FH Technikum Wien

BACHELORARBEIT

Grundlagen des Softwaretests, Testwerkzeuge und Vergleich zweier Webtest-Tools

ausgeführt von Stefan Denner
A-2500 Baden, Hofackergasse 8

Begutachter: Dipl.-Ing. Dr. Gerd Hesina

Baden, 15.04.2009

Ausgeführt an der FH Technikum Wien

Studiengang Informatik

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.“

______________________	___________________________
Ort, Datum	Unterschrift

Abstract

In dieser Arbeit wird die Theorie des Softwaretests, wie sie durch die ISTQB, den DIN- und ISO-Normen sowie durch zahlreiche andere Quellen definiert wird, analysiert und bearbeitet. Es stehen aber nicht nur Normen und Richtlinien sowie Definitionen im Vordergrund, sondern auch Aspekte der Psychologie. Es werden grundlegende Testarten erläutert sowie Methoden des SWT. Es werden die Arten von Test-Tools aufgezählt sowie das Testen im Softwarelebenszyklus und der Gebrauch dieser Tools sowie der Testarten in den einzelnen Phasen des Lebenszyklus' analysiert.

Zum Schluss werden als Praxisbeispiel zwei Open-Source Test-Tools für Webapplikationen, nämlich „Selenium“ und „Canoo WebTest“, miteinander verglichen. Dabei wird auch gleich definiert, was Webtest ist und was es in Hinblick auf die Theorie besonders macht.

Keywords: Softwaretest, SWT, ISTQB, Blackbox, Whitebox, Greybox, funktionaler Test, nicht funktionaler Test, anforderungsbezogener Test, strukturbezogener Test, Test-Tools, Testwerkzeuge, Webtest, Tools für Webapplikationen, Selenium, Canoo WebTest

Danksagung

Ich danke der Firma ANECON für meinen Praktikumsplatz und für die Möglichkeit, mich im Bereich des Softwaretests weiterzubilden. Überdies danke ich all meinen Arbeitskollegen, die ich in den wenigen Monaten meines Praktikums in mein Herz schließen konnte.

Ich danke meiner großen Schwester Sabine. Auch wenn wir einen großen Altersunterschied haben, war sie mir doch immer eine Quelle der Inspiration, des Ansporns und der Liebe.

Ich danke meinen Eltern für ihre Geduld und ihre Unterstützung in meinem Studium, sowohl in finanzieller als auch emotionaler Hinsicht.

Ich danke der Fachhochschule Technikum Wien für meine Ausbildung sowie meinen Studienkollegen und Lektoren für ihre Mühe, ihre Hilfe und ihr Verständnis.

Außerdem möchte ich meinem ehemaligen Informatikprofessor aus dem Gymnasium in Baden, Prof. Mag. Wolfgang Haas, danken. Denn durch ihn habe ich erst mein Interesse an die Informatik entdeckt, welches bis zum heutigen Tage noch weiter gewachsen ist.

Inhaltsverzeichnis

1. Einleitung ... 1
2. Grundlagen des Softwaretests ... 3
2.1. Die Psychologie des Testens ... 3
2.2. Grundbegriffe des Softwaretests ... 4
2.2.1. Der Fehlerbegriff ... 5
2.2.2. Der Testbegriff ... 6
2.2.3. Der Begriff der Softwarequalität ... 7
2.2.4. Der Testaufwand ... 8
2.3. Grundlegende Testarten ... 10
2.3.1. Funktionaler Test ... 10
2.3.2. Nicht funktionaler Test ... 10
2.3.3. Strukturbezogener Test ... 12
2.3.4. Änderungsbezogener Test (Regressionstest) ... 12
2.4. Grundlegende Testmethoden ... 13
2.4.1. Statischer Test ... 13
2.4.2. Dynamischer Test ... 16
2.5. Test-Tools: welche Arten existieren? ... 19
2.5.1. Tools für statische Tests ... 19
2.5.2. Tools für dynamische Tests ... 21
2.5.3. Management-Tools ... 23
2.5.4. Utilities (zusätzliche Hilfsmittel) ... 24
2.6. Testen im Softwarelebenszyklus ... 24
2.6.1. Der Softwarelebenszyklus: das V-Modell ... 24
2.6.2. Analyse der einzelnen Teststufen im V-Modell ... 26
2.7. Fazit zum Softwaretest ... 31
3. Vergleich zweier Open-Source Webtesting-Tools ... 32
3.1. Was ist Webtest? ... 32
3.2. Analyse von „Selenium“ ... 33
3.2.1. Verwendungsweise ... 33
3.2.2. Merkmale des Tools ... 35
3.3. Analyse von „Canoo WebTest“ ... 35
3.3.1. Verwendungsweise ... 35
3.3.2. Merkmale des Tools ... 37

3.4. Fazit des Vergleichs 37
4. Conclusion 38
Literaturverzeichnis 40
Abbildungsverzeichnis 41
Abkürzungsverzeichnis 42

1. Einleitung

Softwaretest ist in der heutigen Zeit, wo Softwareprojekte immer größere Ausmaße annehmen und die Anforderungen der Kunden sowie auch der Konkurrenzdruck zwischen den einzelnen Softwarefirmen immer weiter steigen, ein fester Bestandteil in jeder professionellen und qualitativ hochwertigen Entwicklung. Softwaretest bezieht sich heutzutage nicht mehr nur auf das alleinige Testen der Software auf ihre Funktionalität, sondern beinhaltet zu großen Teilen auch Aspekte, Methoden und Vorgehensweisen der Qualitätssicherung und des Qualitätsmanagements. Die Sicherstellung der Qualität ist wichtig für das Vertrauen des Kunden in das Produkt sowie für das Image der Softwarefirma, denn gute Qualität, egal ob in funktionaler oder nicht funktionaler Hinsicht, führt zur Zufriedenheit des Kunden, welche wiederum zu einer guten Publicity für die Firma selbst führt.

Da Qualität nur subjektiv, also abhängig von einzelnen Personen, betrachtet werden kann, hat jeder der Beteiligten in einem Projekt, sei es der Kunde, der Entwickler oder der Tester, eine eigene Vorstellung bzw. Sichtweise für das Produkt. Um diese Unstimmigkeiten beseitigen zu können, benötigt es Dokumente wie Pflichten- und Lastenhefte, die alle Aspekte des Projektes abzudecken versuchen, damit das Auftreten von Missverständnissen minimiert werden kann. Da in solchen Dokumenten aber immer wieder etwas übersehen werden kann und Widersprüche auftreten können, ist es die Aufgabe der Qualitätssicherung, also des SWT, diese Dokumente zu „reviewen“, um solche „Fehler“ in den Dokumenten zu finden und zu beseitigen. In diesem Fall kann auch vom Testen der Dokumente gesprochen werden (Details hiefür siehe [ISTQB-Syllabus], Kapitel 3.2 „Review process“).

Um die Vorgänge der Qualitätssicherung zu standardisieren und einen Leitfaden zu erhalten, deren Einhaltung einen Nachweis für die Qualität und Professionalität der Firma bzw. des Softwaretesters selbst bedeuten, wurde ein internationaler Standard für Softwaretest erstellt, welcher von dem „International Software Testing Qualifications Board“ (kurz: ISTQB), gepflegt und geschult wird. Jeder professioneller Tester benötigt zumindest ein Zertifikat von der ISTQB, welches er nach einer Schulung und einer erfolgreich abgelegten Prüfung erhält, um in einer seriösen Softwarefirma als Tester angestellt werden zu können. Solch ein Zertifikat zählt als Mindestvoraussetzung für die Einstellung bei den meisten Softwarefirmen.

Zu den Inhalten des Grundzertifikates des ISTQB, auch „Foundation Level“ genannt, zählen nicht nur die theoretischen Grundsätze des SWT, wie sie auch in dieser Arbeit erläutert und erklärt werden, sondern auch Grundlagen des Testmanagements, der Testplanung und Ausführung, welche nicht in dieser Arbeit genauer diskutiert werden.

Im nächsten Kapitel werden die Grundlagen, wie sie im [ISTQB-Syllabus] gelehrt werden, sowie die Grundlagen zu den Testwerkzeugen, erörtert. Dabei wird als erstes die Psychologie des Testens bearbeitet, deren Aspekte mehr Gewichtung besitzen als es auf den ersten Augenblick erscheint.

Danach werden die Grundbegriffe, welche für den Test wichtig sind, darunter der Begriff des Fehlers, der Begriff des Testens, der Begriff der Softwarequalität sowie eine Definition des Testaufwands, genauer definiert und analysiert.

Im vierten Kapitel über die Grundlagen des SWT werden die grundlegenden Testmethoden die dem Softwaretest unterliegen genauer betrachtet. Dazu zählen der funktionale Test, der

nicht funktionale Test, der strukturbezogene Test sowie der änderungsbezogene Test, auch Regressionstest genannt.

Im fünften Kapitel werden die grundlegenden Testarten bearbeitet, zu denen der statische und der dynamische Test gehören. Im statischen Test werden die werkzeugunterstützte Analyse sowie die strukturierten Gruppenprüfungen, auch Reviews genannt, genauer betrachtet. Im dynamischen Test werden das Blackbox und das Whitebox-Verfahren, auch Glassbox- oder Open-Box-Verfahren genannt, bearbeitet, sowie die Mischform namens Greybox-Verfahren.

Daraufhin wird eine Übersicht über die Test-Tools gegeben. Dabei werden diese aufgeteilt in Tools für den statischen Test, Tools für den dynamischen Test, Management-Tools und Utilities, auch zusätzliche Hilfsmittel genannt.

Nachdem die Testwerkzeuge näher betrachtet wurden, wird eine Übersicht zum Softwarelebenszyklus am Beispiel des V-Modells gegeben. Dabei werden die einzelnen Schritte der Softwareentwicklung den dazugehörigen Testschritten gegenübergestellt und der Gebrauch der zuvor vorgestellten Test-Tools in den einzelnen Phasen erläutert.

Zum Abschluss des Grundlagenteils wird noch ein Fazit zum Softwaretest gegeben, bevor der Vergleich der beiden Open-Source Test-Tools „Selenium“ und „Canoo WebTest“ für das Testen von Webapplikationen erfolgt. Dabei wird definiert, was unter dem Begriff Webtest sowie Webapplikation verstanden wird, bevor „Selenium“ und „Canoo WebTest“ zuerst getrennt, und dann im direkten Vergleich, betrachtet werden. Dabei werden die Erkenntnisse aus den Kapiteln zu den Grundlagen des Softwaretests mit eingeflochten und ein angemessener Praxisbezug hergestellt.

2. Grundlagen des Softwaretests

Für ein gutes Allgemeinverständnis für den Softwaretest und dessen Abläufe ist es wichtig die grundlegenden Prinzipien dahinter zu verstehen. Dazu gehören nicht nur das Verstehen der Grundbegriffe und das Wissen derer Definitionen, sondern auch die Analyse der Motivation, die hinter dem Testen steht sowie auch die psychologischen Aspekte, welche für einen Tester mehr Bedeutung haben als es im ersten Augenblick zu erscheinen vermag. Ein Tester hat nämlich in seiner Position sehr viel Einfluss auf seine Entwicklerkollegen, denn jeder Fehler den ein Tester findet, entstand durch die Fehlhandlung eines anderen. Somit können Spannungen entstehen, wenn ein Entwickler einen Fehler von einem Tester „vorgelegt" bekommt, den er selber verursacht hat. Um dies zu verhindern benötigt ein Tester sehr viele „Soft Skills" und darf nicht uneinsichtig gegenüber Fehlern sein, denn jeder macht mal welche, auch ein Tester. Diese Einsicht ist wichtig für die Arbeit eines Softwaretesters. Deswegen beschäftigt sich das erste Kapitel mit den psychologischen Aspekten des Softwaretests bevor die grundlegenden Begriffe, die Testarten, die Testmethoden, die Test-Tools und das Testen im Softwarelebenszyklus genauer betrachtet werden.

2.1. Die Psychologie des Testens

Da Testen Fehler aufdecken soll und diese Fehler von jemand verursacht wurden und überdies noch hinzukommt, dass Menschen ihre eigenen Fehler nur sehr ungern zugeben, besitzt der Softwaretest auch einige Aspekte der Psychologie, welche im folgenden Abschnitt genauer erläutert werden.

Oft werden, wie auch bei [Spillner&Linz05] beschrieben, die Aufgaben der Entwicklung oft als konstruktive und das Testen der Dokumente und der Software als destruktive Tätigkeiten angesehen. Allein schon durch diese Sichtweise entstehen Unterschiede in der Motivation für die eigene Arbeit und die der Kollegen. Diese Ansichtsweise entspricht, auch wenn sie leider sehr oft so gepflegt wird, nicht der Realität. Wie bei [Myers82] auf Seite 16 beschrieben, ist sogar das Gegenteil tat, denn aus dem Englischen übersetzt wird dort folgendes verlautbart: *„Testen ist eine extrem kreative und intellektuell herausfordernde Aufgabe."* Überdies kann Softwaretest nicht als destruktiv angesehen werden, da die Sicherung der Qualität eindeutig zur Verbesserung der Software führt und ein konstruktiver Prozess ist. Dieser Ansicht sind Softwareentwickler aber oft nicht, weil sie sich in ihrer Arbeit eher durch die Tester, wenn sie Fehler entdecken und auf deren Behebung bestehen, gebremst fühlen.

Eine Möglichkeit, um dieser Spannungen zwischen Entwicklern und Testern aus dem Weg zu gehen, ist, wie auch bei [Spillner&Linz05] beschrieben, das Durchführen von Entwicklertests. Hierbei führen die Entwickler selbst die Tests durch und testen ihr selbst erstelltes Programm bzw. Programmstück. In dieser Tatsache liegt auch schon wieder die größte Schwäche des Entwicklertests. Der Autor des Programms ist auch dafür verantwortlich seine eigene Arbeit kritisch zu überprüfen. Da aber nur wenige Menschen in der Lage sind einen angemessenen Abstand zum selbst erschaffenen „Werk" herzustellen, entsteht natürlich ein Konflikt, denn kein Mensch weist gerne seine eigenen Fehler nach. Außerdem liegt es mehr im Interesse des Entwicklers keine Fehlerzustände im eigenen Code zu finden. Zudem steht ein Entwickler seiner eigenen Arbeit meistens zu optimistisch gegenüber, was ihn dazu verleitet, sinnvolle

Testfälle zu vergessen oder nur oberflächlich zu testen, da er sich mehr für das Programmieren interessiert als für das Testen. Überdies kann ein Entwickler auch eine Anforderung falsch verstanden und somit auch falsch programmiert haben, was bei einem Entwicklertest natürlich nicht auffällt. Diese so genannte *„Blindheit gegenüber eigenen Fehlern"* [Spillner&Linz05] kann nur mittels eines eigenen Testteams verhindert werden welches unabhängig von der Entwicklung arbeitet.

Diese Aufteilung führt aber wieder zu den oben genannten Problemen zwischen den Testern und den Entwicklern. Deswegen ist es für einen Tester wichtig, dass dieser ein gewisses Fingerspitzengefühl beim Überbringen von Fehlern mit sich bringt. Ein Tester ist nun mal ein „Überbringer schlechter Nachrichten" und die Art und Weise, wie diese Überbringung erfolgt, beeinflusst die gesamte weitere Zusammenarbeit im Projekt und kann eben förderlich sein oder aber die Zusammenarbeit negativ beeinflussen. Denn wie auch bei [Kaner99] treffend erläutert, ist nicht der beste Tester derjenige, der die meisten Fehler findet oder die meisten Entwickler blamiert, sondern derjenige, der die meisten Fehler behoben bekommt. Und dieses Beheben erfolgt eben nur durch einen Entwickler, der seinen Fehler einsieht und korrigiert, was nur dann möglich ist, wenn der Tester ihn davon überzeugen kann, dass dieser behoben gehört.

Natürlich geht hierbei nichts über die Tatsache, dass ein gegenseitiges Verständnis zwischen Entwickler und Tester die wichtigste Tatsache überhaupt ist. Die Entwickler sollten einsehen, dass das Testen eine wichtige, konstruktive Aufgabe ist und der Qualitätssicherung dient, sowie Tester einsehen müssen, dass Fehler einfach menschlich sind und es keiner Schuldzuweisungen bedarf, denn wie schon das alte lateinische Sprichwort „Errare humanum est" verlautbart ist Irren einfach menschlich.

2.2. Grundbegriffe des Softwaretests

Wie bei [Spillner&Linz05] treffend beschrieben wird, kann der Softwaretest im Allgemeinen mit der Herstellung eines Industrieproduktes verglichen werden. Üblicherweise wird daraufhin kontrolliert, ob die Teil- und Endprodukte den gestellten Anforderungen entsprechen. Hierbei wird geprüft, ob das Produkt auch die geforderte Aufgabe löst, wobei es je nach Produkt unterschiedliche Anforderungen an die Qualität der Lösung geben kann. Erweist sich ein Produkt als fehlerhaft, so müssen gegebenenfalls Korrekturen im Produktionsprozess oder in der Konstruktion erfolgen.

