ist bekannt, dass sich bei der Verwendung von erweiterten Netzelementen Abweichungen zeigen können, die unter Umständen zu Fehlfunktionen führen. Dies ist insbesondere bei Makroknoten der Fall. Um Makroknoten testen zu können, müssen die Schnittstellen des Hauptnetzes (siehe Abbildung 14, Graph 1) mit den Schnittstellen des Unternetzes (Graph 2), jeweils vor

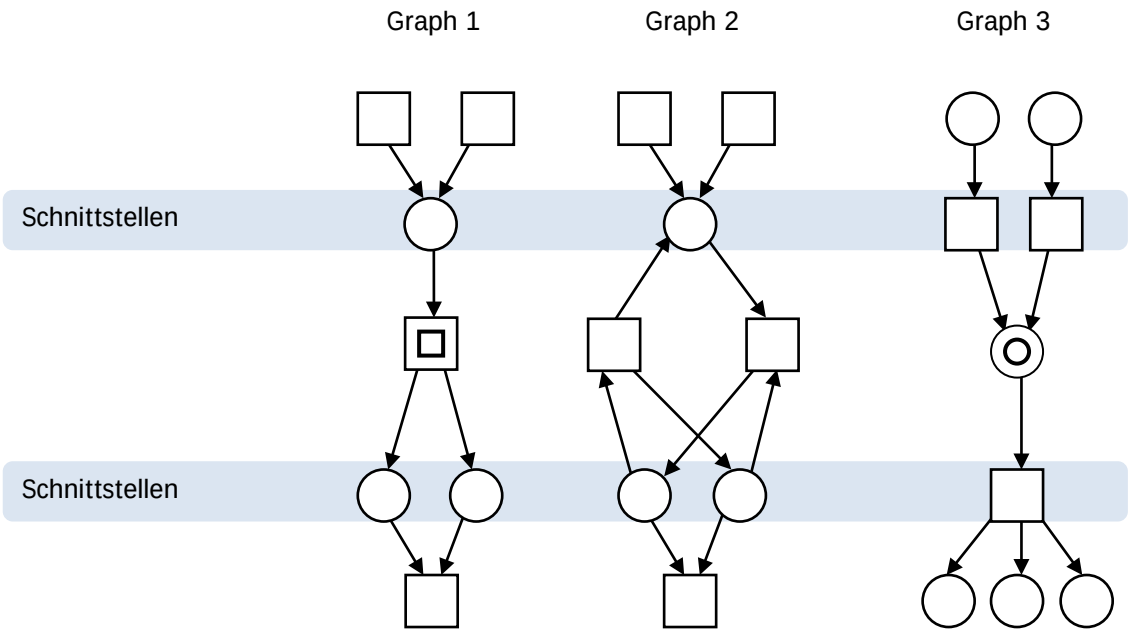


Abbildung 14: Beispiele von Makroknoten eines Petrinetzes

und nach der Animation, verglichen werden. Schnittstellen können sowohl Transitionen (Graph 3), als auch Plätze sein. Für die folgende Betrachtung wird sich jedoch ausschließlich auf Makroknoten bezogen, die als Schnittstellen Plätze aufweisen (im Folgenden Makrotransitionen genannt). Klickt man beim manuellen Test eine Makrotransition an, so wird im Unternetz stets ein Animationsschritt im Maximum-Modus durchgeführt. Das Problem beim automatisierten Testen besteht darin, dass bei komplexen Unternetzen durchaus eine Zustandsänderung stattfinden kann, die allerdings im Hauptnetz nicht registriert werden konnte. Zur Erstellung eines Testfalls würde ein Entwickler bspw. mit Hilfe der Animationssteuerung (im Maximal-Modus) das Unternetz solange simulieren, bis sich dieses entweder in einem toten Zustand befindet oder sich die Markenanzahl der Zielschnittstelle geändert hat. Im ersten Fall wäre der Makroknoten nicht testbar, da kein Vergleich der Zielschnittstellen stattfinden kann. Im zweiten Fall müsste anschließend die Makrotransition des Hauptnetzes genau so oft aktiviert werden, um letztendlich eine Aussage über den Erfolg bzw. Misserfolg eines Testfalls treffen zu können. Makroknoten, die Transitionen als Schnittstellen haben, sind aufgrund ihrer Struktur komplizierter in einem Testfall zu beschreiben. Eine einfache Umformung zu einer Makrotransition würde eine elegante Lösung des Problems darstellen.

Zum Testen der Position oder der Farbe von Petrinetzelementen müsste ein Testwerkzeug genutzt werden, das Grafikerkennungsfunktionen bietet. Bei Verwendung der Patty-Webanwendung sind solche Fehler in vereinzelten Petrinetzen festzustellen. Als Beispiel ist der Kommentarblock eines Petrinetzes zu nennen, der sich laut Definition zentriert unterhalb des Netzes befinden sollte und stattdessen unterhalb-rechts zu beobachten ist. Weiterhin kann es vorkommen, dass sich die Angabe der Markenanzahl nicht mehr innerhalb ihres Platzsymbols befindet, sondern etwas außerhalb versetzt aufzufinden ist. Mit einem automatisierten Testsystem allein könnten die Ursachen solcher Fehler nicht ermitteln werden. Es kann dem Entwickler lediglich zu Assistenzzwecken dienen, indem umständliche Interaktionsfolgen automatisiert werden.

