Bachelorarbeit

Daniel Löffelholz

Konzeption eines konfigurierbaren Testmanagementprozesses
in heterogenen Projektumgebungen
am Beispiel British American Tobacco

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Löffelholz

Konzeption eines konfigurierbaren Testmanagementprozesses in heterogenen Projektumgebungen am Beispiel British American Tobacco

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informatik
am Fachbereich Elektrotechnik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Bettina Buth
Zweitgutachter : Prof. Dr. Olaf Zukunft
Abgegeben am: 24.08.2008

Konzeption eines konfigurierbaren Testmanagementprozesses in heterogenen Projektumgebungen am Beispiel British American Tobacco

Daniel Löffelholz

Stichworte
Softwaretest, Software-Qualität, Testprozessmanagement, Teststrategie, Testhandbuch

Zusammenfassung

Diese Arbeit beschäftigt sich mit der konzeptionellen Verbesserung des Testmanagements bei British American Tobacco. Das Unternehmen verfügt nicht nur über eine Projektumgebung mit vielen verschiedenartigen Projekttypen und Softwaresystemen sondern auch einer Reihe unterschiedlicher Testmethoden und Vorgehensmodellen. Ziel dieser Arbeit ist mit Hilfe eines einheitlichen Testmanagementprozess die Vorgehensweise beim Testen strukturierter und effizienter zu gestalten. Um diesen Zustand zu erreichen werden zunächst der aktuelle Zustand des durchgeführten Testens und der Umfang der Projektlandschaft erhoben. Auf Basis dieser Erhebung werden Verbesserungspotentiale identifiziert und Möglichkeiten aufgezeigt wie sich die zu untersuchende Abteilung in diesen Punkten verbessern kann.

Zum Schluss dieser Arbeit erfolgt die Umsetzung der Erkenntnisse in ein Konzept. Zu den Verbesserungsvorschlägen werden konkrete Methoden und Werkzeuge evaluiert. Resultat ist ein Testhandbuch, das die Basis des zukünftigen Testprozesses bilden soll. Aus ihm können die jeweiligen Vorgehensweisen für jede Teststufe entnommen werden. Als Grundlagen dieser Arbeit dienen neben der Theorie über Testen und Testmanagement, die Arbeitserfahrungen des Autors im Unternehmen als auch die Durchführung von Interviews mit Personen aus allen beteiligten Bereichen.

Conception of a configurable testmanagement process in heterogene project environments using the example of British American Tobacco

Daniel Löffelholz

Keywords

Software-Testing, Software-Quality, Test-Processmanagement, Test-Strategy, Test-Handbook

Abstract

The thesis deals with the conception of a configurable testmanagement process for British American Tobacco. The current situation in British American Tobacco shows a strong heterogene project environment with numerous different types of projects and software systems. Besides these it consists of different test methods and process models. The objective is to analyse existing test processes and to pinpoint their strengths and weaknesses. The test management process resulting from these findings is configurable for different circumstances and ought to be applicable to every software project from maintenance to newly-developed or upgraded proprietary software. Special attention is on the case, that the concept of the process is focussed on an uncomplicated implementation and quick realization under existing conditions. The present study is based on the theoretical knowledge of testing and testprocess management, on interviews and analyses of existing documentation as well as on gained experiences in the office of BAT.

Inhaltsverzeichnis

Inhaltsverzeichnis **1**

Abbildungsverzeichnis **2**

Tabellenverzeichnis **2**

1 Einleitung **2**

1.1 Motivation 2
1.2 Problemstellung der Thesis 4
1.3 Zielsetzung 5
1.4 Vorgehensweise 6
1.5 Abgrenzung der Thematik 7
1.6 Gliederung 7

2 Grundlagen **9**

2.1 Ziele des Softwaretestens 9
2.2 Psychologie des Softwaretestens 12
2.3 Testtechniken 13
2.3.1 Statischer Test 14
2.3.1.1 Verifizierender Test 14
2.3.1.2 Analysierender Test 14
2.3.1.3 Werkzeuggestützte statische Codeanalyse 15
2.3.2 Dynamischer Test 16
2.3.2.1 Black-Box-Test 16
2.3.2.2 White-Box Test 17
2.3.2.3 Grey-Box-Tests 18
2.4 Testen im Softwareentwicklungszyklus 19
2.4.1 Low-Level-Tests 20
2.4.2 High-Level-Tests 21
2.5 Testen als Prozess 23

2.5.1 Phasenmodell ... 24
2.5.2 Organisation ... 25
2.5.3 Infrastruktur und Werkzeuge ... 25
2.5.4 Testtechniken ... 26
2.6 Kritische Faktoren ... 26
2.7 Interviews ... 28
2.7.1 Interviewstruktur ... 28
2.7.2 Fragentechniken ... 29
2.7.3 Interviewarten ... 29
2.7.4 Auswahl der Interviewtechnik ... 30

3 Analyse und Auswertung ... 31
3.3 Aufgaben der Abteilungen ... 31
3.3.1 Anforderungen ... 32
3.3.2 Entwicklungszyklus ... 32
3.4 Testen ... 34
3.4.1 Testmöglichkeiten ... 35
3.4.2 Vorgehen ... 35
3.5 Identifikation von Problemen ... 36
3.5.1 Integration in das Vorgehensmodell ... 36
3.5.2 Technik ... 37
3.5.3 Infrastruktur und Werkzeuge ... 37
3.5.4 Phasenmodell ... 38
3.6 Verbesserungsvorschläge ... 38
3.6.1 Integration in das Vorgehensmodell ... 38
3.6.2 Technik ... 39
3.6.3 Infrastruktur und Werkzeuge ... 40
3.7.4 Phasenmodell ... 41

4 Konzept ... 42
4.1 Maßnahmen ... 43
4.1.1 Integration in das Vorgehensmodell ... 43
4.1.1.1 Komponententest ... 43
4.1.1.2 Integrationstest ... 46

4.1.1.3 Systemtest 48
4.1.2 Technik 49
4.1.3 Infrastruktur und Werkzeuge 50
4.1.3.1 Werkzeuggestützte statische Analyse 51
4.1.3.2 Komponententests und Code-Abdeckung 53
4.1.3.4 Kontinuierliche Integration 56
4.1.4 Phasenmodell 58
4.2 Konzept des Testhandbuchs 58
4.2.1 Ziele für Tester und Entwickler 58
4.2.1 Ziele für das Management 59
4.2.3 Struktur 60
4.2.4 Beispiel: Integrationstest 61

5 Zusammenfassung und Ausblick 62
5.1 Zusammenfassung 62
5.2 Ausblick 63
5.3 Fazit 64

Abkürzungsverzeichnis 67

Literaturverzeichnis 68

Internetquellen 70

Anhang 71
Anhang A: Fragenkatalog 72
Anhang B: Beispielkapitel Testhandbuch - Integrationstest 74

Abbildungsverzeichnis

Abbildung 1: Qualitätskosten nach Juran und Gryna (1988) ... 11
Abbildung 2: Hierarchie der Testtechniken nach Zhu *et al.* (1997) ... 13
Abbildung 3: Dynamischer Test nach Zhu *et al.* (1997) ... 16
Abbildung 4: Black-Box-Test ... 17
Abbildung 5: White-Box-Test ... 18
Abbildung 6: Allgemeines V-Modell nach Boehm (1982) ... 20
Abbildung 7: Auftragen der Teststufen auf „Cost of Change"-Kurve von Boehm (1982) ... 21
Abbildung 8: Die vier Eckpfeiler des strukturierten Testens nach Koomen und Pol (1999) ... 23
Abbildung 9: Phasenmodell des Testprozesses nach Koomen und Pol (1999) ... 24
Abbildung 10: Entwicklungsprozess bei *GSD-CS* ... 33
Abbildung 11: Vorgehen mit V-Modell bei GSD-CS ... 34
Abbildung 12: Teststufen ... 43
Abbildung 13: Flussdiagramm Komponententest ... 46
Abbildung 14: Interaktion zwischen Repository, Entwickler und Integrationsserver ... 47
Abbildung 15: Arbeitsablauf des Integrationstests bei GSD-CS ... 56
Abbildung 16: Testhandbuch: Kapitel 2 ... 60
Abbildung 17: Testhandbuch: Kapitel 3 ... 60