Dieser Vorgang entspricht auch dem Prüfen einer Software, nur mit dem gravierenden Unterschied, dass Software nicht „greifbar" ist und die Prüfung nicht „handfest" vonstatten gehen kann, was die Feststellung von korrekter Funktionalität erschwert [Spillner&Linz05]. Eine optische Prüfung ist im Gegensatz zum Industrieprodukt auch nur beschränkt möglich und besteht bei einer Software im intensiven Lesen der Entwicklungsdokumente und im Durchführen von Code-Reviews bzw. Walkthroughs (Details hiefür siehe Kapitel 2.4.1. Statischer Test, Abschnitt „Strukturierte Gruppenprüfungen".

Somit werden in den folgenden Kapiteln die Grundbegriffe, welche im Softwaretest vorkommen, erläutert und definiert. Dabei werden der Begriff des Fehlers, des Tests, der Softwarequalität und des Testaufwands einer genauen Betrachtung unterzogen.

2.2.1. Der Fehlerbegriff

Wie bei [Spillner&Linz05] beschrieben tritt ein Fehler bzw. ein fehlerhaftes Verhalten eines Systems dann ein, wenn sie nicht anforderungskonform ist. Das heißt, eine Situation kann nur dann als fehlerhaft eingestuft werden, wenn schon im Vorhinein festgelegt wurde, wie die erwartete, korrekte Situation aussehen soll. Ein Fehler ist somit ein „Nichterfüllen“ einer festgelegten Anforderung. Solch eine Anforderung beinhaltet ein „Sollverhalten“, welches in der Spezifikation festgelegt ist und das „richtige“ Verhalten der Software beschreibt. Das „Istverhalten“ hingegen ist die Situation bzw. der Zustand die die Software einnimmt, wenn ein bestimmtes „Sollverhalten“ erreicht werden soll. Ist eine Abweichung bzw. ein Unterschied zwischen dem Ist- und dem Sollverhalten zu erkennen, dann liegt ein Fehler vor.

Des Weiteren gibt es, wie auch bei [Spillner&Linz05] erwähnt, den Begriff des „Mangels“. Dieser beschreibt den Zustand, wenn eine gestellte Anforderung oder ein erwartetes Verhalten nicht hundertprozentig erfüllt wird. Dabei ist der Unterschied zum Fehler derjenige, dass die Funktionalität trotz des gleichzeitigen Daseins einer Beeinträchtigung der Verwendbarkeit oder einer Nichterfüllung einer angemessenen Erwartung gegeben sein kann.

Im Gegensatz zu physikalischen Systemen hat ein Softwaresystem seine Fehler und Mängel von Anfang an inkludiert. Sie entstehen nicht durch Verschleiß oder Alterung wie bei einer Maschine, sondern sind vom Zeitpunkt der Entwicklung an Bestandteil des Programms. Diese Fehler kommen erst bei der Ausführung der Software selbst zum Tragen.

Im Bezug auf Fehler gibt es in der Praxis drei verschiedene Begriffe, welche zu unterscheiden sind und den Begriff des Fehlers verfeinern. Diese Begriffe, die auch bei [Spillner&Linz05] zu finden sind und an der DIN-Norm 66271 angelehnt sind, lauten:

- **Fehlerwirkung (failure)**
  Dieser Begriff beschreibt die Wirkung bzw. das Auftreten eines Fehlers, also die beobachtbare Fehlleistung des Systems beim Ausführen des Programms. Diese Fehlerwirkung wird meistens vom Tester oder vom Anwender während des Betriebes der Software beobachtet bzw. bemerkt. Andere Begriffe hierfür lauten Fehlfunktion, äußerer Fehler oder Ausfall. Der englische Fachbegriff lautet „failure“.

- **Fehlerzustand (fault, bug)**
  Der Fehlerzustand beschreibt den Grund des Auftretens der oben genannten Fehlerwirkung und ist beispielsweise eine falsch programmierte oder vergessene Anweisung im Programm. Einer Fehlerwirkung geht immer ein Fehlerzustand voraus der ihn auslöst. Andere Ausdrücke für den Fehlerzustand wären Defekt oder innerer Fehler. Der englische Fachbegriff lautet „fault“, wobei der Begriff „bug“ hierfür in der Softwareentwicklung gebräuchlicher und bekannter ist.

- **Fehlhandlung (error)**
  Die Fehlhandlung ist die durch eine Person verursachte Fehlleistung die zu einem Fehlerzustand führt, die wiederum zu einer Fehlerwirkung führen kann, vorausgesetzt der Fehler kommt irgendwann (entweder beim Testen oder im Betrieb) zum Tragen. Die Fehlhandlung ist meistens eine durch den Entwickler falsch programmierte oder vergessene Anweisung im Code. Der englische Fachbegriff hierfür lautet „error“.

Zusätzlich zu den drei obigen Begriffen gibt es noch den Begriff der „Fehlermaskierung“. Dieser beschreibt die Tatsache, dass ein Fehlerzustand andere Fehlerzustände „maskieren“, also quasi unsichtbar oder unbemerkt machen kann. Durch die Korrektur eines Fehlerzustandes können somit Seiteneffekte auftreten, welche zu noch mehr Fehlerwirkungen führen. Also das Lösen eines Fehlerzustandes kann nicht nur das gewollte Entfernen einer gewissen Fehlerwirkung haben, sondern auch neue Fehlerwirkungen aufdecken, die wieder anderen Fehlerzuständen zugrunde liegen.

2.2.2. Der Testbegriff

Unter dem Testen von Software wird im Allgemeinen eine stichprobenartige Ausführung des Testobjektes, also der Software selbst oder Teile davon, verstanden. Diese Tests, wo das Programm auch ausgeführt wird, werden „dynamische Tests“ genannt. Im Gegensatz dazu gibt es auch statische Verfahren. Genaueres zum dynamischen und statischen Test wird im Kapitel 2.4. Grundlegende Testmethoden behandelt.

Der Test deckt nur Fehlerwirkungen auf. Das Finden der auslösenden Fehlerzustände und das Beheben der gleichen ist Aufgabe des Entwicklers und wird „Debugging“ genannt, was soviel wie Fehlerbereinigung oder Fehlerkorrektur bedeutet (vgl. vorheriges Kapitel: Begriff „bug“ als Fehlerzustand). Oft wird Debugging mit Testen gleichgestellt, was aber nicht korrekt ist, da völlig verschiedene Motivationen hinter den beiden Aufgaben stehen. Das Ziel des Testens ist das Aufdecken von Fehlerwirkungen, wobei das Ziel des Debuggings das Beheben von Fehlerzuständen ist.

Überdies kann das Ziel des Testens auch variieren. Dieses kann, wie bei [Spillner&Linz05] erläutert, eines der folgenden Ziele sein:

- **Ausführung des Programms mit dem Ziel, Fehlerwirkungen nachzuweisen.**
- **Ausführung des Programms mit dem Ziel, die Qualität zu bestimmen.**
- **Ausführung des Programms mit dem Ziel, Vertrauen in das Programm zu erhöhen.**
- **Analysieren des Programms oder der Dokumente, um Fehlerwirkungen vorzubeugen.**

Dabei ist beim dritten Punkt zu beachten, wie auch bei [Kaner99] festgestellt, dass nie bewiesen werden kann, dass ein Softwareprodukt hundertprozentig fehlerfrei ist. Es kann nur unanfälliger auf Fehler und robuster gemacht werden, was zu einer Steigerung des Vertrauens in die korrekte Funktionsweise des Produktes führt.

Der letzte Punkt ist ein Beispiel für einen statischen Test, da hier das Programm selbst nicht ausgeführt wird. Es werden Analysen sowie Reviews, welche, wie auch im [ISTQB-Syllabus] beschrieben, zu den statischen Tests gehören, durchgeführt und versucht vermeintliche Fehlerwirkungen schon im Vorhinein zu verhindern indem die Fehlerzustände noch vor dem ersten Ausführen der Applikation erkannt und behoben werden.

Zum gesamten Testprozess gehört aber nicht nur das schon oben erwähnte Ausführen des Testobjekts mit Testdaten sondern auch die Planung, Durchführung und Auswertung der Tests, was zum Testmanagement zählt. Ein Testlauf ist ein einmaliges Durchlaufen eines oder mehrerer Testfälle, welche mit Testdaten versehen und unter vorher bestimmten Randbedin-

gungen durchlaufen werden können. Solche Testfälle besitzen gewisse Eingabewerte (die Testdaten) und Ausgabewerte (das Sollverhalten des Testobjekts), welche im Vorhinein festgelegt werden. Die Auswahl dieser Testfälle soll, wie auch bei [Myers82] beschrieben, so getroffen werden, dass die Wahrscheinlichkeit hoch ist, dass sie bisher nicht bekannte Fehlerwirkungen aufdecken können. Diese Testfälle können dann in Testszenarien aneinandergereiht werden, wobei das Ergebnis eines Testfalls als Ausgangssituation des nächsten benutzt wird. Ein gutes Beispiel hierfür ist ein Login in ein System, welches als eigener Testfall definiert werden kann und all die nachfolgenden Testfälle dieses Login voraussetzen (Stichwort: Randbedingungen).

2.2.3. Der Begriff der Softwarequalität

Da Qualität eine subjektive Eigenschaft ist und sich aus der Sicht des einzelnen unterscheiden kann, so kann trotzdem eine Art Regelwerk bzw. eine Definition für Qualität im Bereich Softwareentwicklung erstellt werden. Diese ist in der ISO-Norm 9126 definiert, welche auch bei [Spillner&Linz05] zur Definition der Softwarequalität aufgegriffen wird. Laut dieser Norm lässt sich Qualität in folgende fünf Faktoren unterteilen:

- **Zuverlässigkeit**
  Sie beschreibt die Fähigkeit eines Systems, ein bestimmtes Leistungsniveau unter festen Bedingungen über einen gewissen Zeitraum beizubehalten. Dieser Faktor wird noch unterteilt in:

  - **Reife**
    Darunter wird verstanden, wie oft Fehlerzustände in der Software in einem gewissen Zeitraum ca. vorkommen.

  - **Fehlertoleranz**
    Fehlertoleranz liegt vor, wenn das Softwareprodukt sein Leistungsniveau bewahrt oder wieder erlangt nachdem ein Fehlerzustand aufgetreten ist.

  - **Wiederherstellbarkeit**
    Beschreibt, wie einfach oder schnell bei einem System nach einem Fehler das normale Leistungsniveau wieder erreicht werden kann.

- **Benutzbarkeit**
  Darunter wird verstanden wie einfach die Benutzung, aber auch das Erlernen der Verwendung in Bezug auf verschiedene Benutzergruppen, einer Software ist. Dazu zählen auch Aspekte wie die Einhaltung von Standards, Konventionen und so genannten „Style Guides“. Der gebräuchliche englische Ausdruck hierfür lautet auch „Usability“.

- **Effizienz**
  Durch das Merkmal Effizienz lassen sich messbare Ergebnisse im Bezug auf die Dauer und den Verbrauch von Betriebsmitteln zur Erfüllung einer Aufgabe ermitteln.

- **Änderbarkeit**
  Die Änderbarkeit umfasst die Aspekte der Analysierbarkeit, Modifizierbarkeit, Stabilität und Prüfbarkeit eines Systems im Zuge von Änderungen des selbigen.

- **Übertragbarkeit**
  Die Übertragbarkeit beinhaltet hingegen die Aspekte der Anpassbarkeit, Installierbarkeit, Konformität und Austauschbarkeit. Die Übertragbarkeit ist der Änderbarkeit sehr ähnlich und die Grenzen zwischen den beiden Faktoren verlaufen bei einigen Aspekten sehr fließend, was die Unterscheidung oft erschwert.

Da ein Softwaresystem unmöglich alle Qualitätsmerkmale gleich gut erfüllen kann, muss genau spezifiziert werden, welche Merkmale das Produkt erfüllen soll und welche eher in den Hintergrund gestellt werden. Dabei kann es auch möglich sein, dass die Erfüllung eines Qualitätsmerkmales ein gleichzeitiges Nichterfüllen eines anderen nach sich zieht. So kann eine Effizienzsteigerung bewirkt werden indem eine Besonderheit einer Plattform ausgenutzt wird, was aber gleichzeitig den Aspekt der Übertragbarkeit verschlechtert [Spillner&Linz05].

2.2.4. Der Testaufwand

Wenn ein Tester keine Fehler mehr aufdecken kann, dann bedeutet dies nicht, dass das Testobjekt fehlerfrei ist. Wie bei [Siteur05] treffend beschrieben ist Softwaretest wie Fischen, nur weil man keine Fische mehr fängt heißt das noch lange nicht, dass keine Fische mehr im Wasser sind. Es ist einfach unmöglich ein Programm vollständig zu testen, wie es auch bei [Kaner99] erläutert wird. Um eine vollständige Fehlerfreiheit nachweisen zu können, müsste das Programm in allen möglichen Situationen, mit allen möglichen Eingaben und unter Berücksichtigung aller unterschiedlichen Randbedingungen getestet werden. Solch ein vollständiger Test ist praktisch undurchführbar [Spillner&Linz05]. Ein Austesten aller kombinatorischen Möglichkeiten würde zu einer fast unbegrenzten Anzahl von Tests führen, welche ein Tester nie komplett ausführen könnte. Überdies gibt es immer Möglichkeiten, Schlupflöcher, Randbedingungen ö.ä. die ein Tester bzw. ein Testdesigner nicht bedacht hat und das vollständige Testen noch unmöglicher macht. Außerdem ist die Anzahl der möglichen Programmpfade bei größeren Applikationen so gigantisch groß, dass das gesamte Testen viel zu viel Zeit in Anspruch nehmen würde, was in einem Projekt, welches Zeit und Geld kostet, natürlich inakzeptabel ist.

Ein Beispiel für solch eine „gigantisch große Anzahl“ und für die „Zeit zum kompletten Testen“ ist bei [Kaner99] zu finden. Dort wird ein Programm von Myers aus dem Jahr 1979 beschrieben, welches nur aus einer Schleife und ein paar Verzweigungen (if-Statements) besteht. Solch ein Programm wäre mit 20 Zeilen Code realisierbar und hätte trotzdem umgerechnet an die 100 Billionen(!) verschiedener Pfade. Ein schneller Tester würde für das Testen all dieser Pfade ca. 1 Milliarde(!) Jahre benötigen.

Somit stellt sich die Frage, wie viel Aufwand aufgebracht bzw. wie genau in einem Projekt getestet werden sollte und auf was die Tests abzielen sollten, damit die Qualität der Software so gut wie möglich sichergestellt werden kann. Eine allgemeingültige Höhe für den Test-aufwand kann nicht aufgestellt werden, da dieser sehr stark vom Charakter des Projektes abhängt. Im Allgemeinen kann aber gesagt werden, dass der Aufwand durchschnittlich zwischen 25 und 50 Prozent des gesamten Projektaufwandes ausmacht [Spillner&Linz05]. Dabei gibt es aber auch Projekte, wie z.B. die Jahr-2000-Umstellung, wo der Aufwand für Softwaretests noch höher werden kann, nämlich bis zu 80 Prozent.

Der Aufwand steht immer im Zusammenhang mit dem Risiko, welches dem System bei einem Fehlerfall unterliegt und die Höhe der Kosten, welche beim Auftreten eines Fehlers entstehen. Dabei ist abzuwägen, wie hoch der Testaufwand betrieben werden soll mit Bedacht auf dem Risiko und den Kosten, mit denen ein Fehler behaftet ist. Ein hohes Risiko besteht z.B. bei einem sicherheitsrelevanten System, wo Datenverlust oder unerlaubte Weitergabe von Daten zu großen Schäden führen kann. Die Tests für solche Systeme müssen ausgiebiger und gezielter durchgeführt werden als für Systeme mit einem geringeren Risiko. Im Bezug auf die Kosten kann, wie bei [Pol00] nachzulesen ist, festgestellt werden, dass Testen nur dann ökonomisch sinnvoll ist, wenn die Kosten für das Finden und Beheben von Fehlern im Test niedriger sind als die, die mit dem Auftreten eines Fehlers bei der Nutzung verbunden sind. Ein gutes Beispiel hierfür ist aus einem Zeitungsartikel der „Frankfurter Allgemeinen Zeitung“ vom 17.01.2002 zitiert bei [Spillner&Linz05] mit dem Titel „IT-Systemausfälle kosten viele Millionen“, wo ein einstündiger Systemausfall im Börsenhandel mit Kosten in der Höhe von 7,8Mio US-$ veranschlagt wird. Somit sollte der Testaufwand nicht höher sein als diese 7,8Mio US-$, aber auch nicht viel niedriger, da das System, was hinter dem Börsenhandel steht, extrem risikobehaftet ist und das Übersehen eines Fehlers eben diesen Millionenbetrag ausmachen kann, wenn er zum Tragen kommt.