Testen der Softwaredokumente

Jedes verwendete Softwaredokument stellt eine potenzielle Fehlerquelle dar, die es zu testen gilt. Ob es allerdings sinnvoll ist diesen Vorgang zu automatisieren soll im Folgenden diskutiert werden.

Die SPPED-Datei, welche eine komplette Beschreibung des Netzes in XML enthält, stellt das Ausgangsdokument des Programms dar. Um die Gültigkeit von XML-Dokumenten

zu ermitteln, kommen XML-Schemas zum Einsatz, die eine exakte Beschreibung der Dokumentenstruktur beinhalten. Die eigentliche Überprüfung wird von einem Hilfsprogramm (dem Validator) übernommen, der das vorliegende Dokument mit dem Schema abgleicht. Ein solcher Abgleich könnte automatisiert vor jedem Programmstart erfolgen und dafür Sorge tragen, dass eine korrekte Datenstruktur vorliegt. Im Internet steht eine Vielzahl von kostenlosen Validatoren zur Verfügung, die in den Testprozess eingebunden werden können. Als Beispiel seien hier der Validator der Organisation für Webstandards W3C (World Wide Web Consortium) und der Validator von Validome[16] genannt. Das einzige Problem stellen die verwendeten IDs der XML-Elemente dar, die auf Nummern basieren. Denn XML-Schema erlaubt keine Nummern, sondern verlangt Bezeichner, die mit einem Buchstaben beginnen. Eine entsprechende Anpassung des SPPED-Formats müsste vorgenommen werden. Für die weiteren Untersuchungen wird die Gültigkeit der SPPED-Datei als gegeben betrachtet.

Die Transformationsvorschrift, welche das Petrinetz aus dem SPPED-Format in eine grafische Darstellung (dem SVG-Format) umwandelt, stellt eine weitere potenzielle Fehlerquelle dar, die es zu testen gilt. Dies lässt sich über einen einfachen Vergleich der beiden Formate realisieren, denn zu jedem Element des einen Formats existiert ein äquivalentes Element des anderen Formats. Um diesen Vorgang automatisiert ablaufen lassen zu können, müssten die einzelnen Transformationsregeln umständlich im Testskript definiert werden, was mit sehr viel Aufwand verbunden wäre. Aus diesem Grund sollte an dieser Stelle eher ein manueller Test erfolgen.

Das dritte und letzte, beteiligte Softwaredokument entspricht dem Transformationsergebnis (das Petrinetz im SVG-Format). Dieses stellt die Elemente des Netzes im Webbrowser grafisch dar und greift auf JavaScript-Funktionen zurück, um die Abläufe zu animieren. Hierbei bieten sich mehrere Testansätze an. Man könnte die dargestellten Objekte und deren Animationen auf Konsistenz hin überprüfen. Da Snoopy als Grundlage von Patty anzusehen ist, wäre ein grafischer Vergleich desselben Petrinetzes in beiden Programmen denkbar. Weiterhin sollte die Konsistenz der Darstellungen in verschiedenen Webbrowser abgeglichen werden. Ein menschlicher Tester würde grafische Fehler im Allgemeinen sofort feststellen können, während ein Programm auf Bilderkennungsfunktionen angewiesen ist, deren Umsetzung im Vergleich zu Text-basierten Tests sehr komplex werden kann.

[16] www.validome.org/xml

Konzept eines Funktionstests für Petrinetze

Betrachtet man die Patty-Software jedoch nicht in ihren einzelnen Dokumenten, sondern von einer höheren Abstraktionsebene aus als Ganzes, so können unter bestimmten Voraussetzungen automatisierte Funktionstests erstellt und durchgeführt werden. Patty ist eine reine clientseitige Webanwendung, die nach dem Laden der Softwaredokumente im Browser selbstständig abläuft und somit keine Interaktion mit dem Webserver benötigt. Deshalb kann hierbei kein Testwerkzeug verwendet werden, das die Automatisierung ausschließlich auf HTTP-Ebene ermöglicht, wie bspw. Sahi oder Canoo WebTest. Es muss ein Browserautomatisierungswerkzeug sein, welches das exakte Verhalten eines Nutzers simulieren kann, wie z.B. iMacros oder Selenium. Diese sind in Kapitel 4 vorgestellt worden. Erste Versuche haben allerdings gezeigt, dass die iMacros-Vollversion keine SVG-Formate darstellen kann, was aufgrund der vielversprechenden Produktanpreisung etwas enttäuschend war. Auch das kostenlose Browser-Plug-In von iMacros kommt für dieses Testsystem nicht in Betracht, weil damit die einzelnen Petrinetzelemente nicht aktivierbar bzw. klickbar sind. Allein mit Selenium ließen sich die Transitionen automatisch auslösen und die Marken der Plätze beliebig erhöhen. Ein Verringern ist jedoch nicht möglich, weil beim Klicken zusätzlich die Shift-Taste gehalten werden müsste und dies von keinem der getesteten Werkzeuge unterstützt wird. Um die Anzahl der Marken zu verringern, müsste das Netz neu geladen und die Plätze mit der gewünschten Markenanzahl versehen werden.