Tabellenverzeichnis

Tabelle 1: Phasen eines Reviews 14
Tabelle 2: Kategorien von Reviews nach IEEE 1028-1997.. 15
Tabelle 3: Beschreibung der Phasen einer Teststufe.. 24

Kapitel 1

Einleitung

In diesem Kapitel wird zunächst auf die Motivation des Autors sich mit dem Thema zu befassen eingegangen. Dadurch soll sowohl ein Eindruck von der Notwendigkeit als auch der Herausforderung des Testens als Qualitätssicherungsmaßnahme entstehen. Das Unternehmen British American Tobacco (BAT) wird als Beispiel einer Ausgangssituation für Testprozessverbesserungen herangezogen. Damit sich der Leser ein Bild über die vorgefundenen Gegebenheiten machen kann, wird die aktuelle Situation im Unternehmen beschrieben. Im Abschnitt 1.3 wird auf die Ziele eingegangen die im Laufe der Arbeit erreicht werden sollen. So wird das Vorgehen bei der Entwicklung eines Vorgehens zur Verbesserung des Testprozesses erläutert.

1.1 Motivation

Obwohl das Testen die wohl am weitesten verbreitete Maßnahme der Qualitätssicherung in der Softwareentwicklung darstellt, ist es anscheinend dennoch der am wenigsten verstandene Teil des Entwicklungsprozesses (Whittaker, 2000). Oft wird der Testprozess als zu kostenintensiv und langwierig wahrgenommen woraus resultiert, dass in Zeiten von knappen Budgets und festen Fristen viele Entwickler als Erstes am Testen sparen. Demgegenüber steht dennoch ein steigender Anspruch an die Softwarequalität. Wer in der heutigen globalisierten Welt dem Wettbewerbsdruck standhalten will, kommt um eine kosteneffiziente Entwicklung qualitativ hochwertiger Software nicht herum.

Viele Unternehmen verfügen über kein strukturiertes Testkonzept, weil das Testen von Software erst seit einer - selbst für die IT-Welt - kurzen Zeit als wichtige, ingenieurmäßige Arbeit wahrgenommen wird. Gründe sind hierfür in der fortschreitenden Industrialisierung

der Softwareentwicklung, welche aktuelle Trends wie *Model Driven Development*[1] oder *Rapid Unified Process*[2] belegen, zu finden.

„Those (…) who have been working in IT for a long period of time will remember the days when testing was the task of the most junior person on the team."
Broeders, CAP Gemini NV (Koomen, 1999)

Eine 2005 durchgeführte Studie zeigt auf, dass Testen zwar von den IT-Verantwortlichen als strategisch wichtig angesehen wird, jedoch die gelebte Praxis oft mangelhaft ist. 68% aller IT-Verantwortlichen gaben an, dass Produkte und Services ohne ausreichende Tests auf den Markt kommen. Dies hat schwerwiegende Folgen: 89% der Befragten gaben an, dass 48 Stunden nach Inbetriebnahme problematische Fehler auftraten. 88% der IT-Manager waren sich sicher, dass ein besserer Testprozess zu einer Reduzierung der gesamten Softwareentwicklungskosten führen würde. 74% waren überzeugt, dass ein stukturierter Testansatz wesentlich dazu beitragen würde die Entwicklungskosten zu senken. [3]

Neben festgefahrenen Strukturen und mangelndem Fachwissen ist die unzureichende Durchführung des Testens oft damit zu begründen, dass der Nutzwert vom Softwaretesten nur schwer zu messen ist und es vor allem von den ausführenden Personen als notwendiges Übel wahrgenommen wird: ein unkontrollierter Prozess der nichts Neues schafft sondern lediglich Unzulänglichkeiten einer oft als „fertig" angesehenen Komponente aufdeckt und somit destruktiv ist. Software als Produkt menschlicher Intelligenz ist im Allgemeinen fehlerbehaftet (Wiener, 1994). In Zeiten globaler Teams, kurzer Releasezyklen und komplexer werdender Systeme sollte somit auf ein ausreichendes Testen nicht verzichtet werden, falls man qualitativ hochwertige Software herstellen will.

Die Einführung eines strukturierten Testprozesses fördert die Effektivität und Transparenz des Testens. Die dadurch erreichte höhere Qualität des Softwareprodukts steigert die Kundenzufriedenheit und senkt die Kosten für Nachbesserung und Wartung.

---

[1] Model Driven Developement ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.
[2] Der Rational Unified Process ist ein objektorientiertes Vorgehensmodell zur Softwareentwicklung.
[3] Die europaweite Studie der LogicaCMG „Testing Times for Board Rooms" wurde im September 2005 veröffentlicht. Ingesamt wurden 255 detaillierte Interviews von dem unabhängigen Marktforschungsunternehmen Coleman Parks in England, den Niederlanden und Schweden durchgeführt.

1.2 Problemstellung der Thesis

Die British American Tobacco Plc (BAT) mit Hauptsitz in London ist weltweit der zweitgrößte Produzent von Tabakwaren. Das Unternehmen verfügt über ein Portfolio von mehr als 300 lokalen und internationalen Marken bei einer Präsenz in 180 Märkten. Ihre Tochtergesellschaften und Beteiligungen produzieren in 47 Fabriken in 40 Ländern insgesamt 684 Milliarden Zigaretten jährlich und beschäftigen weltweit mehr als 53000 Mitarbeiter.

Als bedeutendste Einzelgesellschaft der Holding British American Tobacco (Industrie) GmbH ist British American Tobacco (Germany) GmbH, Hamburg, für die Herstellung und die Vermarktung aller Tabakerzeugnisse des Unternehmens in Deutschland verantwortlich. Dazu gehören neben Zigaretten auch die übrigen Tabakwaren wie klassische und vorportionierte Feinschnitttabake, Filtercigarillos sowie Cigarren und Pfeifentabake. Sowohl der Handel als auch die Konsumenten in Deutschland können heute von British American Tobacco über die gesamte Sortimentsbreite aus einer Hand bedient werden. (web: BAT)

Der Hauptanteil informationstechnologischer Aufgaben wird inzwischen von der *Group Service Delivery* (GSD) als globaler *shared services provider* übernommen. GSD bietet den BAT Endmärkten interne und externe IT-Dienste an. Eine Unterabteilung von GSD bildet der Bereich *Customer Solutions* (CS), welcher die Beziehung zum Kunden im Sinne von Zufriedenheit und Verbesserungsmöglichkeiten organisiert. Qualität, Effizienz und Kundenzufriedenheit sind im Rahmen der Gruppenstrategie wichtige Punkte, die es stetig zu verbessern gilt. Insbesondere die Steigerung der Qualität gehört zu den wichtigsten Zielen der Gruppe. (web: GSD)

Neben der Entwicklung von .NET und Windows Forms Applikationen in Visual Basic und Visual C# findet auch eine verstärkte Entwicklung im datawarehouse-nahem Umfeld z.B. mit MDX statt. Aufgrund der verschiedenen Technologien existiert eine große Anzahl an unterschiedlichen Software- und Projekttypen verschiedenen Umfangs mit unterschiedlichen Teamgrößen. Neben dem Einsatz von Proprietärsoftware existieren sowohl zahlreiche angepasste und fremdentwickelte Softwaresysteme als auch Eigenentwicklungen unterschiedlichster Komplexität. Es besteht kein einheitlicher Testprozess und damit auch keine einheitliche Definition von anzuwendenden Testmethoden, weshalb die Abteilung ihr momentan durchgeführtes Testen verbessern möchte. Dies soll vor allem mit Hilfe von Vereinheitlichung und Strukturierung des Vorgehens geschehen. Das unter Beachtung dieser

Vorraussetzungen entstehende Konzept soll anschließend schnell und einfach umgesetzt werden können.