Ein weiterer Aspekt zu den Kosten liegt in der Tatsache, dass Fehler umso weniger kosten, umso früher sie im Entwicklungsprozess gefunden werden. Wie in der Abbildung 1, die die so genannte Boehm'sche Kurve zeigt, zu erkennen ist, steigen die Kosten zur Behebung eines Fehlers stark an umso später dieser im Projekt entdeckt wird. Diese Kurve wurde vor über 25 Jahren erstellt und ist bis heute noch immer aktuell.

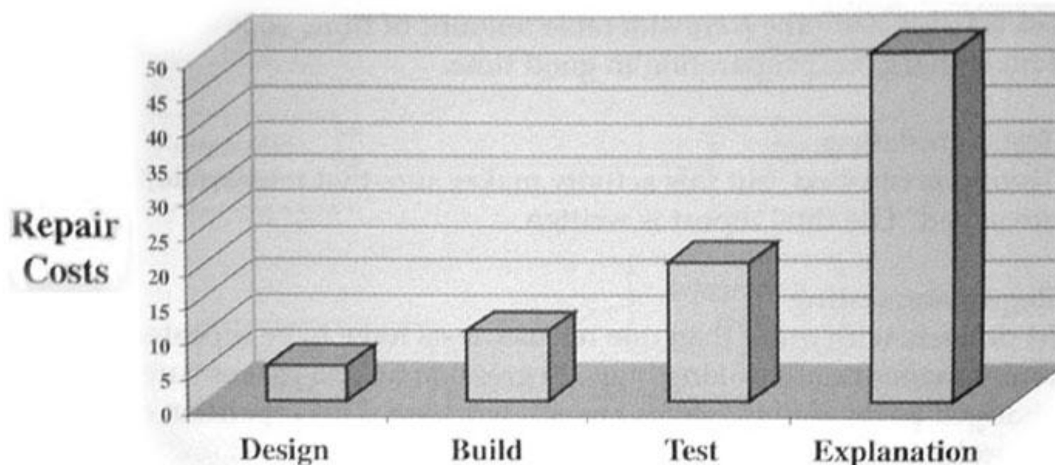

Abbildung 1: Die Kosten eines Fehlers im Entwicklungsprozess (Boehm'sche Kurve) [Siteur05].

2.3. Grundlegende Testarten

Nachdem in den vorherigen Kapiteln die grundlegenden Begriffe des Softwaretests und die Motivation des Testens sowie auch die Psychologie dahinter erläutert wurden, werden in diesem Kapitel der Fokus und die Ziele des Softwaretests und deren Variationen betrachtet. Aus diesem Grund werden in den folgenden Abschnitten die verschiedenen Testarten und deren Verwendung behandelt und analysiert. Laut [Spillner&Linz05] können folgende grundlegenden Testarten unterschieden werden:

- **funktionaler Test**
- **nicht funktionaler Test**
- **strukturbezogener Test**
- **änderungsbezogener Test (Regressionstest)**

2.3.1. Funktionaler Test

Diese Testart beschreibt alle Testmethoden die dem „Blackbox-Verfahren“ (Details hiefür siehe Kapitel 2.4.2. Dynamischer Test) angehören und überprüft das Testobjekt auf derer von außen sichtbaren Ein- und Ausgabeverhalten. Als Basis für den funktionalen Test dienen die so genannten funktionalen Anforderungen, welche das Verhalten des Systems spezifizieren und auch beschreiben, was genau das System zu leisten hat. Merkmale für die Funktionalität nach [Spillner&Linz05] zitiert aus der ISO-Norm 9126 sind Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit und Sicherheit. Diese Anforderungen werden zu Beginn des Projektes in Anforderungsdokumenten wie Lasten- oder Pflichtenhefte oder in einem Anforderungsmanagement-Werkzeug gesammelt und verwaltet.

Eine funktionale Anforderung wird in der Regel nicht durch einen einzigen Testfall sondern durch das Zusammenspiel mehrerer getestet. Sind diese Testfälle zumindest einmal erfolgreich durchlaufen worden, dann wird auch die Anforderung als validiert betrachtet. Diese Testart wird vorzugsweise in einem späteren Entwicklungsprozess wie z.B. dem System- oder Abnahmetest (Details hiefür siehe Kapitel 2.6. Testen im Softwarelebenszyklus) verwendet. Des Weiteren können die funktionalen Anforderungen auch direkt auf Geschäftsprozesse des Kunden abzielen und eine systemnahe Verwendung simulieren. Solche mit dem funktionalen Test verwandte Testmethoden werden auch anwendungsfallbasiert oder geschäftsprozessbasiert genannt [Spillner&Linz05].

2.3.2. Nicht funktionaler Test

Im Gegensatz zu den funktionalen Anforderungen, welche das Verhalten des Systems, also was es genau tut, analysiert, beschäftigt sich der nicht funktionale Test mit dem „Wie“, also mit welcher Qualität und „wie gut“ das System arbeitet. Diese Anforderungen beeinflussen stark die Zufriedenheit des Kunden und die Tatsache, wie gerne der Anwender das Produkt verwendet. Die Merkmale für die nicht funktionalen Anforderungen sind laut [Spillner&Linz05] zitiert aus der ISO-Norm 9126 Zuverlässigkeit, Benutzbarkeit und Effizienz.

Das Problem bei nicht funktionalen Anforderungen ist, dass sie nur schwer in messbare Werte umformuliert werden können. Die Formulierungen wie „Das System muss schnell sein“ oder „schnell reagieren“ sind in dieser Form nicht testbar. Des Weiteren gibt es Anforderungen, die als so selbstverständlich gelten, dass sie nicht in Anforderungsdokumente erwähnt werden müssen. Aber auch diese nicht funktionalen Anforderungen gilt es zu überprüfen.

Um nicht funktionale Anforderungen überprüfen zu können existieren mehrere Testarten, die auf eine gewisse Anforderung bzw. Systemeigenschaft abzielen. Laut [Myers82] können folgende unterschieden werden:

- **Lasttest**
  Dieser Test misst das Systemverhalten in Abhängigkeit von steigender Benutzer- oder Transaktionszahl. Dabei wird die Last nur bis zum höchsten Punkt (Anzahl der User, usw.), der spezifiziert wurde, erhöht.

- **Performanztest (Performance Test)**
  Hier werden die Verarbeitungs- und Antwortzeiten des Systems nach der Spezifikation überprüft, wobei dies in der Regel bei steigender Systemlast geschieht.

- **Volumen-/Massentest**
  Hierbei wird das Systemverhalten unter Verwendung großer Datenmengen betrachtet.

- **Stresstest**
  Ist im Gegensatz zum Lasttest ein Test, der die Last versucht bis über die Grenzen zu steigern und somit das Systemverhalten bei Überlastung überprüft.

- **Test auf Sicherheit (Security-Test)**
  Hierbei wird überprüft, ob der Zugriff auf Daten von außen, der nicht stattfinden darf, auch verhindert wird und ob es Sicherheitslücken im System gibt.

- **Test der Stabilität/Zuverlässigkeit**
  Hierbei wird das System im Dauerbetrieb beobachtet und die Anzahl der Ausfälle pro Betriebsstunde errechnet.

- **Test auf Robustheit**
  Dieser Test beobachtet das Verhalten des Systems gegenüber Fehlbedienung, Fehlprogrammierung und Hardwareausfall sowie das Wiederanlaufverhalten (recovery).

- **Test auf Kompatibilität/Datenkonversion**
  Hierbei wird die Verträglichkeit mit anderen Systemen sowie der Import und Export von Datenbeständen überprüft.

- **Test unterschiedlicher Konfigurationen**
  Das System wird z.B. unter verschiedenen Betriebsystemen, Landessprachen und Hardwareplattformen überprüft.

- **Test auf Benutzerfreundlichkeit (Usability-Test)**
  Dies beinhaltet die Prüfung der Erlernbarkeit sowie der Angemessenheit der Bedienung der Software bezogen auf die Bedürfnisse einer bestimmten Anwendergruppe.

- **Test der Dokumentation**
  Überprüfung und Vergleich des Verhaltens des Systems mit den Informationen in der Dokumentation.

- **Test auf Änderbarkeit/Wartbarkeit**
  Dieser Test beinhaltet die Überprüfung der Verständlichkeit und Aktualität der Entwicklungsdokumente, der Systemstruktur, usw.

2.3.3. Strukturbezogener Test

Der strukturbezogene Test beinhaltet, im Gegensatz zu den funktionalen und nicht funktionalen Testarten, die Methoden des „Whitebox-Verfahrens“ (Details hiefür siehe Kapitel 2.4.2. Dynamischer Test). Die innere Struktur des Testobjekts steht im Mittelpunkt. Analysiert werden z.B. der Kontroll- oder der Datenfluss des Testobjektes, wobei es hier das Ziel ist, möglichst alle Elemente der betrachteten Struktur abzudecken bzw. einmal getestet zu haben [Spillner&Linz05].

Strukturelle Tests werden vorwiegend in frühen Phasen der Softwareentwicklung wie z.B. dem Komponenten- und Integrationstest (Details hiefür siehe Kapitel 2.6. Testen im Softwarelebenszyklus) eingesetzt.

2.3.4. Änderungsbezogener Test (Regressionstest)

Änderungsbezogene Tests bzw. Regressionstests sind vor allem dann notwendig, wenn ein oder mehrere Teile eines Systems durch Wartungsarbeiten oder Weiterentwicklung geändert oder ergänzt wurden. Der Regressionstest ist somit ein Test eines bereits getesteten Programms oder Systems nach einer Modifikation um nachzuweisen, dass durch die Änderung keine neuen Defekte eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden [Spillner&Linz05]. Solche Tests finden in allen Stufen des Entwicklungsprozesses statt und umfassen funktionale, nicht funktionale und strukturelle Tests. Damit diese Tests, da sie ja öfters ausgeführt werden, wiederholbar sind, müssen sie ausreichend dokumentiert sein. Deswegen sind Testfälle für den Regressionstest auch bevorzugte Kandidaten für die Testautomatisierung.

Die Frage die sich bei Regressionstests stellt ist, was aufgrund von Änderungen getestet werden soll. Hierbei lassen sich folgende Möglichkeiten abgrenzen [Spillner&Linz05]:

1. *Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben, deren Ursache der (nun korrigierte) Defekt war (Fehlernachtest).*
2. *Test aller Programmstellen, an denen korrigiert oder geändert wurde (Test geänderter Funktionalität).*
3. *Test aller Programmteile oder Bausteine, die neu eingefügt wurden (Test der neuen Funktionalität).*
4. *Das komplette System (vollständiger Regressionstest).*

Die ersten drei Möglichkeiten, also der Fehlernachtest und die Tests am Ort der Modifikation, sind für die Sicherstellung der Funktionalität zu wenig. Die Funktionalität nach einer Änderung kann nur mittels eines vollständigen Regressionstests passieren, denn teilweise kann eine simple lokale Änderung im System unerwartete Auswirkungen und Seiteneffekte auf beliebige (auch weit entfernte) Systemteile haben [Spillner&Linz05].

Da ein vollständiger Regressionstest in der Regel zu zeitaufwendig und kostenintensiv ist, werden Kriterien gesucht anhand deren entschieden werden kann, welche Testfälle wirklich benötigt werden und welche weggelassen werden können. Wie immer beim Testen ist hierbei eine Abwägung zwischen niedrigeren Kosten und höheres Risiko notwendig. Laut [Spillner&Linz05] können folgende Auswahlkriterien angewendet werden:

- *Wiederholung nur von denjenigen Tests aus dem Testplan, denen hohe Priorität zugeordnet ist.*
- *Bei funktionalen Tests Verzicht auf gewisse Varianten (Sonderfälle).*
- *Einschränkung der Tests auf bestimmte Konfigurationen (z.B. nur Test der englischsprachigen Produktversion, nur Test auf einer bestimmten Betriebs-systemversion u.Ä.).*
- *Einschränkung der Tests auf bestimmte Teilsysteme oder Teststufen.*

Die Auswahl der Kriterien für die Ausführung des Regressionstests benötigt vieles an Erfahrung als Softwaretester. Solche Entscheidungen können nicht durch theoretische oder systematische Denkweisen getroffen werden sondern müssen mit dem „Bauchgefühl“ ent-schieden werden.

2.4. Grundlegende Testmethoden

Im vorhergehenden Kapitel wurden die grundlegenden Testarten des Softwaretests analysiert. Im nächsten Abschnitt wird eine Übersicht über die Testmethoden gegeben, die für den Softwaretest und die Erstellung von Testfällen wichtig sind und ein systematisches Testen erst voraussetzen und möglich machen. Hierbei können zwei grundlegende Testarten unter-schieden werden. Es gibt den statischen Test, welcher auch statische Analyse genannt wird und des Öfteren schon in den vorhergehenden Kapiteln erwähnt wurde, und den dynamischen Test, welcher die Ausführung der zu testenden Applikation voraussetzt, was den Kern des Softwaretests ausmacht. In den folgenden Kapiteln werden diese beiden Testarten genauer erläutert, unterschieden und die darin beinhaltenden Methoden erklärt und analysiert.

2.4.1. Statischer Test

Unter einem statischen Test bzw. einer statischen Analyse wird eine Überprüfung der Dokumente oder des Testobjektes ohne dessen Ausführung verstanden. Dabei können genau zwei Gruppen des statischen Tests unterschieden werden:

- **Werkzeugunterstütze Analyse**
- **Strukturierte Gruppenprüfungen (Reviews)**

Details und Eigenschaften zu den beiden statischen Testgruppen werden in den nächsten beiden Kapiteln erläutert.

Werkzeugunterstützte Analyse

Bei den werkzeugunterstützten Verfahren übernehmen Tools die Analyse des Programms. Ein Compiler z.B. kann während des Kompilierens schon Informationen über Anomalien, Fehler oder Warnungen bereitstellen. Beispiele hierfür sind „unerreichbarer Code“ und falsch oder nicht verwendete Variablen. Es gibt auch Programme, so genannte Analysatoren, welche noch mehr Aufgaben übernehmen können als ein Compiler. Mittels Analysatoren kann z.B. überprüft werden, ob Konventionen oder Codierstandards eingehalten werden. Außerdem können Analysen bezüglich des Datenflusses eines Programms erstellt werden, welche Anomalien aufdecken können. Solche „Datenflussanomalien“ zeigen hierbei keine expliziten Fehler auf, sondern nur potentielle Fehlerquellen aufgrund von ungewöhnlichen Datenflüssen bzw. Verwendungsweisen von Variablen.

Zusätzlich zur Datenflussanalyse gibt es noch die Kontrollflussanalyse, welche mit Hilfe eines Kontrollflussgraphen erfolgt. Ein solcher Kontrollflussgraph ist ein Graph, der alle möglichen Abläufe eines Programms aufzeigt und aus Knoten und Kanten besteht, wobei ein Knoten eine Verzweigung im Codestück und eine Kante eine Verbindung zur nächsten Verzweigung darstellt. Dabei kann solch eine Verzweigung z.B. als eine Entscheidung (if-Statement), eine Schleife oder als ein Rücksprungpunkt im Code angesehen werden. Ein Beispiel für solch einen Kontrollflussgraphen ist in Abbildung 2 zu erkennen.

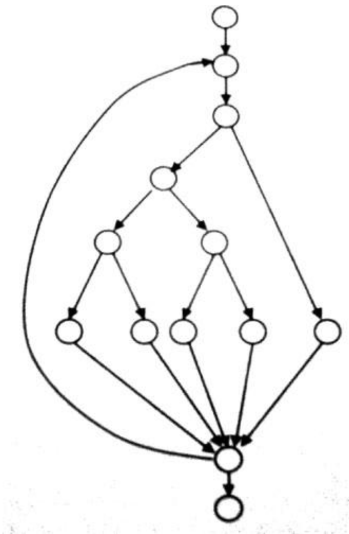

Abbildung 2: Ein Beispiel eines Kontrollflussgraphen [Spillner&Linz05].

Strukturierte Gruppenprüfungen (Reviews)

Im Gegensatz zur werkzeugunterstützten Analyse sind bei strukturierten Gruppenprüfungen nur Personen und ihre menschliche Analyse- und Denkfähigkeit beteiligt. Dabei handelt es sich um das Analysieren eines Programmstückes oder eines Dokuments durch eine oder mehrerer Personen, was auch Review genannt wird. Reviews sind ein gutes Mittel zur

Sicherung der Qualität und können Fehler entdecken noch bevor die erste Zeile programmiert und der erste Test ausgeführt wird. Wie schon im Kapitel 2.2.4. Der Testaufwand erkannt wurde, ist die Fehlerbehebung umso kostengünstigster, desto früher sie im Projektablauf passiert (Stichwort: Boem'sche Kurve, Abbildung 1). Durch Reviews kann dies sehr gut erreicht werden.