Ein weiterer, wichtiger Punkt ist die Verifizierung der Elemente, um feststellen zu können, ob der Test gescheitert oder erfolgreich war. Dies stellt ein Problem dar, da Patty keine Textausgaben, sondern lediglich grafische Strukturen aufweist. Wie bereits oben erwähnt wurde, ist der Status eines Petrinetzes durch die Markenanzahl pro Platz definiert. Durch ein einfaches Abspeichern der Webseite aus dem Browser heraus, werden alle JavaScript-Dateien gesichert und das Transformationsergebnis (SVG-Format) in einer XML-Datei abgelegt. Der aktuelle Status, der zum Zeitpunkt der Speicherung vorlag, ist innerhalb dieser XML-Datei notiert und könnte mit Hilfe externer Skripte automatisiert ausgelesen werden. Den Knackpunkt dieser Methode stellt jedoch das automatisierte Speichern der Webseite dar. Zum Einen ist dieser Vorgang sehr rechenintensiv, da er nach jedem Statuswechsel komplett durchgeführt werden muss. Zum Anderen sehe ich als einzige Möglichkeit zur Umsetzung den Einsatz des JavaScript-Befehls „execCommand()“, der allerdings nur im Internet Explorer funktionsfähig ist. Und da der Internet Explorer standardmäßig das SVG-Format nicht unterstützt, ist diese Art

der Statusermittlung unbefriedigend. Eine zweite Möglichkeit besteht darin, den aktuellen Status des Netzes nach jedem Animationsschritt in Textform auszugeben. Dieser Text könnte bspw. in Tabellenform angegeben werden, der mit Hilfe zusätzlicher JavaScript-Funktionen parallel zur Ausführung aktualisiert wird. Damit kann der Status des Netzes jederzeit vom Testwerkzeug ermittelt und verifiziert werden.

Um den Automatisierungsprozess abzurunden, sollte eine Testumgebung vorhanden sein, welche für eine strukturierte Abfolge sorgt. Da in diesem System Selenium RC als Testwerkzeug zum Einsatz kommt, sind bereits alle definierten Testfälle in Klassen der verwendeten Programmiersprache strukturiert und können mit Hilfe der verwendeten Entwicklungsumgebung gepflegt und verwaltet werden.

Konzept eines Zufall-basierten Funktionstests für Petrinetze

Mit diesem Ansatz soll nicht nur die automatische Durchführung, sondern auch die automatische Erstellung der Testfälle verfolgt werden. Die oben beschriebenen Voraussetzungen dienen hier als Grundlage. Die Architektur dieses Ansatzes besteht aus dem Testfallgenerator und dem Testwerkzeug.

Der interne Ablauf eines möglichen Testfallgenerators ist in Abbildung 15 dargestellt. Dies ist ein Implementierungsvorschlag, der mit Hilfe einer Programmiersprache realisiert werden kann, welche Dateioperationen und die Erzeugung von Zufallszahlen ermöglicht. Die Grundlage der Durchführung bildet hierbei das Petrinetz als SPPED-Datei. Im ersten Schritt werden die einzelnen Elemente des Netzes ermittelt. Jeder Platz erhält eine zufällig bestimmte Anzahl von Marken (bspw. im Bereich von 0 bis 3), wodurch ein zufälliger Status generiert wird. Anschließend wird erneut per Zufall eine Transition ausgewählt, die durch das Testwerkzeug aktiviert wird und damit ein Systemereignis auslöst. Ist die Vorbedingung dieser Transition erfüllt, so kann der zu erwartende Systemstatus ermittelt und der gesamte Ablauf (Initialstatus, Systemereignis, Erwartungsstatus) in der gewählten Testfallbeschreibungssprache formatiert werden. Eine zu wählende Abbruchbedingung beendet den Generierungsprozess, leitet die Testfälle an das Testwerkzeug weiter und startet dieses.

Als Werkzeug kommt auch hierfür Selenium RC zum Einsatz, dem die Testfälle diesmal nicht „per Hand“, sondern automatisch zugespielt werden müssen. In der verwendeten Entwicklungsumgebung muss ein Initialprojekt vorhanden sein, das bereits eine leere Klassendatei enthält. Der Testfallgenerator kann Selenium somit seine generierten

Testfälle über einfache Dateioperationen zukommen lassen und das Testwerkzeug über Kommandozeilenbefehle aufrufen.