1.3 Zielsetzung

Ziel ist es, den aktuell bestehenden Testprozess bei *BAT GSD-CS* zu untersuchen, um anschließend mit Hilfe eines vom Verfasser entwickelten Maßnahmenkatalogs zukünftigen Testphasen eine strukturierte Vorgehensweise zu geben. Strukturiert bedeutet hierbei, dass durch den Einsatz einer dokumentierten Menge an Aktivitäten, Prozeduren, und Techniken möglichst viele Aspekte des Testprozesses abgedeckt werden. Weiterhin werden diejenigen Komponenten des aktuellen Testprozesses mit dem größten Verbesserungspotential identifiziert. In Abstimmung mit dem Management wird anschließend konzipiert welche Aufgaben umgesetzt werden müssen um das Testen auf eine qualitativ höhere Stufe zu heben.

Neben diesen eher „funktionalen“ Anforderungen (Strukturiertheit, Transparenz, Effizienzsteigerung) lassen sich folgende „nicht-funktionale“ Anforderungen an das Konzept des resultierenden Testprozesses festhalten:

Einfachheit
Es soll mit simplen Mitteln ein möglichst großer Effekt erzielt werden. Eine zu hohe Komplexität der einzuführenden Maßnahmen ist ein großes Risiko für die erfolgreiche Umsetzung. Wenn die konzipierten Maßnahmen nicht oder nicht effektiv eingesetzt werden, sind sie nutzlos. Simplizität und Klarheit der Maßnahmen fördert das Verständnis und können somit das Risiko senken.

Akzeptanz
Ohne Akzeptanz wird kein oder kein effektiver Einsatz der konzipierten Maßnahmen erzielt und man erhält somit keinen Nutzen. Dies betrifft Akzeptanz von Seiten des Managements wie auch seitens der Entwickler und der Tester.

Geringe Kosten
Ein wichtiger Faktor, insbesondere für die Steigerung der Akzeptanz seitens des Managements, sind geringe Kosten. In Zeiten knappen Budgets und steigendem

Wettbewerbsdrucks muss auf die Kosten geachtet werden. Hier gilt das Optimumprinzip als Ausprägung des ökomischen Prinzips (Walter und Wünsche, 2005): So geringe Kosten wie möglich, so hohe Kosten wie nötig. Ein möglichst optimales Kosten/Nutzen-Verhältnis schließt dies mit ein.

Nachhaltigkeit

Die Abteilung soll langfristig von den konzipierten Maßnahmen profitieren können. Prozessänderungen sind fast immer mit Initialaufwand verbunden. Die Stärken liegen hier vor allem in der langfristigen Verbesserung anstatt in schnellen Erträgen. Dazu gehört, dass das Konzept auch dementsprechend vermittelt und adäquat dokumentiert wird.

1.4 Vorgehensweise

In einem ersten Schritt ist eine umfassende Situationsanalyse notwendig. Diese deckt folgende Bereiche ab:

- Eingesetzte Technologien
- Vorhandenes Wissen der Mitarbeiter
- Identifikation der beteiligten Personen und Rollen
- Projektstrukturen
- Umfang des betriebenen Testens

Als Basis für die Analyse dienen zum einen Interviews mit Vertretern aller am Testprozess beteiligter Rollen (z.B. Management, Tester, Testmanagement, Designer, Entwickler, Benutzer). Zum anderen erfolgt eine Auswertung im Unternehmen vorhandener Dokumente zur generellen Projektdurchführung und Testvorgehen. Ergänzend hierzu wird durch eine umfassende Literatur- und Theorierecherche der Versuch unternommen eine zur spezifischen Situation bei GSD-CS passendes Vorgehen zur Testprozessverbesserung zu finden. Diese soll letztendlich als theoretische Grundlage herangezogen werden.

Die gewonnenen Informationen werden anschließend analysiert und die Eigenschaften und Gemeinsamkeiten der verschiedenen Projekte herausgearbeitet. Weiterhin sind die Bestimmung des Reifegrads des betriebenen Testens als auch die Identifikation von Risiken

und von wiederkehrenden Fehlerpotenzialen Untersuchungsgegenstand. Dieser Analyse wird der Sollzustand gegenübergestellt, um darauf aufbauend Maßnahmen zu konzipieren, die zu diesem gewünschten Zustand führen sollen.

1.5 Abgrenzung der Thematik

Aufgrund der Vielseitigkeit der Thematik beschränkt sich diese Arbeit auf Maßnahmen deren Umsetzung in Anbetracht von Aufwand und aktuellem Zustand realistisch ist. Somit ist eine Implementierung des entwickelten Prozesses nicht Bestandteil der Untersuchung. Bedingt durch die vorhandene Projekt- und Softwarevielfalt werden vor allem generelle Methoden beschrieben die auf möglichst viele Projekttypen anwendbar sind.

1.6 Gliederung

Diese Bachelorarbeit ist in folgende Kapitel unterteilt:

- Kapitel 1. Einleitung
  - *In diesem Kapitel wird zu erst auf die Motivation des Autors diese Arbeit zu verfassen eingegangen. Anschließend wird die aktuelle Situation bei BAT beschrieben, damit sich der Leser ein Bild der vorgefundenen Situation machen kann. Es wird auf die Ziele eingegangen, die der Autor im Laufe der Arbeit umzusetzen versucht. Weiterhin wird das Vorgehen bei der Entwicklung eines Verfahrens zur Verbesserung des Testprozesses erläutert.*

- Kapitel 2. Grundlagen
  - *In diesem Abschnitt werden einige theoretische Grundlagen dieser Bachelorarbeit beschrieben, um die Verständlichkeit der Ausführungen zu gewährleisten. Diese Grundlagen bilden somit das Fundament der Untersuchungen und Annahmen.*

- Kapitel 3. Analyse und Auswertung:
  - *In diesem Kapitel wird zunächst das Unternehmen beschrieben und die Vorgehensweisen der Abteilungen bezüglich Softwareentwicklung und Testen dargestellt und analysiert. Es werden Verbesserungspotentiale identifiziert und konzeptionelle Verbesserungsmaßnahmen aufgezeigt.*

- Kapitel 4. Konzept
  - *Dieses Kapitel ist in zwei Abschnitte unterteilt. Im ersten Abschnitt werden anhand der in Kapitel 3 geschilderten Analyse-Methoden und Vorgehensweisen praxisnah beschrieben um die Verbesserungsmaßnahmen umzusetzen. Im Abschnitt 2 wird auf der Basis dieser Methoden und Maßnahmen ein Testhandbuch konzipiert.*

- Kapitel 5. Zusammenfassung und Ausblick
    - *Hier wird das Ergebnis der Arbeit evaluiert, gemachte Erfahrungen aufgeführt, und Vor- und Nachteile des angewandten Vorgehens bezüglich der hier vorhandenen Situation beschrieben.*
      Ein Ausblick auf die nächsten Schritte die BAT gehen sollte um den Prozess weiter zu verbessern runden das Kapitel ab.