Bei einem Review werden Dokumente, Sourcecode o.ä. analysiert und in der Gruppe kritisch betrachtet. Solch ein Review kann von seiner Art her zwischen sehr formell bis völlig informell variieren. Den einzelnen Mitwirkenden in einem Review werden bestimmte Rollen zugewiesen. Diese haben klar definierte Aufgaben und sind je nach Formalitätsgrad verschieden streng einzuhalten. Außerdem gibt es verschiedene Arten von Reviews, welche sich in ihren Zielen, ihren Durchführungsgründen und auch in ihrer Formalität unterschieden. Im Gegensatz zu einem informellen Review, welches keiner geregelten Prozesse unterliegt und auf gesundem Menschenverstand und einer guten Zusammenarbeit basiert, können folgende formelle Reviewarten unterschieden werden [ISTQB-Syllabus]:

- **Walkthrough**
  Ein Walkthrough wird vom Autor des Dokumentes geleitet, welcher Erklärungen abgibt und die Beteiligten gegebenenfalls Fragen einwerfen. Ein Walkthrough hat in der Regel ein offenes Ende. Das Ziel soll sein, dass die Beteiligten über das reviewte Dokument, Codestück o.ä. mehr lernen und mehr Verständnis erlangen. Hierbei werden auch sehr leicht Fehler gefunden, welche dem Autor selbst nicht aufgefallen sind und besser von den Außenstehenden entdeckt werden.

- **Technisches Review**
  Ein technisches Review ist ein dokumentierter und definierter Fehlerfindungsprozess der technische Experten beinhaltet und idealer Weise von einem trainierten Moderator geleitet wird, der im Gegensatz zum Walkthrough nicht der Autor sein darf. Es werden optional Checklisten, Reviewreports und Listen mit Anforderungen verwendet. Das Management kann auch an solch einem technischen Review teilnehmen. Die Hauptziele sind: diskutieren, Entscheidungsfindung, Alternativen evaluieren, Fehler finden, technische Probleme lösen und das Überprüfen der Einhaltung von Spezifikationen und Standards.

- **Inspektion**
  Eine Inspektion wird von einem trainierten Moderator geführt, der wie beim technischen Review nicht der Autor ist. Es gibt klar definierte Rollen und Metriken. Eine Inspektion ist die formalste Reviewart und basiert auf Regeln und Checklisten mit Eingangs- und Ausgangskriterien. Zusätzlich gibt es einen Inspektionsreport und eine Liste von Ergebnissen am Ende der Sitzung. Das Ziel einer Inspektion ist das Finden von Fehlern.

Es gibt viele verschiedene Möglichkeiten ein Dokument o.ä. zu reviewen. Dabei verlaufen die Grenzen zwischen den Reviewarten, sowie auch den einzelnen Rollen, sehr fließend und können, wie bei [Spillner&Linz05] erläutert, firmen- und projektspezifische Ausprägungen annehmen. Die Reviews werden den jeweiligen Bedürfnissen und Anforderungen der Projekte angepasst, wobei sich diese Anpassungen wiederum positiv auf die Wirkung der Reviews zur Qualitätssicherung auswirken.

2.4.2. Dynamischer Test

Bei einem dynamischen Test handelt es sich, wie auch schon in den vorhergehenden Kapiteln erläutert, um einen Test, der die Ausführung des Testobjektes auf einem Rechner voraussetzt. Hierfür wird natürlich vorausgesetzt, dass ein lauffähiges Programm vorhanden ist, welches für den Test ausgeführt wird. Da dies aber in den unteren Stufen der Softwareentwicklung, wie z.B. beim Komponenten- und Integrationstest (Details hierzu siehe Kapitel 2.6. Testen im Softwarelebenszyklus) noch nicht möglich ist, muss das Testobjekt in einen so genannten „Testrahmen" eingebettet werden. Eine visuelle Darstellung für das Prinzip des Testrahmens ist in der Abbildung 3 zu erkennen.

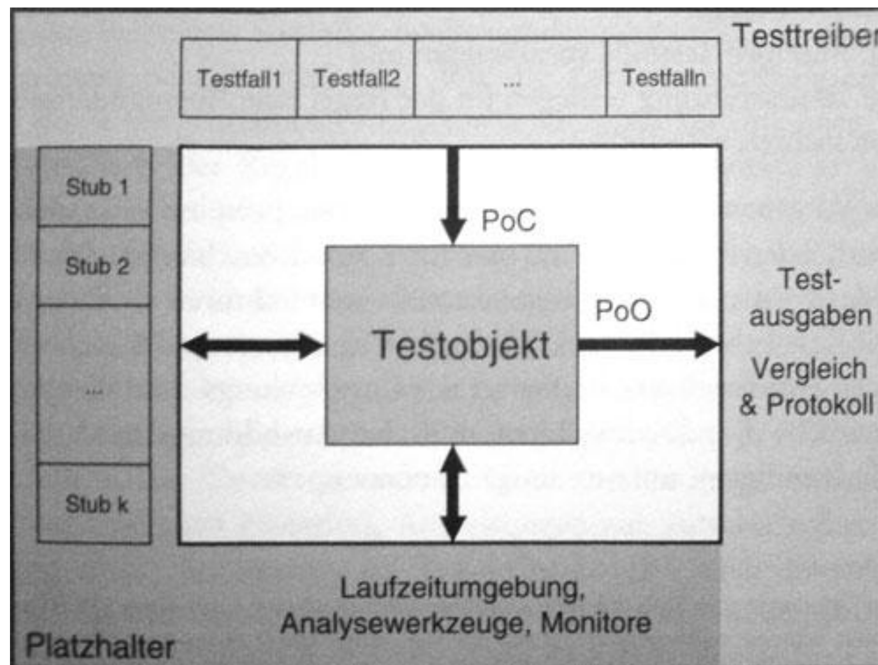

Abbildung 3: Der Testrahmen um ein Testobjekt [Spillner&Linz05].

Wie bei [Spillner&Linz05] erklärt, wird das Testobjekt in diesen Testrahmen eingebettet, welches die Programmteile simuliert, die über definierte Schnittstellen das Testobjekt aufrufen sowie die Teile, die vom Testobjekt selbst aufgerufen werden. Die Teile, die das Testobjekt aufrufen, werden auch Testtreiber genannt. Sie versorgen das Testobjekt mit Daten und simulieren die späteren Programmteile, die das Objekt später aufrufen werden. Diese Schnittstelle zwischen den Testtreibern und dem Testobjekt wird auch „Point of Control" (kurz: PoC) genannt. Die Programmteile, die das Testobjekt aufruft, werden Platzhalter oder auch Stellvertreter genannt. Weitere Ausdrücke hierfür können auch „Stub", „Dummy" und „Mock" sein. Die Ausgabe der Testergebnisse und des Vergleiches der Soll- und Istverhalten des zu testenden Objektes findet, wie auch in Abbildung 3 ersichtlich, über den „Point of Observation" (kurz: PoO) statt.

Dieser Rahmen ermöglicht es eine Softwarekomponente zu testen noch bevor die über- und untergeordneten Programmteile mit ihrer kompletten Funktionalität bereitstehen. Somit kann verhindert werden, dass der Test durch Zeitrückstände in der Entwicklung aufgehalten wird. Der Softwaretest kann unabhängiger von der Entwicklung ablaufen.

Für die Erstellung der Testfälle für das Testobjekt im Testrahmen gibt es eine Reihe unterschiedlicher Verfahren, die verwendet werden können. Diese Verfahren können in zwei große Gruppen aufgeteilt werden. Diese beiden Gruppen sind, wie auch bei [Spillner&Linz05] beschrieben:

- **Blackbox-Verfahren**
- **Whitebox-Verfahren (Glassbox- oder Open-Box-Verfahren)**

Zusätzlich gibt es auch eine Mischform der beiden Verfahren welches Greybox-Verfahren genannt wird. Dieses Verfahren sowie auch die Black- und Whitebox-Verfahren werden in den nächsten Kapiteln genauer erläutert.

Blackbox-Verfahren

Beim Blackbox-Verfahren wird das Testobjekt als schwarzer Kasten angesehen. Über den internen Aufbau und die Abläufe sind keine Informationen vorhanden und sind auch nicht notwendig. Das Verhalten des Testobjektes wird von außen beobachtet, d.h. der „Point of Observation“ PoO liegt außerhalb des Testobjekts. Außerdem ist eine Steuerung des Ablaufs des Testobjekts außer durch die Wahl der Eingabetestdaten nicht möglich, womit auch der „Point of Control“ PoC außerhalb des Testobjektes liegt. Dies kann in Abbildung 4 genauer beobachtet werden.

Aus diesem Grund werden die Testfälle aus der Spezifikation abgeleitet oder liegen schon als Teil derselben vor. Deswegen werden das Blackbox-Verfahren auch funktionale oder spezifikationsbasierte Testverfahren genannt.

Da ein vollständiger Test mit allen möglichen Eingabedaten und Kombinationen unmöglich durchzuführen ist (siehe Kapitel 2.2.4. Der Testaufwand), müssen Methoden verwendet werden um die Testdaten auf ein sinnvolles Minimum zu reduzieren. Dabei sind systematische Vorgehensweisen für die Auswahl der minimalen Anzahl der notwendigen Testfälle zu bevorzugen.

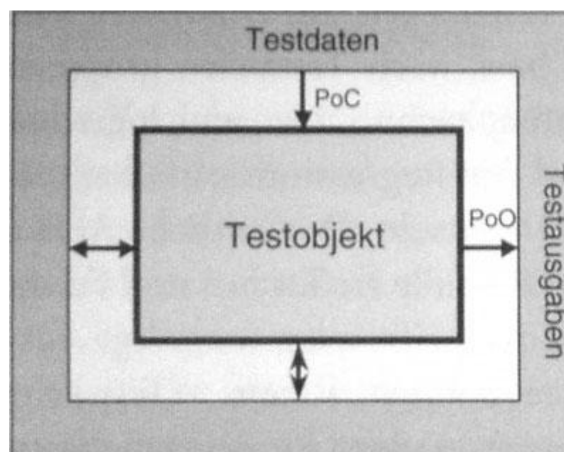

Abbildung 4: Testrahmen beim Blackbox-Verfahren [Spillner&Linz05].

Das Blackbox-Verfahren eignet sich gut um die Funktionalität zu überprüfen, wobei falsch spezifizierte Funktionalitäten nicht erkannt werden können. Es wird nach der falschen Spezifikation getestet und nur durch Menschenverstand können diese Fehler korrigiert werden. Überdies können auch nicht geforderte Funktionalitäten im Testobjekt vorhanden sein, welche über die Spezifikation hinausgehen. Diese können mittels Blackbox-Verfahren nicht erkannt und getestet werden. Hiefür ist das im nächsten Kapitel vorgestellte Whitebox-Verfahren geeignet.

Whitebox-Verfahren (Glassbox- oder Open-Box-Verfahren)

Bei den Whitebox-Verfahren wird das Testobjekt als durchsichtiger Kasten angesehen. Der interne Aufbau, also der Programmtext, ist bekannt und wird auch für die Erstellung der Testfälle herangezogen. Wie beim Blackbox-Verfahren wird das Testobjekt in einen Testrahmen eingespannt, wobei im Gegenteil zum Blackbox-Verfahren der „Point of Control“ PoC und der „Point of Observation“ PoO innerhalb des Testobjektes liegen. Diese Tatsache wird in der Abbildung 5 ersichtlich.

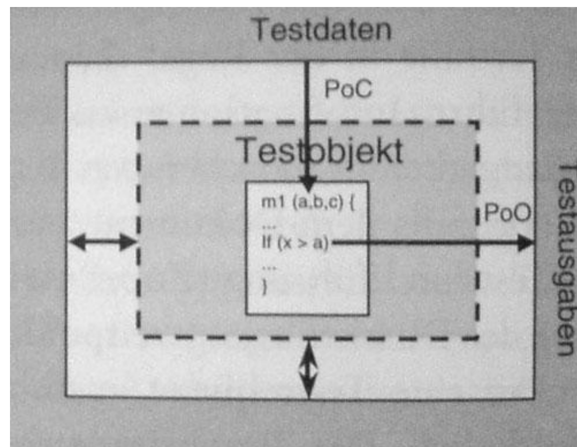

Abbildung 5: Testrahmen beim Whitebox-Verfahren [Spillner&Linz05].

Die grundlegende Idee des Whitebox-Verfahrens ist, wie auch bei [Spillner&Linz05] beschrieben, dass alle Quellcodeteile eines Testobjekts mindestens einmal ausgeführt werden sollen. Deswegen wird das Whitebox-Verfahren auch codebasiertes Testverfahren bezeichnet. Aufgrund der Programmlogik werden ablauforientierte Testfälle ermittelt und ausgeführt. Dabei wird auch die Spezifikation berücksichtigt um die Ausgabe bzw. das Verhalten des Testobjektes als korrekt oder fehlerhaft zu bewerten. Bei der Testfallerstellung selbst steht die Spezifikation, im Gegensatz zum Blackbox-Verfahren, aber nicht im Mittelpunkt.

Greybox-Verfahren

Da Whitebox- und Blackbox-Verfahren ihre Vor- und Nachteile besitzen, welche sich gegenseitig aufheben, ist es ratsam eine Kombination aus beiden zu praktizieren, um die Fehlerfindung so effizient wie möglich zu machen. Dabei werden die Techniken des Blackbox-

Verfahrens mit dem Verständnis der inneren Struktur des Testobjekts angewendet, was in der Regel zu einer besseren Testarbeit führt. Dieses vermischte Verfahren wird auch Greybox-Testing genannt und findet vor allem bei Webapplikationen oft gebrauch, da hier das Zusammenspiel mehrerer Systeme im Vordergrund steht und ein Verständnis für die Hintergründe dem Testen sehr zugute kommt und hilfreich ist [Nguyen03].

Beim Greybox-Verfahren handelt es sich weniger um eine Methode als mehr um eine Einstellung des Testers. Der Tester sollte in der Lage sein einen Blackbox-Test durchzuführen und durch das Verständnis der inneren Funktion des Testobjektes dabei so zielgerichtet zu arbeiten, dass die Fehlerfindungsrate gesteigert wird und die möglichen Schwachstellen des Testobjektes schon im Vorhinein erkannt werden. Um dies wirklich gut durchführen zu können benötigt es sehr viel an Erfahrung und vor allem an Verständnis und Kenntnisse im Bereich der Softwareentwicklung. Deswegen sind für einen Softwaretester Programmierkenntnisse meistens von Vorteil, zumindest müssen sie technisch versiert sein.

2.5. Test-Tools: welche Arten existieren?

Da in dieser Arbeit nicht nur die Grundlagen des Softwaretests zur Debatte stehen sondern auch später zwei Test-Tools verglichen werden sollen, wird in den nächsten Kapiteln eine Übersicht über die Arten von Test-Tools gegeben. Überdies werden der Sinn und die Gebrauchsgebiete der einzelnen Arten analysiert. Später werden die einzelnen Tools sowie auch die Testarten den einzelnen Phasen des Softwarelebenszyklus (siehe Kapitel 2.6. Testen im Softwarelebenszyklus) zugewiesen und deren Sinn und Gebrauch betrachtet.

Laut [Siteur05] können folgende Arten von Test-Tools unterschieden werden:

- **Tools für statische Tests**
- **Tools für dynamische Tests**
- **Management-Tools**
- **Utilities (zusätzliche Hilfsmittel)**

Ein Überblick über die Arten von Test-Tools und die untergeordneten Spezialisierungen können in Abbildung 6 in einer Baumstruktur gesehen werden.

In den folgenden Kapiteln werden die Testarten und die Spezialisierungen genauer erläutert.

2.5.1. Tools für statische Tests

Statische Prüfungen können, wie auch schon im Kapitel 2.4.1. Statischer Test erwähnt, an Sourcecode und an Spezifikationen durchgeführt werden. Tools können dabei helfen Fehler und Unstimmigkeiten schon früh zu erkennen.

Die Test-Tools, die laut [Siteur05] für statische Tests Verwendung finden, werden in den nächsten Abschnitten genauer erläutert.