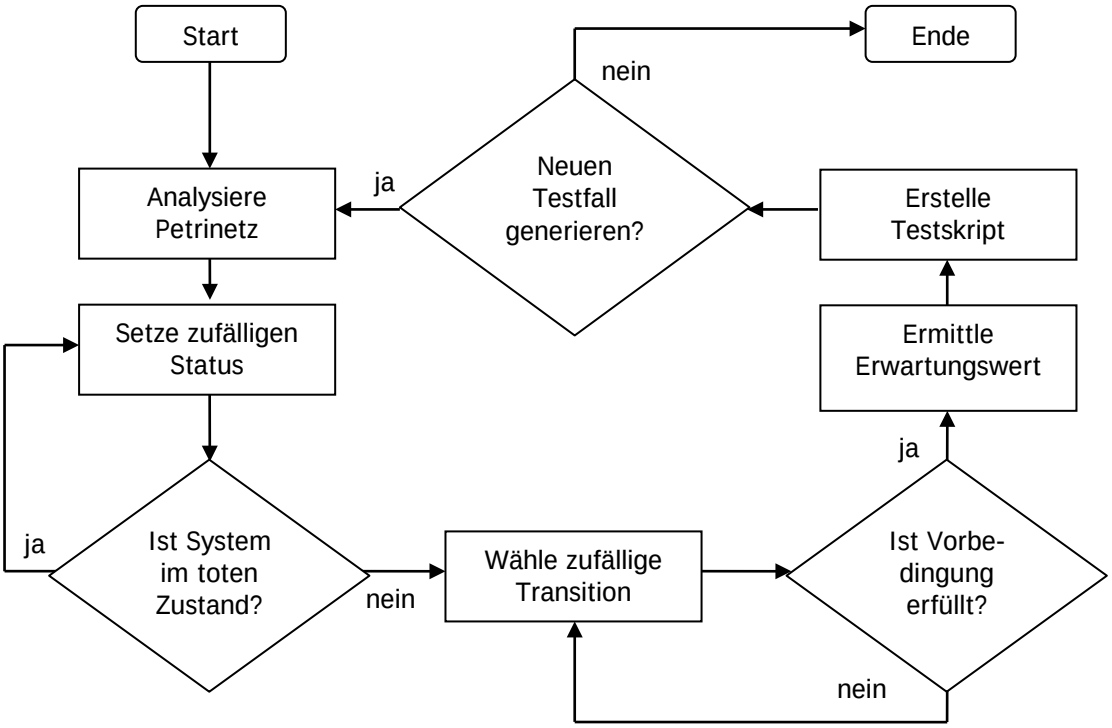


Abbildung 15: Der Algorithmus eines mögl. Testfallgenerators

6 Zusammenfassung

Das Ziel dieser Arbeit war es aktuelle Testwerkzeuge für Webanwendungen zu untersuchen und wesentliche Unterschiede zwischen kommerziellen und quelloffenen Testwerkzeugen herauszuarbeiten. Anhand der Juleica-Webanwendung, welche von der epion GmbH entwickelt worden ist, sollen die Aufwände eines manuellen Testprozesses mit den Aufwänden eines automatisierten Testprozesses näher betrachtet und verglichen werden. Die zweite Fallstudie beschäftigt sich mit dem grafischen Petrinetzsimulator Patty des Lehrstuhls für Datenstrukturen und Softwarezuverlässigkeit der BTU Cottbus. Es soll geklärt werden, welchen Nutzen ein automatisierbares Testsystem hätte und welche Voraussetzungen geschaffen werden müssten, damit ein Solches zum Einsatz käme.

Zur Realisierung eines Web-Testsystems lohnt es sich in den meisten Fällen die Verwendung von quelloffenen Testwerkzeugen in Erwägung zu ziehen. Insbesondere Selenium und Watir stehen kommerziellen Produkten im Allgemeinen in nichts nach und bieten somit eine kostengünstige Alternative. Sowohl Canoo WebTest, als auch Sahi stellen solide Nischenprodukte dar, die ihre Stärken beim Einsatz von reinen HTML-basierten Webanwendungen zum Vorschein bringen. Die Weiterentwicklung vom Testwerkzeug zum Entwicklungswerkzeug bei Programmiersprachen, wie bspw. Java oder C++, könnte sich in Zukunft auch bei Testwerkzeugen für Webanwendungen zeigen. Insbesondere die Umsetzung einer testgetriebenen Entwicklung wäre ein interessanter Ansatz, den man im Rahmen einer wissenschaftlichen Arbeit näher betrachten könnte.

Durch die Nutzung eines automatisierbaren Testsystems kann ein erheblicher Geschwindigkeitsvorteil in der Testdurchführung erreicht werden. Weiterhin können diese Tests mit geringem Aufwand nahezu unendlich oft wiederholt werden. Dem stehen allerdings hohe Entwicklungskosten entgegen, die sich erst durch eine intensive (evtl. projektübergreifende) Verwendung ausgleichen lassen. Für die Juleica-Webanwendung ergibt sich auf Grundlage der Messdaten ab der 76. Testdurchführung eine Aufwandsersparnis von über 10% und ab der 96. Testdurchführung liegt die Einsparung sogar schon bei 25%. Diese Zahlen berücksichtigen allerdings nicht die Aufwände, die zusätzlich bei einer Testfalländerung angefallen wären, da eine Ermittlung dieser Daten selbst in einer Masterarbeit zu zeitintensiv gewesen wäre. Um die Entwicklungskosten des Juleica-Autotestsystems zu verringern, kann es wirtschaftlicher sein, Testskripte ausschließlich für die Hauptfunktionen der Webanwendung zu entwickeln, und selten

verwendete Nebenfunktionen bei Bedarf manuell zu testen. Man darf nicht vergessen, dass es sich bei einem solchen Testsystem ebenfalls um eine Software handelt, die Fehler enthalten kann sowie gewartet und gepflegt werden muss.