- Kapitel 6 Anhang:
    - *Im Anhang befinden sich begleitende Dokumente dieser Arbeit.*

Kapitel 2

Grundlagen

Eine Definition der in der Arbeit verwendeten Fachbegriffe, Methoden und Technologien ist notwendig, um die Verständlichkeit der Ausführungen zu gewährleisten. Somit steht die Erläuterung der Grundlagen am Anfang. Nachdem zunächst auf die generellen Ziele des Softwaretestens und dessen Psychologie eingegangen wird, werden die grundlegenden Testtechniken kurz erläutert. Auch wird aufgezeigt, wo diese Techniken in einen Vorgehensmodell der Softwareentwicklung eingesetzt werden und warum Testaktivitäten als Prozess parallel zur Softwareentwicklung ausgeführt werden sollten. Welche Punkte dabei beachtet werden müssen und mit welchen Risiken man bei der Einführung eines Testprozesses rechnen muss wird anschließend geschildert. Da Interviews im Rahmen dieser Arbeit eine große Rolle bei der Informationsbeschaffung über den aktuellen Ist- und den gewünschten Sollzustand spielen, werden im Anschluss daran Grundlagen zu Interview – und Fragetechniken aufgeführt.

2.1 Ziele des Softwaretestens

„The only man who never makes mistakes is the man who never does anything."
Theodore Roosevelt

Wie bereits in Kapitel 1.1 erläutert, beinhaltet jede Software potentiell Fehlerzustände die sich in der Regel als Fehlwirkungen bemerkbar machen. Somit liegt hier eine Nichterfüllung einer Anforderung oder eines Anspruchs vor und folgendermaßen ein Mangel im Produkt welcher behoben werden muss. Für den Softwareentwickler bedeutet dies, dass vor Inbetriebnahme bzw. Auslieferung des Produkts mit Hilfe geeigneter Testmethoden nach Fehlern gesucht werden sollte, um einerseits durch Fehler- und Risikominimierung die Kundenzufriedenheit zu erhöhen sowie andererseits die Kosten für Wartung und Nachbesserung zu senken. Softwaretesten soll also die Effizienz von Entwicklungs- und Wartungsprozess als auch die

Qualität des resultierenden Programms verbessern. Es ist aber entgegen weit verbreiteter Annahme keine Methode um die Fehlerfreiheit einer Software zu belegen. Sie kann nur dazu benutzt werden, die Anwesenheit von Fehlern, und nicht deren Abwesenheit zu belegen (Djikstra, 1972). Dies begründeten auch Kaner *et al.* (1999) dadurch dass:

(1) der Bereich der möglichen Eingaben zu groß ist
(2) es zu viele mögliche Eingabepfade gibt, und
(3) Design und Spezifikationsaspekte schwer zu testen sind.

Feststellung 1 kann an einem Beispiel von Beizer verdeutlicht werden:
Nach Beizer werden für einen einzelnen String mit 10 Zeichen 2^{80} Testfälle[4] benötigt um das Ergebnis auf Korrektheit zu überprüfen. Die Problematik der Komplexität ist auch in Punkt 2 enthalten, jedoch wird sie zusätzlich dadurch verschärft, dass nicht jeder Pfad eines Programms ausführbar ist und dass es Eingaben geben kann für die das Programm nicht anhält Es kann also ein Halteproblem [5] vorliegen welches bekanntermaßen nicht entscheidbar ist.

Zu Punkt 3 kann man auch nicht-funktionalen Qualitätsmerkmale wie Benutzbarkeit zählen. Unterschiedliche Anwender besitzen unterschiedliche Vorstellungen und Anforderungen an die Benutzbarkeit eines Programms, somit lässt sich ein Programm nie auf die vollständige Erfüllung dieses Merkmals hin testen.

Mit diesen theoretischen Limitierungen des Testens im Hinterkopf ist erkennbar, dass Softwaretesten immer ein Kompromiss zwischen Genauigkeit und Aufwand darstellt. (Young und Taylor, 1989). Young und Taylor (1989) schlussfolgerten, dass es unmöglich sei eine Testmethode vorzuschlagen und zu entwickeln, die vollständig korrekt und auf alle Programme anwendbar ist.

Das Aufdecken von Fehlern und die gleichzeitige Erhöhung der Qualität des Softwareprodukts ist eines der beiden Hauptziele des Testens. Das zweite Ziel ist es, die

---

4 Angenommen der String ist nach erweitertem ASCII encodiert, gibt es 256 = 28 mögliche Zeichen pro Stelle. Bei 10 Stellen sind dies 280

5Das Halteproblem ist ein Problem aus der theoretischen Informatik. Es besteht darin dass mit einer bestimmten Berechnung überprüft werden soll ob ein Programm terminiert oder nicht. Touring bewies als Erster dass dieses Problem nicht lösbar ist.

Unsicherheit bezüglich der Qualität des Software-Produkts zu reduzieren. Beispielsweise durch das Abschätzen der Restfehleranzahl ist es möglich, ein detaillierteres Bild der Qualität der Software zu erhalten.

"You can't control what you can't measure" DeMarco (1982)

Dennoch kostet Testen viel Zeit und Geld. Sollten die Testkosten die potentiellen Fehlerbehebungskosten übertreffen wurde das Ziel verfehlt. Die Herausforderung hierbei ist das Minimum der Summe der geschätzten Fehlerbehebungskosten und Testkosten zu treffen Der kostenminimale und optimale Bereich liegt gemäß Abbildung 1 (Juran und Gryna, 1988), welche die Beziehung der Fehlerkosten und Testkosten verdeutlicht, im Schnittpunkt beider Kurven.

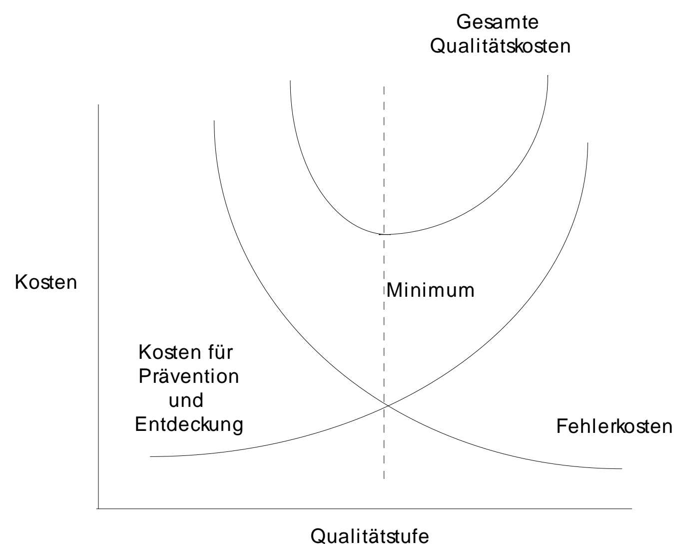


Abbildung 1: Qualitätskosten nach Juran und Gryna (1988)

Um nun fundiert bewerten zu können wann weitere Tests und weitere qualitätssteigernde Maßnahmen ökonomisch nicht mehr tragbar sind, müssen die Qualität des Produkts sowie die bereits angefallenen Kosten für Prävention und Entdeckung von Fehlern bekannt sein.

Aus der Grafik lässt sich weiterhin ableiten auf welche Weise die Qualität der Software verbessert werden kann ohne die Gesamtkosten zu steigern. Sollten die Testkosten verringert werden können, kann mehr getestet werden bevor der Punkt der minimalen Gesamtkosten

erreicht wird. Um die Testkosten in einer unstrukturierten Umgebung zu reduzieren ist eine Strukturierung des Testprozess mit der Identifikation der höchsten Qualitätsrisiken, mit der Priorisierung der verschiedenen Aufgaben der Qualitätssicherung und des Testens und mit dem Anwenden der richtigen Testmethoden notwendig. Dies versucht der Autor mit dieser Arbeit innerhalb von BAT umzusetzen..