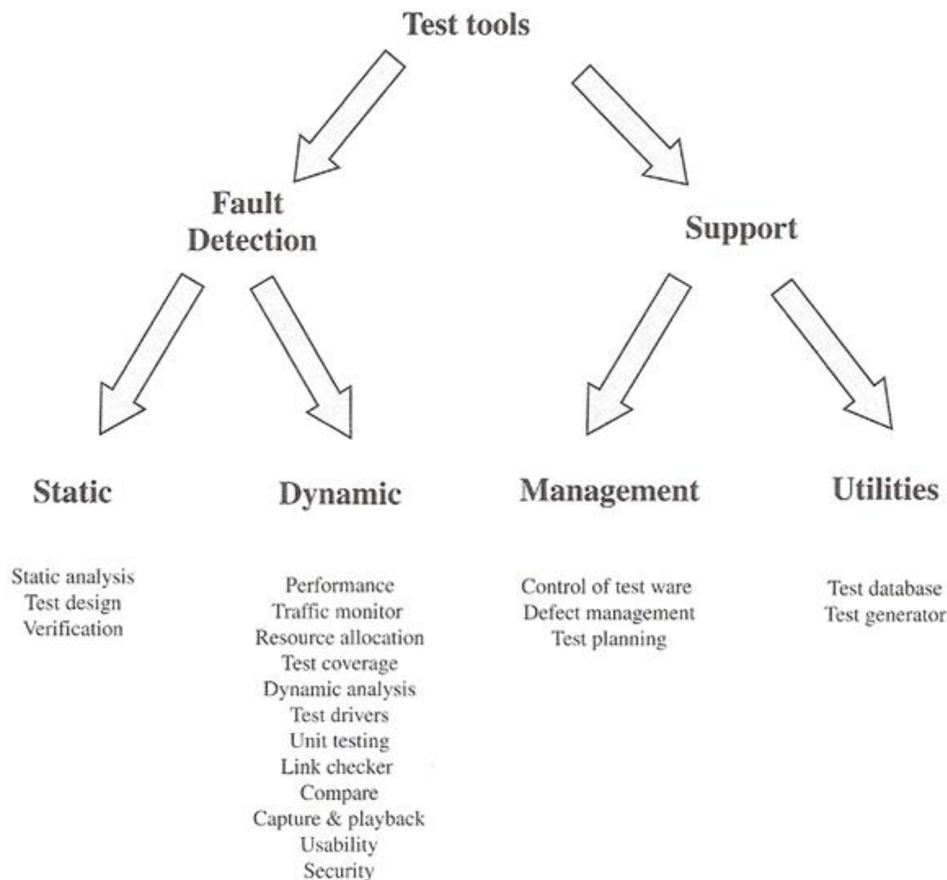

Abbildung 6: Überblick über die Arten von Test-Tools [Siteur05].

Tools für statische Analysen (Static analysis)

Wie schon im Kapitel 2.4.1. Statischer Test unter den werkzeugunterstützen Analysen erwähnt sind Analysatoren Programme, die den Sourcecode analysieren und Metriken und Analysen bezüglich der Daten- und Kontrollflüsse aufstellen. Diese Tools können Diagramme über die interne Struktur eines Programms wie z.B. einen Kontrollflussgraphen erstellen und automatisch Metriken berechnen und bereitstellen.

Test-Design-Tools (Test design)

Der Fokus dieses Test-Tools liegt auf dem Erstellen, Spezifizieren und Identifizieren von Testfällen. Dieses Tool kann aus Spezifikationen und logischen Testfällen konkrete Testfälle ableiten und sie in ein Format bzw. in ein Skript konvertieren, welches durch ein anderes Tool ausgeführt werden kann. Außerdem können mittels eines Test-Design-Tools Spezifikationen mit Testfällen sowie auch Fehler mit Testfällen verknüpft werden, wobei diese Fehler dann auch wieder auf die Spezifikation zurückgeführt werden können. Dieses Zurückführen wird unter Fachkreisen auch „Traceability” genannt [Siteur05].

Verifikationstools (Verification)

Dieser Typ von Tool überprüft entweder Quelltext oder Spezifikationen auf die Einhaltung von spezifischen Standards. Dazu zählen z.B. die Einhaltung von Codierstandards oder von so genannten „Style Guides“. Für Webtechnologien sind hierbei Tools zur Validierung von HTML-Dateien ein gutes Beispiel. Sehr bekannt in diesem Bereich ist der Validationsservice des World Wide Web Consortiums (W3C). Dieser ist über das Internet erreichbar und mit dessen Hilfe können Websites und andere Dateien validiert werden [W3CValidationService].

2.5.2. Tools für dynamische Tests

Unter Tools für dynamische Tests werden im Allgemeinen Werkzeuge verstanden, die der automatisierten Durchführung von Testfällen dienen. Sie entlasten den Tester und nehmen ihm lästige mechanische Arbeiten bzw. das Abarbeiten der Testfälle ab. Diese Tools versorgen das Testobjekt mit Eingabedaten und protokollieren und bereiten die Ausgabedaten auf [Spillner&Linz05]. Diese Eigenschaften sind äquivalent zu den PoO und PoC des Testrahmens aus Kapitel 2.4.2. Dynamischer Test.

Die Tools für dynamische Tests, wie sie auch in [Siteur05] zu finden sind, werden in den nächsten Abschnitten genauer erklärt.

Performanz-Tools (Performance)

Mit einem Performanz-Tool kann ein gleichzeitiger Zugriff auf das System von mehreren Usern simuliert werden. Anstatt dass mehrere Tester Testfälle gleichzeitig durchführen, führt das Tool selbst die Tests aus und protokolliert gleichzeitig auch das Verhalten des Systems auf die Last der User. Dies ist vor allem bei Webtechnologien eine sehr wichtige Testart, da über das Internet mehrere User gleichzeitig die Möglichkeit haben auf ein System zuzugreifen. Ein sehr bekannter Repräsentant dieser Art von Tools ist der LoadRunner von der Firma HP [HPLoadRunner].

Traffic Monitor

Unter einem Traffic Monitor versteht sich ein Tool welches die Zugriffe auf ein System wie z.B. auf einem Webserver protokolliert. Dadurch lässt sich die Effektivität des Systems und auch die Verwendungsweise beobachten und Änderungen können basierend auf diese Erkenntnisse durchgeführt werden.

Tools zur Ressourcenverwendung (Ressource allocation)

Diese Tools analysieren die Effizienz der Verwendung der Hardware durch die Software. Dabei liegt das Hauptaugenmerk auf Metriken wie z.B. die Prozessorlast, der verwendete Arbeitsspeicher und die Auslastung des Netzwerkes. Einige Performance-Tools wie z.B. der [HPLoadRunner] können dies zusätzlich zur eigentlichen Funktionalität.

Tools zur Testabdeckung (Test coverage)

Diese Tools, welche auf dem Whitebox-Verfahren aufbauen und den Quelltext als Referenz hernehmen, analysieren die durchgeführten Tests und vergleichen, wie viel vom Sourcecode durch die Tests abgedeckt wurde. Dies wird vor allem im Komponententest (Details siehe Kapitel 2.6. Testen im Softwarelebenszyklus) oft verwendet. Dabei ist es das Ziel, so viel Code wie möglich abzudecken, wobei dies nie zu hundert Prozent möglich sein wird. Ein Wert von 90% reicht oft schon aus, vor allem wenn Codeteile nur durch Exceptions ausgelöst werden können, welche nicht in einem normalen Testfall erreicht werden können [Siteur05].

Tools für dynamische Analysen (Dynamic analysis)

Diese Tools sind vergleichbar mit den Tools zur statischen Analyse des Codes, nur dass die Tools zur dynamischen Analyse das Laufzeitverhalten und den Code während der Ausführung beobachtet. Dabei können Laufzeitprobleme wie Ressourcenverwendung und Freigabe auf Codezeilen genau gefunden werden. Dies kann vor allem bei gescripteten Programmiersprachen wie JAVA oder .NET gut durchgeführt werden.

Testtreiber (Test drivers)

Wie im Kapitel 2.4.2. Dynamischer Test beim Testrahmen erwähnt ist ein Testtreiber ein Programmbaustein, welches das Testobjekt aufruft. Diese Aufgabe kann durch ein Tool übernommen und vereinfacht werden.

Tools für Komponententest (Unit testing)

Für den Komponententest gibt es vor allem für objektorientierte Programmiersprachen eigens Tools um die Objekte bzw. die Klassen auf deren richtigen Funktionalität zu testen. Unter Java-Entwicklern ist das Tool namens [JUnit] wohl der bekannteste Vertreter dieser Toolart.

Link checker

Diese Tools werden für Webapplikationen verwendet und überprüfen die Gültigkeit von Hyperlinks auf einer Website. Diese Tools sind nur für Webtechnologien interessant.

Tools zum Vergleichen (Compare)

Diese Art von Tools vergleicht den Output eines Testobjekts mit einem Erwartungswert. Dies entspricht dem „Point of Observation" im Testrahmen beim dynamischen Test (vergleiche Kapitel 2.4.2. Dynamischer Test). Dabei können nicht nur Output, sondern auch Dateien und Datenbankeinträge verglichen werden, welches wiederum komplexer vonstatten geht. Die meisten Capture and Playback-Tools unterstützen solche Vergleichsfunktionalitäten.

Capture and Playback-Tools

Diese Art von Tools sind die meist verbreiteten und sind für die Testautomatisierung wie z.B. beim Regressionstest wichtig. Es werden Testfälle von einem Tester ausgeführt, während dieses Tool die Aktionen aufzeichnet. Später können diese Tests immer wieder mittels eines Skripts mit dem Tool automatisiert ausgeführt werden. Dabei können Testfälle zu ganzen Szenarien zusammengelegt werden. Des Öfteren wird statt dem Aufzeichnen des Testfalls auch gleich ein Testskript vom Tester selbst programmiert.

Usability-Tools

Diese Tools beobachten und bewerten die Benutzerfreundlichkeit eines Systems. Dabei werden die Eingaben eines Users protokolliert sowie auch die Dauer, welche er oder sie benötigt um durch das System zu navigieren und ob es Stellen gibt, die den User irritieren.

Tools zum Prüfen der Sicherheit (Security)

Diese Tools überprüfen Attacken auf das System oder halten nach verdächtigen Daten Ausschau. All dies wird gemacht um ungewollte Daten und Transaktionen außerhalb zu halten.

2.5.3. Management-Tools

Management-Tools bieten Mechanismen zur bequemen Erfassung, Katalogisierung und Verwaltung von Testfällen an. Sie erlauben, den Status der Testfälle, also ob sie erfolgreich waren oder nicht, zu überwachen. Überdies unterstützen einige Testmanagement-Tools auch Aspekte des Projektmanagements wie Zeit- und Ressourcenplanung. Sie verhelfen dem Testmanager, die Tests zu planen und jederzeit den Überblick über hunderte oder tausende von Testfällen zu behalten. Auch anforderungsbasiertes Testen und das Verknüpfen von Requirements mit Testfällen wird durch fortgeschrittene Testmanagement-Tools unterstützt [Spillner&Linz05].

Zu diesen Test-Tools gehören die in den folgenden Abschnitten erläuterte Arten [Siteur05].

Tools zur Kontrolle der Testware (Control of testware)

Versionskontrolle von Testfällen, Testdaten und alles was mit Testen zu tun hat wird von diesem Tooltyp registriert und verwaltet. Die meisten Capture and Playback-Tools haben oft solch Funktionalitäten bereits inkludiert.

Fehlermanagement (Defect management)

Durch diese Tools können Fehler, welche durch Tester gefunden wurden, den Entwicklern vorgelegt werden sowie durch sie bearbeitet werden. Diese Art von Test-Tools kann auch als

Kommunikationsersatz zwischen Entwickler und Tester angesehen werden. Die gefundenen Fehler werden dem Tool bekannt gegeben und die Entwickler arbeiten diese ab.

Tools zur Testplanung (Test planning)

Die Planung von Testaktivitäten kann am besten mit einem reinen Projektplanungswerkzeug durchgeführt werden. Die Planung der Ausführung von Testfällen ist der Fokus von Tools zur Testplanung.

2.5.4. Utilities (zusätzliche Hilfsmittel)

Unter Utilities oder zusätzliche Hilfsmittel werden alle Arten von Tools bzw. Software verstanden, welche für den Testprozess hilfreich sein können aber im direkten Sinne nichts mit dem Softwaretest zu tun haben. Hierzu gehören, wie auch bei [Siteur05] beschrieben, Tools zur Manipulation von Daten in Datenbanken. Ein bekannter Vertreter dieser Gattung von Tools für Oracle Datenbanken ist [TOAD]. Mit diesem Tool ist es möglich auf eine Datenbank zuzugreifen und SQL-Statements abzusetzen, um die Datenbank zu manipulieren oder zu überprüfen. Dies ist vor allem beim Vor- und Nachbereiten von Tests wichtig.

Überdies gelten auch ein Textbearbeitungsprogramm wie Microsoft Word oder ein Tabellenkalkulationsprogramm wie Microsoft Excel als Hilfsmittel. Mit Word können Spezifikationen oder Reports geschrieben werden und mit Excel werden auch des Öfteren Testfälle verwaltet und Auswertungen erstellt.

2.6. Testen im Softwarelebenszyklus

Das Testen ist heutzutage kein zur Softwareentwicklung abgegrenzter Prozess mehr sondern gliedert sich in den einzelnen Phasen der Entwicklung ein. Dabei findet sowohl die Testplanung als auch die Ausführung der Tests zeitgleich mit der Planung und Durchführung der Entwicklungsarbeiten statt. Dabei werden für die einzelnen Phasen im Entwicklungsprozess verschiedene Methoden, Techniken und Tools des SWT verwendet und eingesetzt. Diese Unterschiede in den Phasen bezüglich der Verwendung der Testmethoden sowie auch von Tools im Softwarelebenszyklus werden in den folgenden Kapiteln erklärt. Davor folgt aber eine kurze Erläuterung des Softwarelebenszyklus' anhand des V-Modells von Boehm, in welcher die einzelnen Phasen aufgezeigt werden. Diesen Phasen werden später die Testmethoden und die Tools zugewiesen.

2.6.1. Der Softwarelebenszyklus: das V-Modell

Die Grundidee des V-Modells ist, dass Entwicklungsarbeiten und Testarbeiten zueinander korrespondierende, gleichberechtigte Tätigkeiten sind. Bildlich wird dies durch die zwei Äste eines „V" dargestellt. Der linke Ast beschreibt die Schritt für Schritt erstellte Spezifikation mit anschließender Programmierung der Software. Der rechte Ast steht für die Testarbeiten,

in deren Verlauf elementare Programmbausteine sukzessive zu größeren Teilsystemen zusammengesetzt werden und jeweils auf die richtige Funktion der korrespondierenden Spezifikation des linken Astes geprüft werden. Dabei können die Phasen in unterschiedlichen Versionen in Namen und Anzahl sowie von Firma zur Firma variieren [Spillner&Linz05].

Ein bildliches Beispiel für ein allgemeines V-Modell kann in Abbildung 7 gesehen werden.

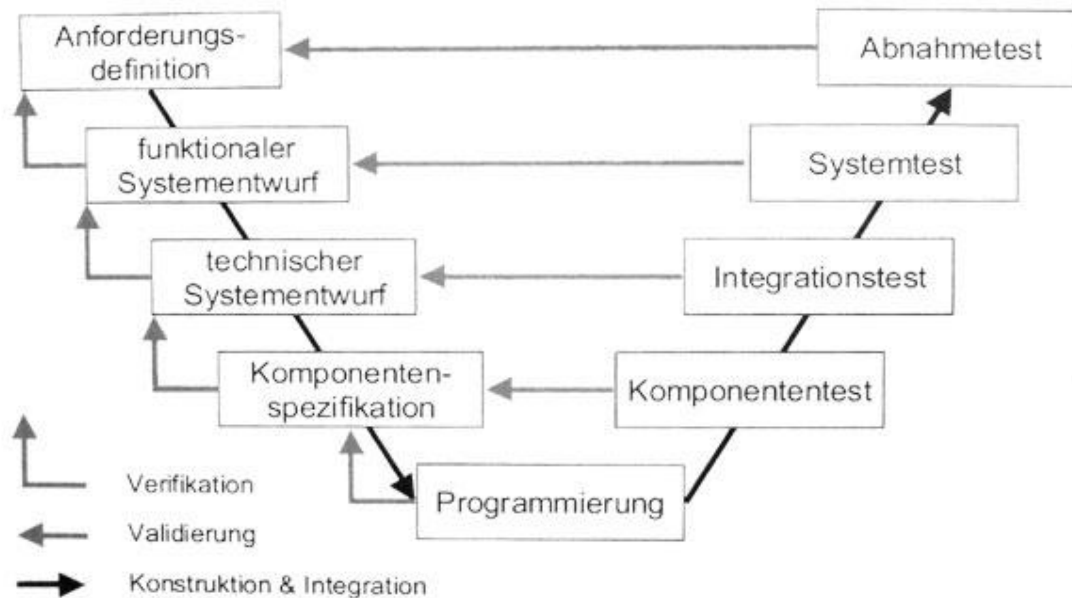

Abbildung 7: Der Softwarelebenszyklus des allgemeinen V-Modells [Spillner&Linz05].