Im Rahmen der Patty-Machbarkeitsstudie ist ein Konzept zur Realisierung von Funktionstests entwickelt worden, das um einen Generator zur automatischen Erzeugung von Testskripten erweitert werden kann. Zur Realisierung dieses Konzepts muss der Systemzustand der Petrinetze mit jedem Animationsschritt der Webanwendung als Text ausgegeben werden, damit das Testwerkzeug diesen verifizieren kann. Die Implementierung dieses Konzeptschritts stellt dabei den größten Aufwand der Umsetzung dar. Darstellungsfehler wie z.Bsp. Farb- oder Positionsänderungen können damit allerdings nicht getestet werden. Für eine prototypische Realisierung des Konzepts hat die Bearbeitungszeit leider nicht mehr ausgereicht. Dies könnte jedoch das Thema einer weiteren wissenschaftlichen Arbeit sein.

Literaturverzeichnis

[ActiveX]
Microsoft. Introduction to ActiveX Controls.
http://msdn.microsoft.com/en-us/library/aa751972(VS.85).aspx
Letzter Zugriff am 06.09.2009

[BA08]
o.A. Lack of software testing to blame for Terminal 5 fiasco. Veröffentlicht am 08.05.2008
http://www.computerweekly.com/Articles/2008/05/08/230602/lack-of-software-testing-to-blame-for-terminal-5-fiasco-ba-executive-tells.htm
Letzter Zugriff am 26.06.2009

[Balzert08]
Helmut Balzert. Lehrbuch der Softwaretechnik, Softwaremanagement. Spektrum Akademischer Verlag, 2008

[Balzert99]
Helmut Balzert. Lehrbuch Grundlagen der Informatik. Spektrum Verlag, 1999

[Cone06]
Edward Cone. Inside eBay's Innovation Machine. Veröffentlicht am 06.12.2006
http://www.cioinsight.com/c/a/Case-Studies/Inside-eBays-Innovation-Machine/2/
Letzter Zugriff am 26.06.2009

[epion]
epion GmbH. Juleica-Autotest, Arbeitspapier 3, Version 1.2.
(vertraulicher Inhalt. Einsicht möglich, falls für die Bewertung der vorliegenden Arbeit relevant)

[GNU1]
GNU.de. Open Source.
http://www.gnu.de/free-software/open-source.de.html
Letzter Zugriff am 10.06.2009

[GNU2]
GNU.de. Quelltext.
http://www.gnu.de/free-software/source.de.html
Letzter Zugriff am 10.06.2009

[HWM06]
Hauser, Wenz, Maurice. Das Website Handbuch. Markt+Technik Verlag, 2006

[HTMLUnit]
Gargoyle Software Inc. HTMLUnit. Veröffentlicht am 21.04.2009
http://htmlunit.sourceforge.net/
Letzter Zugriff am 11.07.2009

[IABG]
Industrieanlagen-Betriebsgesellschaft mbH. Das V-Modell.
http://v-modell.iabg.de/
Letzter Zugriff am 23.07.2009

[iMacros]
iOpus Software GmbH. iMacros Feature Comparison - Free and Business Editions
http://www.iopus.com/imacros/compare/all/index.htm
Letzter Zugriff am 06.06.2009

[JavaApp]
Galileo Computing. Applets.
http://openbook.galileocomputing.de/javainsel8/javainsel_21_001.htm#mj7e21089d9491c64f1edfd41d5c869327
Letzter Zugriff am 06.09.2009

[Juleica1]
Juleica. Antrag und Infos.
http://www.juleica.de/14.0.html
Letzter Zugriff am 01.09.2009

[Juleica2]
Juleica. Grafiken.
http://www.juleica.de/35.0.html
Letzter Zugriff am 01.09.2009

[Juleica3]
Juleica. Antragsverfahren.
http://www.juleica.de/17.0.html
Letzter Zugriff am 01.10.2009

[Liquid1]
JadeLiquid Software. LiquidTest product break-down.
http://www.jadeliquid.com/liquidtest/docs/doku.php?id=general:developers_and_testers:home
Letzter Zugriff am 06.06.2009

[Liquid2]
JadeLiquid Software. What is Liquidtest.
http://www.jadeliquid.com/liquidtest/docs/doku.php?id=general:what_is_liquidtest:home
Letzter Zugriff am 12.08.2009

[Long07]
Tony Long. The Man Who Saved the World by Doing ... Nothing. Veröffentlicht am 26.09.2007
http://www.wired.com/science/discoveries/news/2007/09/dayintech_0926
Letzter Zugriff am 26.06.2009

[LR]
Lausitzer Rundschau. Juleica leichter zu haben. Veröffentlicht am 13.02.2009
http://www.lr-online.de/nachrichten/wirtschaft/wirtschaft-lr/Juleica-leichter-zu-haben;art1067,2400366,0
Letzter Zugriff am 06.06.2009

[QEngine]
ZOHO Corporation**.** AdventNet Inc. is now ZOHO Corporation.
http://www.manageengine.com/news/zoho-corporation-formerly-adventnet.html
Letzter Zugriff am 07.06.2009

[Sahi]
Sahi Software. How Sahi works.
http://sahi.co.in/w/how-sahi-works
Letzter Zugriff am 08.06.2009

[Schulz08]
Krispin Schulz. Bachelorarbeit – Eine Erweiterung der Software Snoopy zur Verarbeitung und Verwaltung von Petrinetz-Animationen. Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit an der Brandenburgischen Technischen Universität Cottbus, 2008.