Dieser Abschnitt soll unter anderem verdeutlichen, dass das effektive Testen von Systemen nicht trivial ist. Neben den aufgeführten theoretischen Limitierungen und ökonomischen Anforderungen darf man außerdem nicht den menschlichen Faktor außer acht lassen. Als konzeptionierende und ausführende Instanz ist der Mensch im Softwaretesten omnipräsent. Die daraus resultierenden Herausforderungen werden deshalb in den Abschnitten 2.2 (Psychologie des Softwaretestens) und 2.6 (Kritische Faktoren) betrachtet.

2.2 Psychologie des Softwaretestens

"Kein Produkt menschlicher Intelligenz kommt fehlerfrei zur Welt. Wir formulieren Sätze um, trennen Nähte wieder auf, setzen Pflanzen um, planen Häuser neu und reparieren Brücken. Warum sollte es uns mit Software anders gehen?" Wiener (1994)

Die Aussage von Lauren Ruth Wiener wird unter den meisten Softwareentwicklern Zustimmung finden, dennoch werden eigene Fehler nicht gerne eingestanden. Ein identifizierter Fehlerzustand bedeutet zusätzliche Arbeit die ein Entwickler investieren muss. Generell ist auch eine optimistischere Einschätzung des eigenen Codes vorhanden und so werden kritische Abschnitte oft lediglich mit wenigen oder nicht wirklich sinnvollen Testfällen geprüft. Dem Entwickler fehlt der nötige Abstand zum eigenen erstellten Produkt. Das Testen wird als destruktive Maßnahme wahrgenommen, ganz im Gegensatz zum konstruktiven Programmieren.

Zusätzlich können Designfehler, wie falsch verstandene Requirements, durch den Autor des Codes selten entdeckt werden, da dieser beim Entwurf der Testfälle wieder ähnliche Fehler begehen wird. Somit ist ein für viele Tests ein unabhängiges Testteam unverzichtbar. Eine Ausnahme bilden hier die strukturbasierten Tests für die eine Kenntnis des Codes notwendig

ist. Hier würde der Aufwand der Einarbeitung eines komplett unabhängigen Testers den Nutzen des Testens oft zu Nichte machen. Jedoch können beispielsweise mit Walkthroughs[6] die unabhängigen Tester integriert werden. Hierbei ist jedoch zu beachten, dass die Tester durch die Präsentation selbst nicht vollkommen unbeeinflusst testen können. Falsche Annahmen und Verständnisprobleme bei der Anforderungsanalyse können vom Autor auf das Testteam übergehen.

Zusätzlich herrscht oft ein stetiger Druck seitens der Kunden oder des Managements, deren Ziel eine weitestgehende Kosten- und Aufwandreduktion ist, wogegen mit den Kunden oft Probleme in der Benutzer-Entwicklerkommunikation auftreten. Die positiven Auswirkungen des Testens lassen sich im Nachhinein nur aufwändig numerisch messen und folglich wird das Testen oft als zu kostenintensiv wahrgenommen. Dies liegt jedoch meistens in einer falschen Auffassung von Qualitätssicherung und der Bedeutung von Softwarequalität oder an der mangelhaften Durchführung des Testens.

2.3 Testtechniken

Programme oder Programmteile lassen sich mit Hilfe von unterschiedlichen Testtechniken auf Fehler überprüfen. Grundsätzlich ist zu Unterscheiden ob sich das Programm beim Testen in Ausführung befindet oder nicht. In Abbildung 2 sind dies auf Ebene 2 die dynamischen und statischen Tests. Auf weitere Unterscheidungen wird folgend weiter eingegangen.

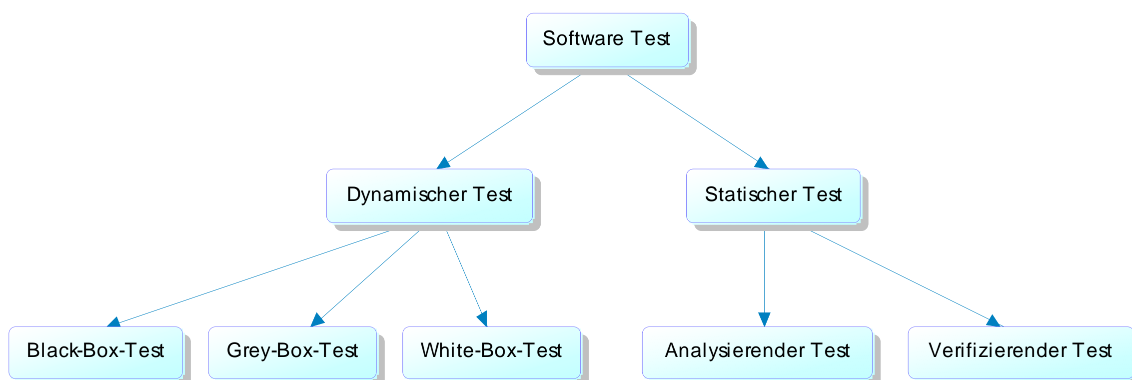


Abbildung 2: Hierarchie der Testtechniken nach Zhu *et al.* (1997)

[6]Bei Walkthroughs erläutert der Autor seinen Code einer Gruppe von Experten, bevor man gemeinsam das Programm analysiert

2.3.1 Statischer Test

Ein statischer Test ist die Prüfung einer Komponente oder eines Systems ohne deren Ausführung. Statische Tests lassen sich gemäß Abbildung 2 grundsätzlich in analysierende und verifizierende Tests kategorisieren.

2.3.1.1 Verifizierender Test

Der verifizierende Test prüft ein Programm auf Richtigkeit gemäß einer formalen Anforderung. Es werden oft mathematische oder Vorgehensweisen der formalen Informatik eingesetzt. Aufgrund des hohen Aufwandes des vollständigen Tests von Software, sollte er nur auf extrem sicherheitskritische Programmteile angewendet werden.

2.3.1.2 Analysierender Test

Analysierende Tests werden typischerweise mit Hilfe von Reviews vollzogen. Reviews sind die Bewertung eines Softwareprodukts oder von Projekten zur Überprüfung der geplanten Arbeitsergebnisse und möglichen Verbesserungspotentialen. Ein typisches Review beinhaltet gemäß Tabelle 1 folgende Phasen:

Planung	Auswahl der beteiligten Personen und Festlegen von Bedingungen
Kick-Off	Erläuterung von Zielen sowie Überprüfen der Vorbedingungen
Individuelle Vorbereitung	Vermerken von Kommentaren und potentiellen Fehlerquellen
Reviewsitzung	Präsentation und Diskussion der Ergebnisse, Festlegen von Konsequenzen
Überarbeitung	Beheben der Fehler, typischerweise durch den Autor
Nachbearbeitung	Überprüfung der Nachbedingungen und Test-Ende-Kriterien

Tabelle 1: Phasen eines Reviews

Nach IEEE Standard 1082-1997 gibt es folgende Kategorien von Reviews:

Technische Reviews	Eine Überprüfung eines entscheidenden Dokumentes (z.B. Architekturentwurf einer Komponente) durch qualifizierte Mitarbeiter unter anderem auf Anforderungen sowie Angemessenheit bezüglich den geplanten Einsatz darstellt
Management-Reviews	Eine systematische Überprüfung des Softwareentwicklungs-prozesses und von Aktivitäten wie Softwarewartung und Betrieb.
Software-Inspektionen	Aufdecken von Fehlern in einer Komponente
Walkthrough	Gemeinschaftliches Suchen von Verbesserungspotential und Fehlern im Quellcode
Audits	Allgemeine Überprüfung von Vorgaben, Richtlinien, und Standards auf Einhaltung

Tabelle 2: Kategorien von Reviews nach IEEE 1028-1997

2.3.1.3 Werkzeuggestützte statische Codeanalyse

Eine automatisierte statische Codeanalyse gehört zu den statischen Tests. Sie ist ein Mittel zur frühzeitigen präventiven Fehlerfindung oder dem Identifizieren von fehlerträchtigen Stellen. Passende Werkzeuge liefern Maßzahlen zu verschiedenen Charakteristika eines Dokumentes. Vorraussetzung ist, dass das Dokument eine formale Struktur aufweist (XML, UML, Quelltext). Hier sind sehr umfangreiche unterschiedliche Analysen denkbar. Als Basisfunktionalität bieten die meisten Werkzeuge eine Überprüfung auf syntaktische Konventionen. Denkbar wären ein Ausschließen von Konstrukten, die:

- Fehler begünstigen
- Qualität mindern
- Portabilität verhindern
- Verständlichkeit reduzieren

Weitere mögliche Funktionalitäten sind die Aufdeckung von Datenflussanomalien[7] oder Deadlocks[8].