Die Aktivitäten des linken Astes sind schon aus dem Wasserfallmodell bekannt und lassen sich wie folgt erklären [Spillner&Linz05]:

- **Anforderungsdefinition**
  Die Wünsche und Anforderungen des Kunden oder des späteren Anwenders werden gesammelt, spezifiziert und verabschiedet. Der Zweck und die gewünschten Leistungsmerkmale des zu erstellenden Systems liegen damit fest.

- **Funktionaler Systementwurf**
  Die Anforderungen werden auf Funktionen und Dialogabläufe des neuen Systems abgebildet.

- **Technischer Systementwurf**
  Die technische Realisierung des Systems wird entworfen. Hierzu gehören unter anderem die Definition der Schnittstellen zur Systemumwelt sowie die Zerlegung des Systems in überschaubare Teilsysteme (Systemarchitektur), die möglichst unabhängig voneinander entwickelt werden können.

- **Komponentenspezifikation**
  Für jedes Teilsystem werden die Aufgabe, das Verhalten sowie auch der innere Aufbau und Schnittstellen zu anderen Teilsystemen definiert.

- **Programmierung**
  Implementierung der Programmteile laut Spezifikation in einer Programmiersprache.

Der rechte Ast beschreibt zu dem oben angeführten Spezifikations- bzw. Konstruktionsschritt eine dazu korrespondierende Teststufe. Diese Teststufen lauten wie folgt [Spillner&Linz05]:

- **Komponententest**
  Überprüft, ob jeder Softwarebaustein bzw. jede Komponente der Spezifikation entspricht.

- **Integrationstest**
  Überprüft, ob die Komponenten wie im technischen Systementwurf vorgesehen über deren Schnittstellen zusammenarbeiten.

- **Systemtest**
  Überprüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt.

- **Abnahmetest**
  Überprüft, ob das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale aufweist.

In jeder Teststufe ist somit zu überprüfen, ob die Entwicklungsergebnisse diejenigen Anforderungen erfüllen, welche auf der jeweiligen Stufe relevant bzw. spezifiziert sind. Dieses Prüfen der Entwicklungsergebnisse gegen die ursprünglichen Anforderungen wird Validierung genannt. Die Verifikation hingegen ist auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu seiner Spezifikation nachweisen [Spillner&Linz05]. Dies kann auch in Abbildung 7 durch die Pfeile gekennzeichnet betrachtet werden.

Details zu den einzelnen Teststufen sowie zur Verwendung von Tools in den Stufen werden im nächsten Kapitel genauer erläutert.

2.6.2. Analyse der einzelnen Teststufen im V-Modell

Komponententest (Unit-Test)

Dies ist die erste Teststufe in der Softwareentwicklung und überprüft die Funktionalität und auch die Robustheit einzelner Softwarebausteine unabhängig bzw. isoliert voneinander. Diese Isolierung hat den Zweck, dass komponentenexterne Einflüsse beim Test ausgeschlossen werden können und auftretende Fehler nur innerhalb der Komponente zu finden sind. Die Komponente selbst kann je nach Programmiersprache anders genannt werden, wie z.B. Module, Units oder Klassen. Auch die Bezeichnung der Testart kann auf diese Art und Weise variieren, wobei der Ausdruck „Unit-Test“ bei objektorientierten Sprachen wie JAVA und .NET am gebräuchlichsten ist.

Wie auch bei [Siteur05] beschrieben werden in dieser Phase größtenteils Verfahren des Whitebox-Tests eingesetzt. Die Komponente wird auf Codeebene betrachtet und mit Hilfe von Tools für den Komponententest (Unit testing) getestet. Zusätzlich werden auch Tools zur Testabdeckung (Test coverage) eingesetzt um zu ermitteln, ob genügend Quellcode der Komponente auch durch die Tests abgedeckt wurde.

Es werden auch statische Analysen mit Quellcode-Analysatoren und Compilern sowie Reviews, meistens informelle Walkthroughs zwischen zwei Entwicklern, durchgeführt. In dieser Phase liegt das Testen noch in der Hand der Entwickler, da sie die innere Struktur der Komponente kennen und somit auch am besten Whitebox-Verfahren anwenden können.

Der Komponententest wird oft auch automatisiert, wozu wiederum Automatisierungstools bzw. Testskripte von Nöten sind. Überdies werden wie bei [Siteru05] erwähnt Testreiber benötigt, da die Komponente selbst nur ein Baustein ist, der meist übergeordnete Bausteine benötigt, die ihn aufrufen. Diese Treiber werden von den Entwicklern selbst programmiert und meistens sogar noch vor der eigentlichen Komponente selbst. Hierbei wird dann vom „Test first"-Ansatz oder von „Test-Driven Development" (kurz: TDD), auf Deutsch „Testgetriebene Entwicklung", gesprochen, welche auch bei [Spillner&Linz05] erwähnt werden.

Verifikationstools werden in dieser Phase auch verwendet um Codierstandards und „Style Guides" zu überprüfen. Dies ist vor allem wichtig um die Lesbarkeit und Wartbarkeit des Sourcecodes zu gewährleisten.

Integrationstest (Integration Test)

Die Teststufe nach dem Komponententest ist der Integrationstest, bei dem das Zusammenspiel zwischen den einzelnen Komponenten, also deren Schnittstellen, im Mittelpunkt steht. Dabei liegt das Hauptaugenmerk auf die innere Struktur des Testobjektes und es werden Whitebox-Verfahren angewendet. Auch Tools für den Komponententest können in dieser Phase Verwendung finden [Siteur05].

Der Integrationstest setzt voraus, dass die Komponenten fehlerfrei sind, damit Fehler bei der Integration auf die Schnittstellen zurückgeführt werden können. Es werden so lange die einzelnen Komponenten miteinander verknüpft und getestet, bis alle Komponenten wie es sein soll zusammenarbeiten und das System zum ersten Mal als Ganzes steht. Bis es aber soweit ist, muss der Integrationstest durchgeführt werden. Hierfür gibt es, wie auch bei [Spillner&Linz05] erwähnt, mehrere Ansätze, die ihre Vor- und Nachteile besitzen. Dabei handelt es sich um folgende:

- **Top-down-Integration**
  Der Test beginnt mit der obersten Komponente, also diejenige, die nicht von anderen Bausteinen aufgerufen wird außer vom Betriebssystem. Die untergeordneten Komponenten werden durch Platzhalter ersetzt. Später werden diese Platzhalter durch die richtigen untergeordneten Bausteine ersetzt, wobei die vorhin getestete Komponente als Testreiber fungiert. Für die neue zu testende Komponente werden wiederum Platzhalter benötigt, was mit der Zeit sehr aufwendig werden kann.

- **Bottom-up-Integration**
  Dieser Test beginnt mit der elementarsten Komponente des Systems, welche keine anderen Komponenten mehr aufruft außer Funktionen des Betriebssystems. Es werden Testreiber benötigt, welche die Tests für die jeweilige Komponente ausführen. Diese Treiber werden im Laufe der Entwicklung durch die richtigen aufrufenden Komponenten

ersetzt, für die wiederum Testtreiber benötigt werden. Auch dies kann mit der Zeit sehr aufwendig werden.

- **Ad-hoc-Integration**
  Dieser Test integriert die Komponenten in der Reihenfolge wie sie von der Entwicklung fertig gestellt werden. Dabei werden, wie der Name eben sagt, ad hoc die Testtreiber und Platzhalter programmiert und die Komponente getestet. Der Vorteil dabei ist im Gegensatz zu den oberen Methoden, dass ein Zeitgewinn erlangt werden kann weil die Komponenten so früh wie möglich integriert werden können. Der Nachteil ist dabei natürlich, dass im Gegensatz zu den obigen sowohl Testtreiber als auch Platzhalter benötigt werden.

- **Backbone-Integration**
  Bei diesem Test wird, bevor das eigentliche System programmiert wird, ein so genanntes „Backbone“ (auf Deutsch: Rückrad) erstellt, welches schon im Vorhinein alle Testtreiber und Platzhalter enthält und in dem die Komponenten in beliebiger Reihenfolge integriert werden können. Der Vorteil liegt wie bei der Ad-hoc-Integration im Zeitgewinn, da die Komponente gleich nach deren Fertigstellung getestet werden kann. Der Nachteil besteht natürlich in der Tatsache, dass dieses „Backbone“, welches sehr groß und komplex sein kann, erst programmiert werden muss und bei zu hoher Komplexität selbst auch wieder Fehler enthalten kann. Somit müsste auch das „Backbone“ selbst erst getestet werden, was vom Hundertsten ins Tausendste führen kann.

Zum Schluss kann noch, was aber vermieden werden sollte, die „Big Bang“-Integration erwähnt werden, bei der alle Bausteine nach ihrer Fertigstellung gleichzeitig integriert werden. Es werden keine Testreiber und Platzhalter benötigt sowie auch kein „Backbone“. Die Fehlerfindung kann ins Unmögliche führen, da die Lokalisierung eines Fehlers bei vielen Komponenten oft nicht möglich ist. Überdies ist die Wartezeit, bis alle Komponenten fertig gestellt worden sind, verlorene Testzeit und kann nicht mehr aufgeholt werden.

Systemtest

Nach dem abgeschlossenen Integrationstest folgt der Systemtest. Dieser betrachtet das System, welches in einer eigenen Testumgebung aufgesetzt ist aus möglichst allen später verwendeten Software- und Hardwareprodukten besteht und der Produktivumgebung des Kunden schon sehr nahe kommt. In diesem Test werden zum ersten Mal, wie auch bei [Siteur05] beschrieben, Blackbox-Verfahren angewendet. Die Spezifikation steht von nun an im Mittelpunkt des Interesses. Zudem sind ab dieser Stelle keine Programmierkenntnisse mehr vorausgesetzt und es werden auch keine Testtreiber oder Platzhalter mehr benötigt.

In dieser Testphase werden sowohl funktionale als auch nicht funktionale Anforderungen getestet. Die Sicht des Kunden bzw. des späteren Anwenders muss beim Testen eingenommen werden, was durch eigene Tester besser erreicht werden kann als durch Entwickler. Ab dieser Phase werden die meisten Testfälle erstellt, was den Gebrauch von einer Vielzahl von Tools begünstigt.

Wie bei [Siteur05] beschrieben werden beim funktionalen Systemtest Test-Design-Tools benötigt, damit die Tests aus der Spezifikation heraus identifiziert und geplant werden können. Außerdem werden Capture and Playback-Tools benötigt, um die Tests automatisiert

ablaufen und immer wieder kehrende Eingaben automatisch durchführen lassen zu können. Natürlich werden auch Tools zum Vergleich benötigt, damit die Testergebnisse mit der Spezifikation verglichen werden können, wobei die meisten Capture and Playback-Tools diese Funktionalität schon inbegriffen haben.

Beim nicht funktionalen Systemtest werden auch Capture and Playback-Tools benötigt, aber zusätzlich werden noch Tools für die spezifischen Arten von nicht funktionalen Tests benötigt (vergleiche Kapitel 2.3.2. Nicht funktionaler Test). Dazu gehören Tools zum Messen der Performanz des Systems sowie für das Testen der Sicherheit. Auch Tools zur Ressourcenverwendung werden verwendet, um die Effektivität des Systems zu ermitteln.

Systemintegrationstest (System integration test)

Der Systemintegrationstest ist vergleichbar mit dem oben genannten Integrationstest, nur werden nicht einzelne Komponenten zu einem Ganzen integriert sondern mehrere Systeme zu einem großen Gesamtsystem. Dabei kann weder von einem Blackbox-Test noch von einem Whitebox-Test gesprochen werden. Es beinhaltet, da es schon zu groß und komplex ist für einen Whitebox-Test, das Ziel aber ein technisches bzw. programmiertechnisches ist, sowohl Aspekte des einen als auch des anderen Testverfahrens. Der Tester braucht keine Kenntnisse über die Funktion der Systeme selbst zu haben, aber er muss die Schnittstellen und die Kommunikation zwischen den einzelnen Systemen verstehen. Hier kann von einem Greybox-Test gesprochen werden.

Die Tools, die in dieser Phase verwendet werden, sind wie beim Integrationstest Testtreiber und Platzhalter, die ein anderes System simulieren sowie auch Tools zum Vergleichen von Output wie Dateien oder Datenbankeinträgen.

Abnahme-, Akzeptanztest (Acceptance Test)

In dieser Testphase liegt zum ersten Mal nicht mehr die Verantwortung beim Hersteller der Software sondern beim Kunden. Das System wird lokal beim Kunden aufgesetzt und betriebsnah verwendet bzw. getestet. Hierbei stehen die Sicht und das Urteil des Kunden bzw. des späteren Anwenders im Vordergrund. Dieser Test ist womöglich der einzige Test, der beim Kunden selbst durchgeführt werden muss und den der Kunde auch nachvollziehen kann.

In der Regel gibt es mehrere Formen eines solchen Abnahmetests. Laut [Spillner&Linz05] können folgende unterschieden werden:

- **Test auf vertragliche Akzeptanz**
  Bei diesem Abnahmetest gelten vertraglich geregelte Testkriterien als Richtwert für die Erfüllung von Anforderungen. Auch die Erfüllung eventuell relevanter gesetzlicher Vorschriften, Normen oder Sicherheitsbestimmungen gehören hier dazu. Für den Abnahmetest reicht es aus, die gemäß Vertrag abnahmerelevanten Testfälle durchzuführen und dem Kunden zu demonstrieren, dass die Akzeptanzkriterien des Vertrags erfüllt sind. Dabei ist es wichtig, dass der Kunde selbst die Testfälle entwirft oder sie einem sorgfältigen Review unterzieht, denn der Softwarehersteller könnte die vertraglich vereinbarten Akzeptanzkriterien missverstanden haben [Spillner&Linz05].

- **Test auf Benutzerakzeptanz**
  Ein weiterer wichtiger Aspekt der Akzeptanz ist die Benutzerakzeptanz, auch „Usability" genannt. Solch ein Test ist immer dann zu empfehlen, wenn der Kunde und der Anwender des Systems verschiedene Personengruppen angehören. Wie bei [Spillner&Linz05] erwähnt haben die unterschiedlichen Anwendergruppen in der Regel ganz verschiedene Erwartungen an ein neues System. Wenn eine Gruppe das System ablehnt, kann die Abnahme nicht erfolgen, obwohl das System funktional vollkommen in Ordnung ist. In diesem Fall gibt es kein Problem mit der Funktionalität, sondern mit den nicht funktionalen Anforderungen wie sie in Kapitel 2.3.2. Nicht funktionaler Test erläutert werden.

- **Akzeptanz durch Systembetreiber**
  Ein Test auf Akzeptanz durch die Systembetreiber soll sicherstellen, dass sich das neue System aus Sicht der Systemadministratoren in die vorhandene IT-Landschaft einfügt. Untersucht werden z.B. die Backup-Routinen, der Wiederanlauf nach Systemabschaltung, die Benutzerverwaltung oder Aspekte der Datensicherheit.

- **Feldtest (Alpha- und Beta-Tests)**
  Dies ist eine Sonderform des Akzeptanztests und wird vor allem dann durchgeführt wenn die Software in sehr vielen verschiedenen Produktivumgebungen betrieben werden soll und es unmöglich ist, alle Möglichkeiten und Kombinationen in einer Testumgebung nachzubilden. Hierzu liefert der Hersteller stabile Vorabversionen an einen ausgewählten Kundenkreis, der den Markt für die Software gut repräsentiert. Solche Feldtests werden sehr oft in der Computerspielindustrie eingesetzt, bei der jeder Endanwender mit Sicherheit eine andere Hardware und Software in Verwendung hat. Dabei ist der Alpha-Test ein Feldtest in der Testumgebung des Herstellers und ein Beta-Test ein Feldtest in der Produktivumgebung des Kunden.

Im Bezug auf Tools kann beim Abnahmetest nicht mehr viel verwendet werden. Wenn solche Abnahmentests in der Testumgebung des Herstellers „geprobt" werden, dann können natürlich Capture and Playback-Tools verwendet werden um die Tests zu automatisieren. Wenn möglich kann dies beim Kunden genauso automatisiert wiederholt werden. Test-Design-Tools können auch zum Finden von Testfällen für den Abnahmetest herangezogen werden. Für den Test der Benutzerakzeptanz können natürlich auch, wie bei [Siteur05] erläutert, Usability-Tools von Nutzen sein.

Test nach Änderung (Wartung, Weiterentwicklung, etc.)