[Selenium1]
ThoughtWorks. Selenium-RC.
http://seleniumhq.org/docs/05_selenium_rc.html
Letzter Zugriff am 23.07.2009

[Selenium2]
OpenQA. Testing Flash with Selenium RC.
http://wiki.openqa.org/display/SRC/Testing+Flash+with+Selenium+RC
Letzter Zugriff am 06.11.2009

[Thaller02]
Georg Erwin Thaller. Software-Test, Verifikation und Validation. Heise Verlag, 2002

[Vigenschow05]
Uwe Vigenschow. Objektorientiertes Testen und Testautomatisierung in der Praxis.
dpunkt.Verlag, 2005.

[VModellXT]
Industrieanlagen-Betriebsgesellschaft mbH
http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Deutsch-V1.3.pdf
Letzter Zugriff am 06.06.2009

[WebTest]
Canoo Engineering AG. Manual.
http://webtest.canoo.com/webtest/manual/manualOverview.html
Letzter Zugriff am 12.07.2009

[Westphal06]
Frank Westphal. Testgetriebene Entwicklung mit JUnit & FIT – Wie Software änderbar bleibt.
dpunkt.Verlag, 2006.

[XP]
Frank Westphal. Extreme Programming. Veröffentlicht am 26.08.2001
http://www.frankwestphal.de/ExtremeProgramming.html
Letzter Zugriff am 06.06.2009

Onlinequellen für den Abschnitt 4.3.2

Aus Gründen der Übersichtlichkeit wird Abschnitt 4.3.2 (Vergleich aktueller Testwerkzeuge) nicht für jedes einzelne Feature ein extra Verweis zur Onlinequelle angegeben. Die für diesen Vergleich genutzten Quellen werden hier aufgelistet und sind speziell markiert. So bezieht sich z.B. die Quellenangabe mit dem Kürzel [VaT-Sele1] auf die genutzte Online-Quelle, wobei VaT (Vergleich aktueller Testwerkzeuge) für diesen Abschnitt und Sele für das betrachtete Web-Testwerkzeug (hier Selenium) steht. Dies schließt jedoch nicht aus, dass diese Quellen auch in anderen Abschnitten der Arbeit genutzt werden können.

[VaT-iMac1]
iOpus Software. iOpus Internet Macros.
http://www.iopus.com/download/box/imacros-flyer-web.pdf
Letzter Zugriff am 11.06.2009

[VaT-iMac2]
iMacros Wiki. Examples for using iMacros efficiently.
http://wiki.imacros.net/Sample_Code
Letzter Zugriff am 29.09.2009

[VaT-iMac3]
iOpus Software. iMacros.
http://www.iopus.com/imacros
Letzter Zugriff am 12.06.2009

[VaT-iMac4]
iOpus Software. What‘s new.
http://www.iopus.com/imacros/support/changelog/
Letzter Zugriff am 12.06.2009

[VaT-iMac5]
iOpus Software. Software Store.
http://www.iopus.com/store/
Letzter Zugriff am 12.06.2009

[VaT-Liqu1]
JadeLiquid Software. LiquidTest Dokumentation.
http://www.jadeliquid.com/liquidtest/docs/doku.php
Letzter Zugriff am 11.06.2009

[VaT-Liqu2]
JadeLiquid Software. LiquidTest 1.0.8 Update.
http://www.jadeliquid.com/liquidtest/docs/doku.php?id=general:updates:update_1_0_8:home
Letzter Zugriff am 12.06.2009

[VaT-Liqu3]
JadeLiquid Software. Introductory Pricing Structure.
http://www.jadeliquid.com/liquidtest/pricing/index.php?z=d
Letzter Zugriff am 12.06.2009

[VaT-QEng1]
ZOHO Corporation. AdventNet, QEngine Web Functional Test.
http://www.manageengine.com/products/qengine/functional-testing-datasheet.pdf
Letzter Zugriff am 11.06.2009

[VaT-QEng2]
ZOHO Corporation. QEngine - Functional Testing.
http://www.manageengine.com/products/qengine/functional-testing.html
Letzter Zugriff am 12.06.2009

[VaT-QEng3]
ZOHO Corporation. AdventNet QEngine WebTest.
https://store.adventnet.com/product/PD.jsp?filter=10034&p=10492
Letzter Zugriff am 12.06.2009

[VaT-Sahi1]
Sahi Software. Features.
http://sahi.co.in/w/features
Letzter Zugriff am 11.06.2009