2.3.2 Dynamischer Test

Bei dynamischen bzw. ausführungsbasierten Tests befindet sich das Programm während des Tests in der Ausführung. Sie lassen sich nach Abbildung 2 in drei Kategorien unterteilen: Black-Box-, White-Box-, und die Mischform Grey-Box-Tests (Abbildung 3).

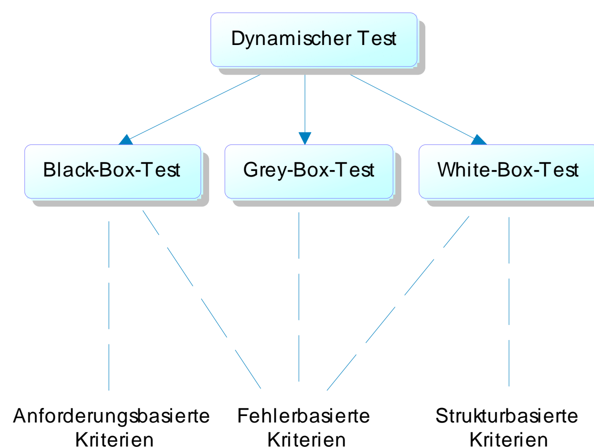


Abbildung 3: Dynamischer Test nach Zhu *et al.* (1997)

2.3.2.1 Black-Box-Test

Black-Box-Testen ist eine auf den Anforderungen basierende Testmethode, bei der ohne Kenntnis der Implementation oder inneren Funktionsweise getestet wird (Abbildung 4). Aus den formalen oder informalen Spezifikationen werden Testfalle erarbeitet die den Funktionsumfang der Software repräsentieren sollen. Dazu ist es nötig, dass die Anforderungen möglichst genau erfasst worden sind, da ansonsten das Aufstellen der Testfälle sehr aufwändig oder zum Teil nicht möglich sein könnte. Allein der Entwurf von Black-Box-Tests kann helfen Lücken in der Spezifikation aufzudecken, da beim Erstellen der

7 Fehlerhafte Sequenz von Attribut- bzw. Objektzugriffen. z.B. der Zugriff auf eine Variable oder ein Attribut bevor es erzeugt wurde (Null-Pointer Exception) oder die zweimalige Wertzuweisung ohne Verwendung des ersten Wertes

8 Sperr- oder Synchronisationsmechanismen die nicht korrekt verwendet werden, mit dem Resultat dass sich verschiedene Programmteile beim Ressourcenzugriff gegenseitig blockieren.

Testfälle auf Basis fehlerhafter oder unvollständiger Anforderungen meist Unklarheiten entstehen.

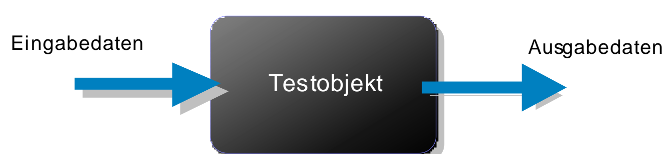


Abbildung 4: Black-Box-Test

Da der Autor durch die Kenntnis der internen Repräsentation bei der Auswahl der Testdaten beeinflusst werden kann, sind Black-Box-Tests in der Regel von einem vom Autor unabhängigen Testteam durchzuführen und zu entwerfen. Die geeignete Auswahl der Testdaten spielt jedoch eine wichtige Rolle und so gibt es dazu unterschiedliche Techniken. Als Beispiel anzuführen wäre die Äquivalenzklassenbildung, bei der Anhand der Spezifikation versucht wird sich ähnlich zu verhaltene Mengen von Eingaben zu identifizieren aus denen dann jeweils ein oder mehrere Repräsentanten getestet werden. Eine weitere Methode, die sich mit der Äquivalenzklassenbildung gut ergänzt ist die Grenzwertanalyse, bei der die Grenzen der Äquivalenzklassen oder auch die Ober und Untergrenze des spezifizierten Eingabebereichs zusätzlich getestet werden, da hier oft Fehlerwirkungen zu erwarten sind. Dadurch erzielt man ein gutes Aufwand-Risikominimierungsverhältnis (Spillner, 2005).

Weitere Auswahlverfahren sind zum Beispiel (Spillner, 2005):

- Entscheidungstabellen
- Ursache- Wirkungsgrad
- Risikoanalyse
- Zustandsbezogene Tests
- Erfahrungsbasiertes Testen (z.B. Smoke Test, Error Guessing)

Auf diese Verfahren wird jedoch nicht weiter eingegangen.

2.3.2.2 White-Box Test

White-Box-Testen, oder auch programmbasiertes Testen ist eine Testmethode, bei der im Gegensatz zum Black-Box-Testen der Programmcode verwendet wird. Es sind also genaue Kenntnisse über die innere Funktionsweise erlaubt und sogar erwünscht (Abbildung 5). Ziel des White-Box-Testens ist normalerweise Fehler in Teilkomponenten aufzudecken und nicht

wie beim Black-Box-Test die Erfüllung der Spezifikation zu überprüfen. Es wird also lediglich getestet ob das Programm funktioniert, nicht ob es sich bezüglich der Spezifikation richtig verhält.

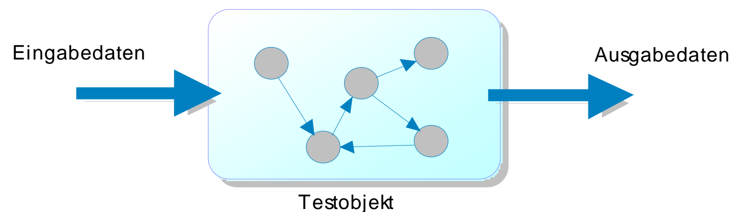


Abbildung 5: White-Box-Test

Es existieren verschiedene Hinlänglichkeitskriterien für Testfälle in Bezug auf die Abdeckung des Quellcodes (oft *Code Coverage)*. Dazu zählen:

- Zeilenüberdeckung: Durchlaufene Zeilen im Verhältnis zu der Gesamtanzahl von Zeilen
- Anweisungsüberdeckung: Durchlaufene Anweisungen im Verhältnis zu der Gesamtanzahl von Anweisungen
- Bedingungsüberdeckung: Durchlaufene Bedingungen im Verhältnis zu der Gesamtanzahl von Bedingungen
- Zweigüberdeckung: Durchlaufene Verzweigungen eines Programms im Verhältnis zu der Gesamtanzahl von Verzweigungen
- Pfadüberdeckung: Anzahl der durchlaufenen Pfade (Konkatenation der Zweige) im Verhältnis zu der Gesamtanzahl von Pfaden

Die Verantwortlichen für die Konzipierung und Durchführung der White-Box-Tests sind meistens im selben Team wie die Entwickler, wenn nicht sogar die Entwickler selbst. Durch die geforderten Kenntnisse des Programmcodes wäre eine Einarbeitung eines unabhängigen Teams meistens zu aufwändig.