Bei [Spillner&Linz05] wird nach dem Akzeptanztest noch die Möglichkeit einer Weiterentwicklung oder einer Änderung des Systems im Nachhinein in Betracht gezogen. Software ist Jahre oder oft auch Jahrzehntelang im Einsatz und wird des Öfteren korrigiert, geändert und erweitert. Es entsteht jedes Mal eine neue Version des Produktes, welche wieder getestet werden muss. Dabei sind Regressionstest, Systemtests und auch Abnahmetests durchzuführen, um die richtige Funktionalität zu beweisen. Dabei sind Tests nach Wartung der Software, also Änderungen aufgrund neuer Updates des Betriebssystems oder Patches von verwendeten Fremdsystemen, oder nach einer Weiterentwicklung durchzuführen.

2.7. Fazit zum Softwaretest

Zum Abschluss der Theorie werden noch die sieben Grundprinzipien des Softwaretests beschrieben, welche sich in den letzen 40 Jahren im Softwaretest herauskristallisiert haben und bei [Spillner&Linz05] aufgezählt werden. Diese Prinzipien haben bis zum heutigen Tag Gültigkeit und können als genereller Leitfaden für das Testen angesehen werden.

1. Grundsatz: Testen zeigt die Anwesenheit von Fehlern
Wie schon in dieser Arbeit erwähnt hat Testen als Motivation das Finden von Fehlerwirkungen in einem Programm oder Codestück. Unter Testen darf nicht eine Tätigkeit verstanden werden, welche nur die richtige Funktion einer Software überprüft. Laut der Definition bei [Kaner99] ist ein Test der keine Fehler findet verschwendete Zeit.

2. Grundsatz: Vollständiges Testen ist nicht möglich
Wie schon oben erwähnt ist das vollständige Testen eines Testobjektes nicht möglich. Es gibt viel zu viele Möglichkeiten, Randbedingungen und Situationen in denen ein System geraten kann und diese kann ein Tester nicht alle beachten. Außerdem werden auch oft Möglichkeiten für Szenarien einfach nicht bedacht.

3. Grundsatz: Mit dem Testen frühzeitig beginnen
Testaktivitäten sollen im Softwarelebenszyklus so früh wie möglich begonnen werden. Denn wie schon erwähnt wurde, sind die Kosten für das Finden und Beheben von Fehlern am Anfang geringer und steigen mit dem Fortschritt des Projektes immer weiter an (Stichwort: Boehm'sche Kurve, Abbildung 1).

4. Grundsatz: Häufung von Fehlern
Dieser Grundsatz beschreibt die Tatsache, dass an einer Stelle eines Testobjektes, wo ein Fehler entdeckt wurde, sich mit hoher Wahrscheinlichkeit noch mehr befinden und sich an einer Stelle „häufen“ (Stichwort: Fehlermaskierung).

5. Grundsatz: Zunehmende Testresistenz (Pesticide paradox)
Dies beschreibt die Tatsache, dass Tests, die immer wieder wiederholt werden, mit niedriger Wahrscheinlichkeit neue Fehler aufdecken. Genauso wie Schädlinge gegen Pflanzenschutzmitteln resistent werden, werden auch Fehler resistent gegenüber Tests. Deswegen sollten Testfälle regelmäßig auf deren Aktualität und Effektivität geprüft werden und gegebenenfalls durch neue oder modifizierte Testfälle ersetzt werden.

6. Grundsatz: Testen ist abhängig vom Umfeld
Dieser Grundsatz beschreibt die Tatsache, dass keine zwei zu prüfenden Systeme gleich sind und das Testen immer an den gegebenen Voraussetzungen anzupassen ist. Wie in dieser Arbeit schon einmal erwähnt wurde müssen z.B. die Tests bei einem sicherheitsrelevanten System anders aussehen als bei einem anderen System ohne sicherheitsrelevante Aspekte.

7. Grundsatz: Trugschluss: Keine Fehler bedeutet ein brauchbares System
Dieser Grundsatz beschreibt den Trugschluss, dass bei einem System, bei dem keine Fehlerwirkungen auftreten, ein gutes ist. Wenn die Vorstellungen und Anforderungen nicht den Kundenwünschen entsprechen ist dies mindestens genauso schlimm wie ein völlig fehlerbehaftetes System, welche die spezifizierten Anforderungen erfüllen würde.

Im nächsten Kapitel folgt nun der Vergleich der beiden Open-Source Webtest-Tools „Selenium“ und „Canoo WebTest“, welche über das Internet frei erhältlich sind.

3. Vergleich zweier Open-Source Webtesting-Tools

In den nächsten Kapiteln werden zwei Open-Source Tools für das Testen von Webapplikationen namens „Selenium“ ([Selenium]) und „Canoo WebTest“ ([Canoo WebTest]) verglichen. Diese beiden Tools sind über das Internet erhältlich und auch dokumentiert. Es werden allgemeine Informationen über die Tools bereitgestellt sowie kurz die Funktionsweise erläutert. Es wird analysiert, welche Aufgabenbereiche die Tools abdecken und wo die Stärken bzw. die Schwächen der Tools liegen. Zum Schluss wird noch ein Resümee zum Vergleich bereitgestellt, welches die Vor- und Nachteile von „Selenium“ und „Canoo WebTest“ abwägt.

Aber bevor der Vergleich durchgeführt wird, wird im nächsten Kapitel kurz erläutert, was unter den Begriffen Webtest und Webapplikation verstanden wird. Überdies wird analysiert, inwiefern sich der Test in diesem Bereich mit den allgemeinen Regeln und Grundlagen des Softwaretests, welche im Grundlagenteil dieser Arbeit besprochen wurden, unterscheiden und was die Besonderheiten sind.

3.1. Was ist Webtest?

Unter dem Begriff Webtest wird das Testen von Webapplikationen verstanden, wobei es sich hierbei um ein Client-Server-System handelt, welches serverseitig mit einem Webserver und clientseitig mit einem Webbrowser wie dem Microsoft Internet Explorer oder dem Mozilla Firefox arbeitet. Die Kommunikation zwischen dem Server und dem Client findet über ein Netzwerk wie z.B. dem Internet statt. Als Protokoll für die Kommunikation dient das „Hypertext Transfer Protocol“ (kurz: HTTP), welches Webseiten von Server zum Client überträgt, welche der Browser dann rendert bzw. darstellt. Diese Webseiten werden mit Hilfe der „Hypertext Markup Language“ (kurz: HTML) beschrieben, welche eine textbasierte Auszeichnungssprache ist und von XML („Extensible Markup Language“) abstammt.

Wie bei [Nguyen03] beschrieben, ist ein Client im Allgemeinen in der Lage durch ein Userinterface Daten an einen Server zu schicken und zu empfangen. Im Unterschied zu einem traditionellen Client, der plattformspezifisch ist und extra programmiert wurde, ist ein Webbasierter Client plattformunabhängig, solange ein Webbrowser auf dem Client vorhanden, aktuell und mit den gängigen Standards, wie die der W3C, kompatibel ist. Dabei liegt der Vorteil auf der Hand, da der Client nicht mehr programmiert werden muss sondern durch einen Webbrowser ersetzt wird. Die Unterschiede der einzelnen Webbrowser müssen hierbei beachtet werden, wobei diese sich soweit wie möglich an die Standards halten, wobei Fehler und Abweichungen nie ausgeschlossen werden können und auch des Öfteren auftreten.

Für das Testen solcher Applikationen ist das schon im Kapitel 2.4.2. Dynamischer Test vorgestellte Greybox-Verfahren gut geeignet. Wie bei [Nguyen03] beschrieben besitzt dieses Verfahren Methoden und Werkzeuge, welche sich vom Wissen über den internen Aufbau der Applikation sowie der Umgebung, mit dem das System interagiert, beschäftigen. Deswegen ist das Greybox-Verfahren für den Webtest so gut geeignet, da es sich im hohen Maße mit Systemdesign, Systemumgebungen und Konditionen der Interoperabilität, welche beim Webtest beachtet werden müssen, beschäftigt.

3.2. Analyse von „Selenium“

Unter „Selenium“ versteht sich eine Gruppe von Test-Tools für den automatisierten Test mit jeweils unterschiedlichen Funktionalitäten. Dabei können folgende Teile unterschieden werden [Selenium]:

- **Selenium IDE**
  Darum handelt es sich um ein Add-On für den Mozilla Firefox, welches in der Lage ist Mausklicke, Tastenanschläge und andere Aktionen aufzunehmen. Dieses aufgenommene Skript dient dann als Testfall, welches dann wieder im Browser ausgeführt werden kann. Dieses Tool entspricht einem Capture and Playback-Tool.

- **Selenium Remote Control (RC)**
  Dieses Tool lässt die durch den Selenium IDE aufgenommenen Testfälle in verschiedene Browser und auf verschiedene Plattformen ablaufen.

- **Selenium Grid**
  Dieses Tool erweitert den Selenium RC und ist in der Lage, die Testfälle auf mehrere Rechner zu verteilen. So kann Zeit gespart werden, da die Tests parallel durchgeführt werden können.

Für den Vergleich wird nur das Capture and Playback-Tool und Firefox Add-On Selenium IDE herangezogen und analysiert.

3.2.1. Verwendungsweise

Auf die Installation des Firefox Add-Ons wird hier nicht näher eingegangen und funktioniert problemlos und einfach. Nach der Installation kann Selenium IDE unter „Extras“ und „Selenium IDE“ gestartet werden. Es öffnet sich ein Fenster wie es in Abbildung 8 zu erkennen ist. Ab diesem Zeitpunkt werden alle Aktionen des Users aufgezeichnet und die Tests können durchgeführt werden.

Ein Beispielszenario:

1. Öffne die Webseite „www.google.at“
2. Schreibe „selenium“ ins Suchfeld
3. Klicke auf „Goolge-Suche“
4. Klicke auf den ersten Suchtreffer (Selenium web application testing system)
5. Wähle den Text „Get started with Selenium!“ aus, mach einen Rechtsklick und wähle die Option „verifyTextPresent Get started with Selenium!“ (Dies überprüft, ob der Text auf der Seite vorhanden ist. Es gibt noch mehrere Funktionen zur Verifikation.)
6. Klicke bei der Selenium IDE auf den roten Knopf zum Beenden der Aufnahme

Danach sind die einzelnen Schritte, so wie sie auch im Tab „Table“ in der Selnenium IDE in Abbildung 8 zu sehen sind, aufgezeichnet. Es können die einzelnen Schritte noch modifiziert werden. Dabei ist es im Tab „Source“ auch möglich den Sourcecode, den die IDE automatisch erstellt, direkt zu modifizieren, wobei hierfür Programmierkenntnisse erforderlich sind. Es kann auch die Programmiersprache bzw. das Format, in denen die Testfälle gespeichert werden sollen, ausgewählt werden.

Folgende Auswahlmöglichkeiten für das Format bzw. die Programmiersprache gibt es:

- **HTML**
- **JAVA – Selenium RC**
- **C# - Selenium RC**
- **Perl – Selenium RC**
- **PHP – Selenium RC**
- **Python – Selenium RC**
- **Ruby – Selenium RC**

Wie erkennbar ist steht bei den Optionen auch gleich dabei, ob sie vom Selenium RC unterstützt werden. Dies ist für den späteren Prozess, wenn die Remote Control oder Selenium Grid verwendet werden sollen, sehr nützlich zu wissen.

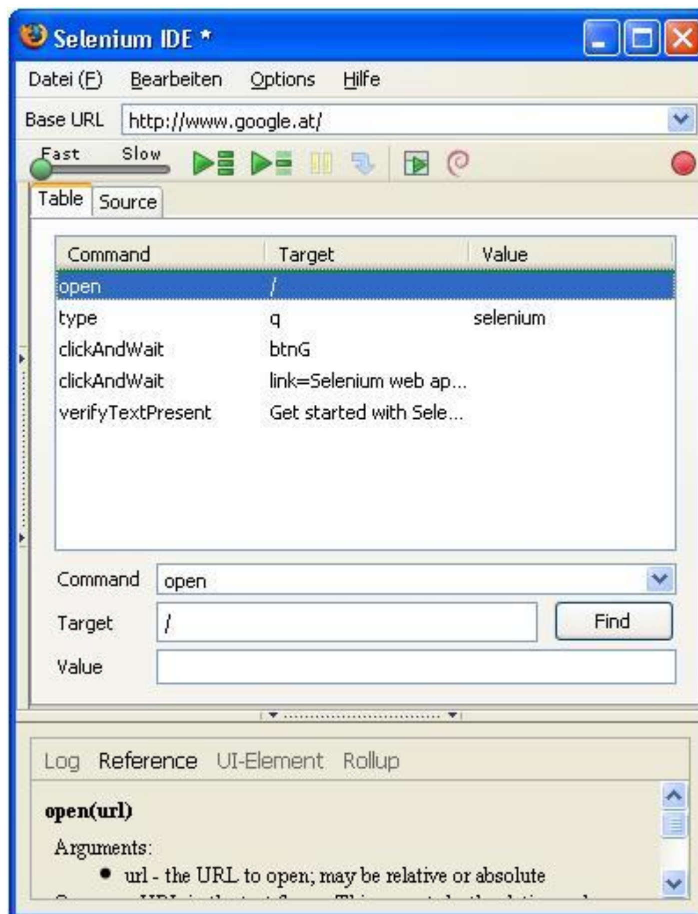

Abbildung 8: Ansicht des Firefox Add-Ons „Selenium IDE“.

Das Abspielen von Testcases bzw. Szenarios ist insofern einfach, da nur der grüne Button „Play entire test suite“ oder „Play current test case“ gedrückt werden muss. Dadurch werden die Kommandos abgearbeitet und der Mozilla Firefox öffnet die Seiten und drückt die Links automatisch bis zum Ende des Testfalls bzw. des Szenarios. Das Speichern bzw. Exportieren der Tests ist auch möglich, wobei das Format bzw. die Programmiersprache, so wie oben schon erklärt, ausgewählt werden kann.

3.2.2. Merkmale des Tools

Ein Vorteil der Selenium IDE ist die einfache Bedienung. Alles erklärt sich von selbst und die Basics sind auch auf der Webseite von „Selenium“ ([Selenium]) zu finden. Die Bedienung ist sehr übersichtlich und einfach. Gerade aufgenommene Tests können sofort noch einmal durchlaufen werden sowie auch nach Modifikationen. Dies vereinfacht die Überprüfbarkeit der Testfälle auf Fehler und die Sicherstellung der Richtigkeit. Überdies werden bei den Eingabefeldern im Tab „Table“ bei Eingabe die möglichen Kommandos vorgeschlagen, welche sich vom Namen her selbst erklären. Für den Fall, dass die Funktionsweise einer Funktion einmal nicht klar sein sollte, dann kann diese problemlos ausprobiert werden.

3.3. Analyse von „Canoo WebTest“

Unter „Canoo WebTest“ versteht sich ein Framework zum Testen von Webapplikationen mit einer sehr guten Reporting-Funktion. Testfälle und Szenarien für dieses Tool werden im Normalfall programmiert, aber es existiert auch ein Firefox Add-On zum Aufzeichnen von Tests vergleichbar mit der Selenium IDE namens „WebTestRecorder“. Dieses Tool ist sehr banal gehalten und ist im Grunde ein Capture and Playback-Tool, wobei Tests nur aufgenommen aber nicht mehr abgespielt werden können.

3.3.1. Verwendungsweise

Auf die Installation des Firefox Add-Ons wird hier nicht näher eingegangen und funktioniert problemlos und einfach. Nach der Installation kann der WebTestRecorder unter „Extras“ und „Webtest Recorder Sidebar“ gestartet werden. Es erscheint eine Sidebar wie sie in der Abbildung 9 zu erkennen ist, die die Aktionen des Users mitschreibt und in Form von XML, Groovy-Code (eine JAVA-Erweiterung) und WebDriver (Details hierfür siehe [WebDriver]) anzeigt.

Ein Beispielszenario:

1. Öffne die Webseite „www.google.at“
2. Schreibe „canoo webtest“ ins Suchfeld
3. Klicke auf „Goolge-Suche“
4. Klicke auf den ersten Suchtreffer (Canoo WebTest)
5. Wähle den Text „Canoo WebTest“ aus, mach einen Rechtsklick und wähle die Option „Add verifyText for ‚Canoo WebTest'“ (Dies überprüft, ob der Text auf der Seite vorhanden ist. Es gibt noch weitere Funktionen zur Verifikation.)

Obwohl es die Möglichkeit gibt, wie auch in Abbildung 9 ersichtlich, ein Häkchen bei „Enable recording“ zu setzen bzw. zu deaktivieren, um die Aufnahme zu starten bzw. zu beenden, scheint das Tool trotzdem immer weiter aufzuzeichnen. Die Handhabung lässt zu wünschen übrig und ist sehr minimalistisch gehalten, was aber auch wieder von Vorteil sein kann. Der Code, den der WebTestRecorder erstellt, kann beliebig modifiziert werden. Sogar während der Aufnahme ist das Editieren möglich. Dieses Tool ist sehr gut auf Spezialisten abgeschnitten denen die minimalistische Funktionsweise des Rekorders ansprechen.