[VaT-Sahi2]
Sahi Software. Sahi V2 Release 2009-05-21 is now available.
http://blog.sahi.co.in/
Letzter Zugriff am 12.06.2009

[VaT-Sele1]
ThoughtWorks. Introducing Selenium.
http://seleniumhq.org/docs/01_introducing_selenium.html
Letzter Zugriff am 11.06.2009

[VaT-Sele2]
ThoughtWorkers on Open Source: Selenium 1.0 released.
http://fav.or.it/post/1491306/thoughtworkers-on-open-source-selenium-10-released
Letzter Zugriff am 12.06.2009

[VaT-Squi1]
froglogic GmbH. Squish for Web.
http://www.froglogic.com/download/datasheet_squishweb.pdf
Letzter Zugriff am 11.06.2009

[VaT-Squi2]
froglogic GmbH. Squish for Web.
http://www.froglogic.com/pg?id=NewsEvents&category=110
Letzter Zugriff am 11.06.2009

[VaT-Squi3]
froglogic GmbH. Price & Licensing.
http://www.froglogic.com/pg?id=Products&category=squish&sub=price
Letzter Zugriff am 12.06.2009

[VaT-Wati1]
RubyForge. Platforms.
http://wtr.rubyforge.org/platforms.html
Letzter Zugriff am 11.06.2009

[VaT-Wati2]
Paul Rogers.
http://members.shaw.ca/paul_rogers/index.html
Letzter Zugriff am 08.06.2009

[VaT-Wati3]
Prasanth James. Selenium vs. Watir. Veröffentlicht am 04.01.2009
http://prasanthjames.blogspot.com/2009/01/selenium-vs-watir.html
Letzter Zugriff am 11.06.2009

[VaT-Wati4]
Open QA. FAQ.
http://wiki.openqa.org/display/WTR/FAQ
Letzter Zugriff am 12.06.2009

[VaT-Wati5]
RubyForge. Files.
http://rubyforge.org/frs/?group_id=104
Letzter Zugriff am 12.06.2009

[VaT-Web21]
itCampus Software- und Systemhaus GmbH. Web2Test, Web Application Test Tool.
http://web2test.de/wp-content/uploads/whitepaper_de.pdf
Letzter Zugriff am 11.06.2009

[VaT-Web22]
itCampus Software- und Systemhaus GmbH. New Maintenance Release 1.2.1
http://de.web2test.de/2009/05/06/release-121/#more-677
Letzter Zugriff am 12.06.2009

[VaT-Web23]
itCampus Software- und Systemhaus GmbH. Keyfeatures und Vorteile.
http://de.web2test.de/funktionsliste/
Letzter Zugriff am 12.06.2009

[VaT-WebI1]
Corey Goldberg. WebInject Manual.
http://www.webinject.org/manual.html
Letzter Zugriff am 11.06.2009

[VaT-WebI2]
Corey Goldberg. Frequently Asked Questions About WebInject.
http://www.webinject.org/faq.html
Letzter Zugriff am 12.06.2009

[VaT-WebI3]
Corey Goldberg. Download WebInject.
http://www.webinject.org/download.html
Letzter Zugriff am 12.06.2009

[VaT-WebT1]
Canoo Engineering AG. WebTest Key Characteristics.
http://webtest.canoo.com/webtest/manual/keyCharacteristics.html
Letzter Zugriff am 11.06.2009

[VaT-WebT2]
Canoo Engineering AG. config.
http://webtest.canoo.com/webtest/manual/config.html
Letzter Zugriff am 12.06.2009

[VaT-WebT3]
Canoo Engineering AG. Software License.
http://webtest.canoo.com/webtest/manual/license.html
Letzter Zugriff am 12.06.2009