2.3.2.3 Grey-Box-Tests

Grey-Box-Testen ist ein Verfahren aus dem Extreme Programming. Hierbei sollen die Vorteile von Black- und White-Box-Testen kombiniert werden. Die Testfälle werden von den

Entwicklern vor der Implementation geschrieben (*Test-First-Design*[9]), sprich bevor der Entwickler den Code kennt.

2.4 Testen im Softwareentwicklungszyklus

In den meisten Vorgehensmodellen der Softwareentwicklung nehmen Testaktivitäten einen festen Platz ein. Je nach Modell sind diese in einzelne Teststufen eingeteilt. Dies hat den Hintergrund, dass unterschiedliche Tests erforderlich sind um herauszufinden inwiefern Programme gemäß dem technischen und fachlichen Entwurf arbeiten und inwieweit das gesamte System den Anforderungen des Kunden entspricht. Zur Organisation dieser unterschiedlichen Tests kommen verschiedenen Testtechniken auf unterschiedlichen Teststufen zum Einsatz. (Koomen, 1999).

„Eine Teststufe ist eine Gruppe von Testaktivitäten, die gemeinsam organisiert und gelenkt werden“ Koomen (1999)

Ausgehend von einem sequenziellen[10] Entwicklungsmodell, wie zum Beispiel des allgemeinen[11] V-Modells (folgend V-Modell) (Boehm, 1982) kann man folgende Teststufen identifizieren (Abbildung 6):

- Komponententest
- Integrationstest
- Systemtest
- Akzeptanztest

[9] „Test First“ ist eine Regel des Extreme Programmings welche besagt, dass Tests grundsätzlich vor der Implementierung geschrieben werden sollen.

[10] Neben den sequenziellen, gibt es auch noch die iterativen Vorgehensmodelle. Da BAT GSD-CS nach einer ersten Vorabanalyse ein sequentielles Vorgehen einsetzt, wird hier exemplarische das allgemeine V-Modell verwendet.

[11] Der Zusatz „allgemein“, dient er Unterscheidung zu dem *Vorgehensmodell des Bundes und der Länder*, welches oft mit dem Zusatz 92 oder 97 versehen ist, jedoch in der Literatur oftmals auch mit V-Modell bezeichnet wird.

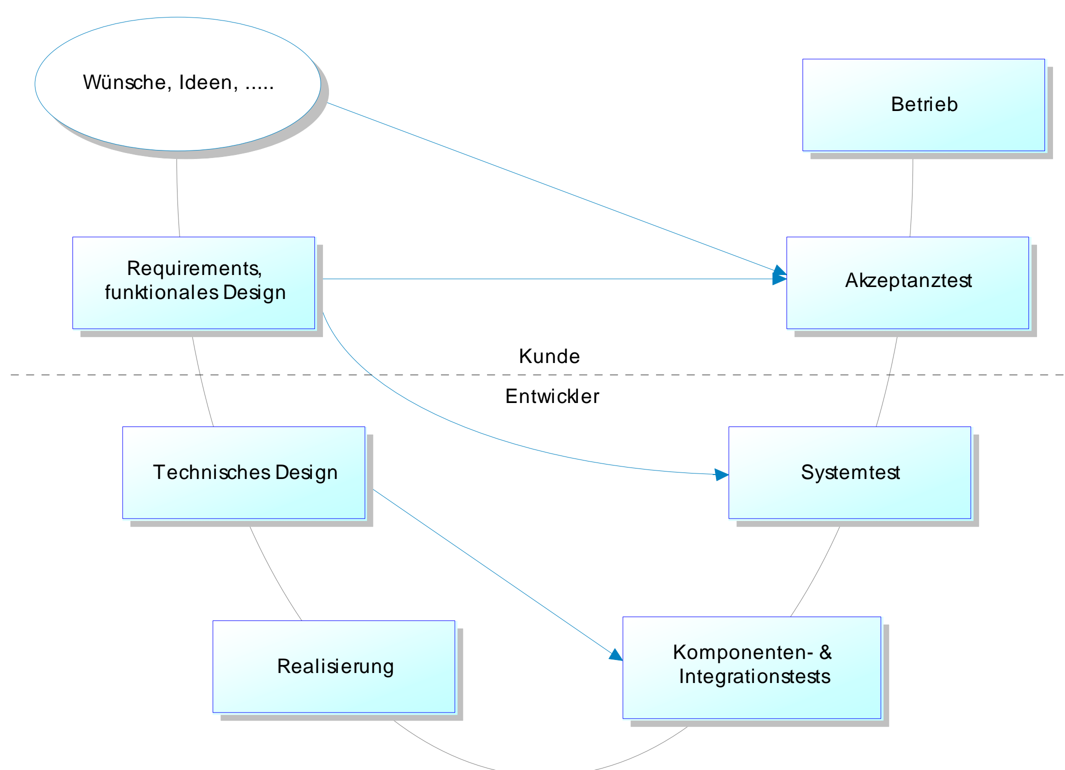


Abbildung 6: Allgemeines V-Modell nach Boehm (1982)

Grundsätzlich lassen sich die Teststufen weiter kategorisieren: in Low-Level- und High-Level-Tests, deren Eigenschaften folgend erläutert werden.

2.4.1 Low-Level-Tests

Zu den Low-Level-Tests zählen die Komponenten- und Integrationstests. Sie unterscheiden sich von den High-Level-Tests durch die ausführenden Personen und die jeweilige Testtechnik. Diese Tests werden typischerweise von den Entwicklern durchgeführt. Bereits nach der Entstehung der ersten Komponenten können erste Komponententests (oder auch Unit- oder Modultests) durchgeführt werden. Maßgeblich für den Umfang des Einsatzes der Komponententests ist die Systemumgebung und Programmiersprache. Der Komponententest zielt auf die Gewährleistung ab, dass die elementarsten Programmteile entsprechend ihrer technischen Spezifikation funktionieren.

Nachdem der Komponententest mit einem zufrieden stellendem Ergebnis abgeschlossen wird, werden gemäß des technischen Konzepts Kombinationen gebildet, um den sogenannten Integrationstest durchzuführen.

Der Test soll Schnittstellen- und Datenflussprobleme aufdecken. Sollten diese zusammengesetzten logischen Blöcke der technischen Spezifikation entsprechen, können sie als einzelnes größeres Modul in weitere umfangreichere Systeme integriert und getestet werden.

Low-Level-Tests erfordern immer eine gute Kenntnis der internen Struktur. Somit werden hier typischerweise White-Box-Techniken angewandt (siehe 2.4.2.2).