Das Speichern von den aufgenommenen Tests ist nicht möglich, da aber der erstellte Sourcecode, wie in der Abbildung 9 in XML ersichtlich, in der Sidebar angezeigt wird, kann dieser einfach kopiert und mit Hilfe eines Texteditors gespeichert werden.

Abbildung 9: Ansicht der Sidebar des Firefox Add-Ons „Webtest Recorder Sidebar“.

3.3.2. Merkmale des Tools

Wie schon oben erwähnt ist der WebTestRecorder von „Canoo WebTest“ sehr minimalistisch gehalten. Er ist ein reines Tool zum Aufnehmen von Testfällen für den „Canoo WebTest“, wobei diese nicht mehr im WebTestRecorder selbst abgespielt werden können. Somit sind die Tests nicht mehr ad hoc auf deren richtigen Funktionsweise prüfbar, was zu Fehlern führen kann. Überdies ist dieses Tool nur für Spezialisten geeignet, welche die Kommandos des Canoo WebTest-Tools kennen und Fehler allein durch das Lesen des Quellcodes im Recorder erkennen und modifizieren können.

3.4. Fazit des Vergleichs

Beide Tools sind zum Aufzeichnen von automatisierten Testfällen konzipiert und verfolgen dasselbe Ziel. Dieses versuchen die beiden Tools aber auf verschiedene Wege zu erreichen.

Die Selenium IDE ist sehr benutzerfreundlich und gut bedienbar. Das Userinterface ist übersichtlich und erklärt sich von selbst. Somit ist der Einstieg für Anfänger sowie für technisch eher unversierte Tester einfach, da im Grunde keine Programmierkenntnisse vorausgesetzt werden. Falls Kommandos per Hand in den Tab „Table“ eingefügt werden müssen, dann werden die möglichen Eingaben mittels Popup vorgeschlagen. Zudem sind die möglichen Eingaben auf der Webseite von „Selenium“ ([Selenium]) gut dokumentiert. Auch der Quellcode kann im Nachhinein modifiziert werden, was das Tool auch für Spezialisten interessant macht. Außerdem können Tests gleich nach deren Aufnahme noch mal durchgeführt werden, was die Überprüfung der richtigen Funktion der Tests auch für technisch unversierte Benutzer einfach und nachvollziehbar macht.

Der WebTestRecorder von „Canoo WebTest“ ist hingegen ein sehr minimalistisch gehaltenes Tool für Spezialisten. Die Bedienung lässt nicht mehr zu als dass die Aktionen des Users nach dem Öffnen der „Webtest Recorder Sidebar“ aufgezeichnet werden. Ein Vorteil zur Selenium IDE ist aber die Tatsache, dass die Sidebar des WebTestRecorders im Gegensatz zum extra Fenster der IDE immer sichtbar ist und die Aktionen gleich nach deren Durchführung in der Sidebar sichtbar werden. Somit können gleich Zeile für Zeile Verifikationen über die Richtigkeit der Kommandos durchgeführt werden, vorausgesetzt der User versteht den Quelltext, den der Recorder anzeigt. Die möglichen Kommandos bzw. die Syntax ist nur schlecht dokumentiert. Nur ein erfahrener Benutzer dieses Tools mit genügend technischem Knowhow kann damit etwas anfangen.

Zum Schluss müssen noch die unterschiedlichen Funktionsweisen der Testframeworks erwähnt werden. „Selenium“ arbeitet immer mit einem Webbrowser, um die Tests durchzuführen. Das Framework öffnet selbstständig Browserfenster, führt die Tests durch und schließt das Fenster wieder. „Canoo WebTest“ hingegen führt die Tests im Hintergrund im Tool selbst aus und verwendet keinen Browser als Testtreiber, was es schneller als „Selenium“ macht. Somit ist „Canoo WebTest“ während der Entwicklung für schnelle Tests und gute Reports sehr zu empfehlen. Für einen endgültigen Test ist hingegen „Selenium“ zu empfehlen, weil nur mit diesem Tool auch browserspezifische Unterschiede völlig automatisiert getestet werden können.

4. Conclusion

Der Softwaretest ist in den letzten Jahren eine eigene, ernstzunehmende Disziplin in der Branche der Softwareentwicklung geworden. Keine professionelle Entwicklung kann mehr ohne ein dediziertes Team von Softwaretestern funktionieren. Der Test wird immer mehr in den Mittelpunkt gestellt, was auch beim „Test-Driven Development“ (TDD) bzw. beim „Test-first“-Ansatz gesehen werden kann, wo die Planung und Erstellung der Tests noch vor der eigentlichen Entwicklung begonnen wird und in dem auch Entwickler involviert sind welche Testtreiber, Platzhalter oder ein „Backbone“ für den Integrationstest programmieren.

Überdies gibt es Standards wie die Normen der DIN und der ISO, sowie auch Zertifikate wie z.B. die der ISTQB, welche im Bereich Softwaretest messbare Richtlinien und Normen beschreiben. Diese Normen und Richtlinien dienen mittlerweile schon International als Standard und machen es möglich, dass etwas so subjektives wie Qualität von Software vergleichbar gemacht werden kann.

Über die Grundlagen des Softwaretests und deren Verständnis steht aber die unabstreitbare Tatsache, dass ein Tester erst durch Erfahrung „gut“ wird. Durch die Theorie kann vielleicht ein Grundverständnis über die Methoden erlangt werden, aber erst durch viel Erfahrung und Praxis kann sich ein Tester ein gutes Gefühl für das Finden von Fehlern in einer Software aneignen. Eine weitere Eigenschaft, die ein guter Tester mit sich bringen muss, sind viele „Soft Skills“. Als Überbringer von „schlechten Nachrichten“ bzw. von Fehlern, welche ein anderer verursacht hat, benötigt ein Tester ein gewisses Maß an Feingefühl, um die Entwickler überzeugen zu können, dass sie den gefundenen Fehler korrigieren sollen. Somit sind Überredenskünste sowie auch eine gewisse Dominanz und Überzeugungskraft notwendig und gefragt für einen Softwaretester.

Außerdem ist es für einen Tester unabdingbar ein paar Begriffe, welche für den Softwaretest wichtig sind, und deren Definition zu kennen. Hierzu zählt auch der Begriff des Fehlers, welcher unterschiedliche Definitionen und Namen haben kann, darauf ankommend, auf was sich der Fehler bezieht und wie sich dieser auswirkt. Auch die Definition des Testens muss sich ein Tester immer gewahr sein, um seine Arbeit mit bestem Wissen und Gewissen ausführen zu können. Da Qualität und deren Sicherung auch zu den Aufgabenbereichen des Softwaretesters gehören, muss auch dieser Begriff der Softwarequalität und deren Definition zum Wissen eines Testers zählen, wobei diese durch die ISO-Norm 9126 definiert sind. Zudem muss ein Tester, da er in einem Projekt nur eine gewisse Zeitspanne zum Testen besitzt, auch den angebrachten Aufwand für seine Arbeit schätzen können und wissen, wie dieser ermittelt werden kann.

Die grundlegenden Testarten, die für die Durchführung geeignet sind und sich im Ziel und in der Motivation unterscheiden, muss sich ein Tester genau einprägen, damit seine Arbeit so zielgerichtet und effizient wie möglich sein kann. Dazu zählen die Ziele und Methoden des funktionalen Tests, welcher die reine Funktionalität des Systems, wie sie in einer Spezifikation beschrieben stehen, betrachtet. Im Gegensatz dazu existiert auch der nicht funktionale Test, der sich mit Aspekten wie der Benutzerfreundlichkeit und der Akzeptanz des späteren Anwenders beschäftigt. Dabei steht nicht die Funktion des Systems im Mittelpunkt, sondern das „Wie“ bzw. die Tatsache, mit welcher Qualität das System seine Aufgaben verrichtet. Im Gegensatz dazu kann noch der strukturbezogene Test unterschieden

werden, welcher sich nicht an Spezifikationen hält, sondern den Quellcode der Software betrachtet und diesen Tests unterzieht. Hierbei ist es das Ziel, dass soviel Quellcode mit Tests abgedeckt wird, sodass fehlerhafte Programmteile zu einem späteren Zeitpunkt ausgeschlossen werden können. Zudem kann der so genannte änderungsbezogene Test, auch Regressionstest genannt, unterschieden werden. Bei dieser Testart werden Programmteile, die schon einmal erfolgreich getestet wurden, nach einer Änderung des Systems einer weiteren Überprüfung unterzogen. Dies stellt bei mehreren Wiederholungen sicher, dass Änderungen keine ungewollten Nebeneffekte nach sich ziehen. Auch bei dieser Testart ist es von hoher Wichtigkeit, dass der Aufwand der Testarbeiten nicht zu groß wird und wie dieser Aufwand ermittelt werden kann.

Um ein systematisches Testen erst anwendbar zu machen, benötigt es Methoden, um Testfälle erstellen bzw. eine Sicht auf das Testobjekt bekommen zu können. Hierbei sind Testmethoden wie das Blackbox-Verfahren, welches das Testen aufgrund von Spezifikationen durchführt, und das Whitebox-Verfahren, welches den Programmtext als Referenz für die Tests hernimmt, zu betrachten. Dabei können bei beiden Methoden verschiedene Ansichten auf das Testobjekt eingenommen werden, welche sich sehr unterscheiden und einen guten Gesamtüberblick über das System darbieten. Durch die großen Unterschiede zwischen den beiden Testarten ist es somit von Vorteil eine Mischform von beiden Methoden anzuwenden. Diese hat einen eigenen Namen und wird Greybox-Verfahren genannt.

Da in der heutigen Zeit Softwareprojekte immer größere Ausmaße annehmen, werden Hilfswerkzeuge für Tester immer wichtiger und zum Teil auch unabdingbar. Ein Tester sollte in der Lage sein, die einzelnen Arten von Tools sowie deren Funktionsweisen und Anwendungsgebiete zu unterscheiden. Auch der Verwendung dieser Tools während der einzelnen Schritte im Entwicklungsprozess sollte sich ein Tester immer gewahr sein. Genauso sollten auch die Teststufen, welche den Entwicklungsstufen gegenüberstehen und im V-Modell von Boehm gesehen werden können (siehe Abbildung 7), von einem Tester beachtet werden und sich deren Ziele und Methoden gewahr sein.

Über den ganzen Methoden, Testarten und Begriffe des SWT, welche auch von der ISTQB im [ISTQB-Syllabus] so gelehrt werden, steht die Tatsache, dass Softwaretest heutzutage nicht mehr in der Softwareentwicklung weg zu denken ist. Da ein Softwaretester für die Sicherung der Qualität des Produktes sowie der Dokumentation und der übrigen Projektdokumente verantwortlich ist, können die Entwickler sich auf das Programmieren, Designen und Beheben von Fehlzuständen im Programm selbst konzentrieren. Sie müssen die Fehler nicht mehr selbst suchen oder nachweisen, sondern sie bekommen sie durch die Tester vorgelegt. Diese Arbeitsweise hat sich als sehr effektiv und effizient herausgestellt, vorausgesetzt die Zusammenarbeit sowie das gegenseitige Verständnis zwischen den Testern und den Entwicklern ist gegeben und funktioniert.

Somit kann die Schlussfolgerung gezogen werden, dass die Softwaretester, auch wenn sie nicht im Mittelpunkt der Entwicklung stehen und eher im Hintergrund arbeiten, eine wichtige Rolle in der Softwareentwicklung der heutigen Zeit spielen. Die Priorität des SWT wird sogar noch um einiges steigen, da Systeme immer größer und komplexer werden und das Testen immer mehr Aufwand annehmen wird, wobei dieser so gering und gleichzeitig so effizient wie möglich gehalten werden soll. Da Arbeitszeiten und Kosten auch in Zukunft knapp berechnet sein werden, hat dieser Aspekt mehr Gewichtung als es für einen Außenstehenden im ersten Augenblick zu erscheinen vermag.

Literaturverzeichnis

[Canoo WebTest] Canoo WebTest Tool [online], Canoo Engineering AG, Basel, Schweiz. Verfügbar bei: http://webtest.canoo.com/webtest/manual/WebTestHome.html [Zugang am 26.Mai 2009]

[HPLoadRunner] HP LoadRunner Software [online], Hewlett-Packard Development Company, USA. Verfügbar bei: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100__ [Zugang am 19.Mai 2009]

[ISTQB-Syllabus] Certified Tester Foundation Level Syllabus, Version 2007 [online], International Software Testing Qualifications Board (ISTQB), Brüssel, Belgien. Verfügbar bei: http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf [Zugang am 19.Mai 2009]

[JUnit] JUnit Open Source Testing Framework [online], Open-Source Projekt. Verfügbar bei: http://www.junit.org/ [Zugang am 19.Mai 2009]

[Kaner99] Kaner, Cem. Falk, Jack. Nguyen, Hung Quoc (1999). Testing Computer Software, Second Edition. USA: Wiley & Sons. ISBN-13: 978-0-47135-846-6.

[Myers82] Myers, Glenford J. (1982). The Art of Software Testing. USA: Wiley & Sons. ISBN: 0-471-04328-1.

[Nguyen03] Nguyen, Hung Quoc. Johnson, Bob. Hackett, Michael. Johnson, Robert (2003). Testing Applications on the Web, Second Edition. USA: Wiley & Sons. ISBN-13: 978-0-47120-100-7.

[Pol00] Pol, Martin. Koomen, Tim. Spillner, Andreas (2000). Management und Optimierung des Testprozesses. Heidelberg: dpunkt.verlag. ISBN: 3-932-58865-7.

[Selenium] Selenium Web Application Testing System [online], Open-Source Projekt. Verfügbar bei: http://seleniumhq.org/ [Zugang am 26.Mai 2009]

[Siteur05] Siteur, Maurice M. (2005). Automate your testing! Sleep while you are working. Amsterdam: Centraal Boekhuis. ISBN-13: 978-9-03952-442-8.

[Spillner&Linz05] Spillner, Andreas. Linz, Tilo (2005). Basiswissen Softwaretest, 3. Auflage. Heidelberg: dpunkt.verlag. ISBN-13: 978-3-89864-358-0.

[TOAD] TOAD (Tool For Application Developers) [online]. Quest Software Inc., USA. Verfügbar bei: http://www.toadsoft.com/ [Zugang am 19.Mai 2009]

[W3CValidationService] The W3C Markup Validation Service [online], World Wide Web Consortium (W3C). Verfügbar bei: http://validator.w3.org/ [Zugang am 19.Mai 2009]

[WebDriver] WebDriver: A developer focused tool for automated testing of webapps [online]. Google, USA. Verfügbar bei: http://code.google.com/p/webdriver/ [Zugang am 27.Mai 2009]

Abbildungsverzeichnis

Abbildung 1: Die Kosten eines Fehlers im Entwicklungsprozess (Boehm'sche Kurve) [Siteur05]. ... 9
Abbildung 2: Ein Beispiel eines Kontrollflussgraphen [Spillner&Linz05]. ... 14
Abbildung 3: Der Testrahmen um ein Testobjekt [Spillner&Linz05]. ... 16
Abbildung 4: Testrahmen beim Blackbox-Verfahren [Spillner&Linz05]. ... 17
Abbildung 5: Testrahmen beim Whitebox-Verfahren [Spillner&Linz05]. ... 18
Abbildung 6: Überblick über die Arten von Test-Tools [Siteur05]. ... 20
Abbildung 7: Der Softwarelebenszyklus des allgemeinen V-Modells [Spillner&Linz05]. ... 25
Abbildung 8: Ansicht des Firefox Add-Ons „Selenium IDE“. ... 34
Abbildung 9: Ansicht der Sidebar des Firefox Add-Ons „Webtest Recorder Sidebar“. ... 36

Abkürzungsverzeichnis

HTML	„Hypertext Markup Language“ – Textbasierte Auszeichnungssprache zum Beschreiben und Anzeigen von Webseiten (XML-Derivat)
HTTP	„Hypertext Transfer Protocol“ – Protokoll zum Übertragen von Webseiten
ISTQB	International Software Testing Qualifications Board (http://www.istqb.org/)
PoC	„Point of Control“ – Teil des Testrahmens im dynamischen Test
PoO	„Point of Observation“ – Teil des Testrahmens im dynamischen Test
SWT	Softwaretest
TDD	Test-Driven Development (Testgetriebene Entwicklung), auch „Test first“-Ansatz genannt
W3C	World Wide Web Consortium (http://www.w3.org/)
XML	„Extensible Markup Language“ - erweiterbare Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Text