- Anhang -

Anhang A

Testskript 1: Stellen eines Antrags nach vorläufiger Ablehnung durch den einen Träger

```
<?xml version="1.0"?>
<!DOCTYPE project SYSTEM "../../dtd/Project.dtd" [<!ENTITY config SYSTEM
"../../includes/config.xml">]>
<project default="Autotest">
  <target name="Juleica">
    <webtest name="Stellen eines Antrags nach vorläufiger Ablehnung durch den einen
Träger">
      &config;
      <steps>
        <echo>##*Schritte des UseCase;;;Seite;;;User;;;Benutzergruppe;;;Kurzbezeichnung;;;
Bezug zum Konzept</echo>
        <echo>##*Nutzer Login;;;1;;;Test User;;;Antragsteller;;;Der Nutzer loggt sich
ein.;;;GP01, M01</echo>
        <invoke url="http://juleicaautotest.epion.de/"/>
        <setInputField name="user" value="juleica_test_03@epion.de"/>
        <setInputField name="pass" value="iw9IxzglfdJXMPgdto6s"/>
        <clickButton label="Login"/>
        <verifyTitle text="Status"/>
        <verifyText text="Antrag wurde vorläufig abgelehnt."/>
        <echo>##*Antrag bearbeiten;;;10;;;Test User;;;Antragsteller;;;Der Antragsteller kann
seinen früher begonnen Antrag bearbeiten.;;;GP03, M02</echo>
        <clickButton label="Antrag bearbeiten"/>
        <clickButton label="Antrag speichern + weiter zu Schritt 2"/>
        <clickButton label="Weiter zu Schritt 3"/>
        <clickButton label="Weiter zu Schritt 4"/>
        <clickButton label="Weiter zu Schritt 5"/>
        <clickButton label="Weiter zu Schritt 6"/>
        <clickButton label="Weiter zu Schritt 7"/>
        <verifyTitle text="Antrag stellen"/>
        <verifyText text="FT Ebene 1 Brandenburg"/>
        <echo>##*Bestätigung des Antrages;;;23;;;Test User;;;Antragsteller;;;Der Antragsteller
erhält eine Bestätigung das der Antrag gestellt wurde.;;;GP03, M10</echo>
        <clickButton label="Juleica-Antrag stellen"/>
        <verifyText text="Vielen Dank, Test User!"/>
        <verifyText text="Dein Antrag auf eine Juleica ist erfolgreich bei uns eingegangen und
wird umgehend bearbeitet."/>
        <verifyText text="Über den Bearbeitungsstatus Deines Antrages wirst Du entsprechend
informiert."/>
        <verifyText text="Eine Druckversion zum Antrag kannst Du über den unten stehenden
Link aufrufen."/>
        <verifyText text="Mit freundlichen Grüßen"/>
        <verifyText text="das Juleica-Team"/>
        <echo>##*Logout;;;23;;;Test User;;;Antragsteller;;;Logout durch den
User;;;GP01</echo>
        <clickElement xpath="//html/body/table/tbody/tr/td/div[@id='content-wrap']/div
[@id='content']/div[@id='menumain']/div[3]/form"/>
        <verifyTitle text="Startseite"/>
        <verifyText text="Herzlich Willkommen!"/>
      </steps>
    </webtest>
  </target>
</project>
```

Testskript 2: Antrag genehmigen durch Freien Träger

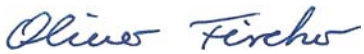
```
<?xml version="1.0"?>
<!DOCTYPE project SYSTEM "../../dtd/Project.dtd" [<!ENTITY config SYSTEM
"../../includes/config.xml">]>
<project default="Autotest">
  <target name="Juleica">
    <webtest name="Antrag genehmigen durch Freien Träger">
      &config;
      <steps>
          <echo>##*Schritte des UseCase;;;Seite;;;User;;;Benutzergruppe;;;Kurzbezeichnung;;;
Bezug zum Konzept</echo>
          <echo>##*Nutzer Login;;;1;;;Test User;;;Antragsteller;;;Der Nutzer loggt sich
ein.;;;GP01, M01</echo>
          <invoke url="http://juleicaautotest.epion.de/"/>
          <setInputField name="user" value="juleica_test_02@epion.de"/>
          <setInputField name="pass" value="iw9IxzglfdJXMPgdto6s"/>
          <clickButton label="Login"/>
          <echo>##*Übersicht der der Anträge;;;42;;;FT Ebene 1 Brandenburg;;;Freier
Träger;;;Der Träger erhält eine detaillierte Übersicht der Anträge.;;;GP31, M31</echo>
          <invoke url="http://juleicaautotest.epion.de/index.php?id=42&action=menue"/>
          <echo>##*Details zum Antrag;;;34;;;FT Ebene 1 Brandenburg;;;Freier
Träger;;;Anzeige der Details zum bestehenden Antrag;;;GP31, M32</echo>
          <clickElement xpath="//html/body/table/tbody/tr/td/div[@id='content-wrap']/div
[@id='content']/div[2]/div[2]/div[@id='content-left']/div[11]/table[2]/tbody/tr[2]/td[6]/div/
form"/>
          <echo>##*Antrag genehmigen;;;34;;;FT Ebene 1 Brandenburg;;;Freier Träger;;;Der
Antrag wird genehmigt;;;GP32, M32</echo>
          <clickButton label="Antrag genehmigen"/>
          <verifyTitle text="Details zum Antrag"/>
          <verifyText text="Der Antrag wurde erfolgreich genehmigt. Der Antragsteller erhält
hierzu eine entsprechende Info-E-Mail."/>
          <verifyText text="zur Genehmigung (ÖT)"/>
          <echo>##*Logout;;;23;;;Test User;;;Antragsteller;;;Logout durch den
User;;;GP01</echo>
          <clickElement xpath="//html/body/table/tbody/tr/td/div[@id='content-
wrap']/div[@id='content']/div[@id='menumain']/div[4]/form"/>
      </steps>
    </webtest>
  </target>
</project>
```

Danksagung

Für viele nützliche Anregungen und der Opferung ihrer wertvollen Zeit möchte ich mich bei meinen Betreuern Herr Dr.-Ing. Steffen Jurk von der epion GmbH und Frau Prof. Dr.-Ing. Monika Heiner vom Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit (BTU Cottbus) ganz herzlich bedanken.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende wissenschaftliche Arbeit selbständig und ohne unerlaubte Hilfe angefertigt habe, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Cottbus, den 06.10.2009 

Ort, Datum

Unterschrift des Verfassers