Auch wenn die Möglichkeiten dieser Tests limitiert sind sollten diese Tests trotzdem ausführlich durchgeführt werden, da durchgereichte Fehler immer teurer werden. Boehm (1982) belegte dies mit seiner in Abbildung 7 dargestellten *Cost-of-Change*-Kurve. Diese verdeutlicht die Kosten von Änderungen im Verhältnis zu verstrichener Zeit. Die einzelnen Teststufen lassen sich auf diese Kurve auftragen (Abbildung 7). (web: Ambler, 2006)

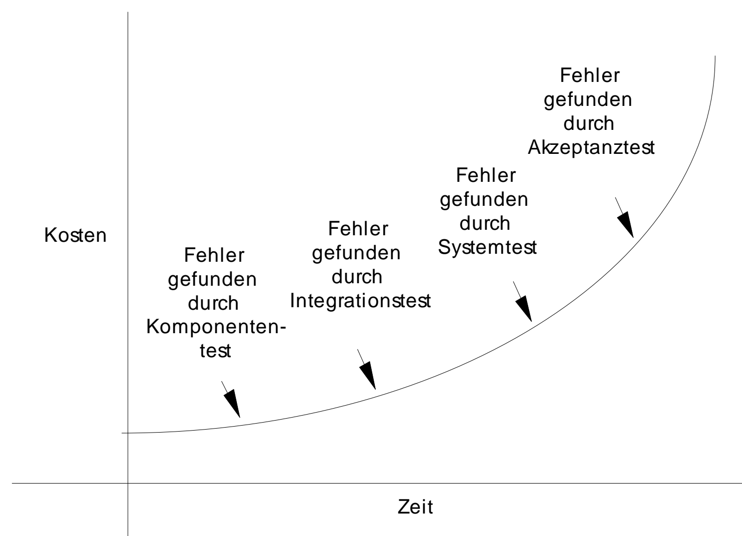


Abbildung 7: Auftragen der Teststufen auf „Cost of Change“-Kurve von Boehm (1982)

Der progressive Kurvenverlauf verdeutlicht, dass die Kosten zur Fehlerbehebung in späten Stadien wesentlich höher ausfallen.

2.4.2 High-Level-Tests

Nachdem die Low-Level-Tests erfolgreich durchgeführt und gefundene Fehler korrigiert wurden, wird vom Entwickler oder dem Testteam typischerweise der Systemtest durchführt.

Obwohl die Komponenten des Systems schon in früheren Teststufen ausgiebig getestet worden sein sollten, ist noch nicht sicher ob das Gesamtsystem die geforderten Qualitätskriterien erfüllt. Ein Systemtest prüft deshalb die Software ob sie den funktionalen und technischen Anforderungen genügt. Die Testfälle leiten sich hier aus dem funktionalen sowie technischen Design und den Anforderungsdokumenten des Kunden oder Anwenders ab.

“Bridges don't fall down because of bad steel, but because of bad architecture.”
Beizer (1984)

Um die vorgegebenen Qualitätskriterien zu testen, kann der Systemtest laut Beizer (1984) funktionale Tests, Stress-, Last-, Performanz- , Konfigurations- , Sicherheitstests, und statische Tests umfassen.. Deshalb werden beim Systemtest hauptsächlich Black-Box-, aber auch White-Box-Techniken eingesetzt.

Der Systemtest stellt die umfangreichste Teststufe dar, da hier alle relevanten Qualitätskriterien getestet werden. Fehler die im Systemtest gefunden werden können oft schwierig auf einzelne Komponenten zurückgeführt werden. Von daher sind die Low-Level-Tests so intensiv wie möglich durchzuführen.

Am Schluss des eigentlichen Entwicklungsprozesses erfolgt der Akzeptanztest beim Kunden.
Der Kunde führt hierbei selbst Tests aus um das das Programm mit den ursprünglichen Anforderungen zu vergleichen. Dies können funktionale oder nicht-funktionale Anforderungen sein. Funktionale Anforderungen beschreiben die Fähigkeiten eines Systems die ein Anwender erwartet, um mit Hilfe des Systems ein fachliches Problem zu lösen. Nicht-funktionale Anforderungen beschreiben Anforderungen an das System die nicht-fachlicher Natur sind, jedoch entscheidend zur Anwendbarkeit des Systems beitragen. Sie definieren beispielsweise Benutzbarkeitsanforderungen, Sicherheitsanforderungen oder Performanzanforderungen. (web: V-Modell XT). Der Systemtest testet ausschließlich aus Anwendersicht, weshalb für Akzeptanztests typischerweise Black-Box-Techniken verwendet werden.

2.5 Testen als Prozess

Die meisten Vorgehensmodelle der Softwareentwicklung sehen für das Testen eine feste Position vor, die nach dem Ende der Implementierung angesiedelt ist. Jedoch sollten sich Testaktivitäten nicht nur auf die in den Vorgehensmodellen geplanten Phasen beschränken. Es ist wichtig, sich schon frühzeitig im Softwareentwicklungszyklus über das Testen und die Testbarkeit des Systems und seiner Bestandteile Gedanken zu machen. Beispielsweise bei der Definition der Anforderungen. Diese sollten bereits frühzeitig in einer Form festgelegt sein, die nachher als Basis für Komponenten- oder Akzeptanztests dienen kann. Das Testen muss ein frühzeitig beginnender Prozess sein, der in jeder Phase organisiert und strukturiert sein sollte. Koomen und Pol (1999) stellen jedoch fest, dass ein Phasenmodell lediglich ein Eckpfeiler eines strukturierten Testprozesses ist. Ingesamt benennen Koomen und Pol (1999) vier Eckpfeiler (Abbildung 8):

- ein Phasenmodell (L) im Zusammenhang mit dem Entwicklungsprozess
- eine organisatorische Einbettung (O)
- die richtige Infrastruktur und Werkzeuge (I)
- sowie Techniken (T) , die Aktivitäten durchführen zu können

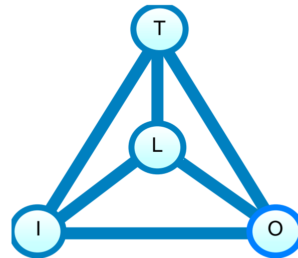


Abbildung 8: Die vier Eckpfeiler des strukturierten Testens nach Koomen und Pol (1999)

Bei jeder Testart und Test von jeder Stufe im Entwicklungsprozess müssen diese vier Punkte beachtet werden. Folgend werden die genauen Inhalte dieser vier Eckpfeiler und welche Rolle im Testprozess einnehmen erläutert.

2.5.1 Phasenmodell

Tests von jeder Stufe sind Prozesse und können in Phasen eingeteilt werden (Tabelle 3). Koomen und Pol (1999) schlagen ein Phasenmodell aus fünf Stufen vor (Abbildung 9).

Planung & Verwaltung	Zeitplanung, administrative Tätigkeiten, Erstellen von Teststrategie, Testplan, Berichten, Dokumentation
Vorbereitung	Zu testende Komponenten, Testfallspezifikationsmethoden, Einrichten der Infrastruktur
Spezifikation	Definition der Testfälle durch Testfallspezifikationsmethoden, Testskripts
Durchführung	Vorbereitungstests, Tests, Prüfung und Auswertung der Ergebnisse, Erstellung von Fehlerberichten
Abschluss	Sichern der Testware, Bewertung des Testobjekts und des Testprozesses, Abschlussbericht

Tabelle 3: Beschreibung der Phasen einer Teststufe

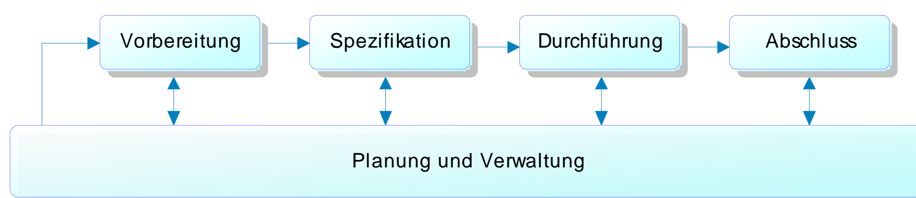


Abbildung 9: Phasenmodell des Testprozesses nach Koomen und Pol (1999)

Es ist stark von dem Unternehmen und dessen Projekten abhängig, wie genau man sich an diesen Vorgaben orientieren sollte. Auch sind in den jeweiligen Teststufen der Anteil der Aktivitäten unterschiedlich hoch. Die von Koomen und Pol aufgeführten Aktivitäten und Phasen bieten jedoch eine Orientierung welche Punkte zu beachten sind. Prinzipiell sollte es jede Phase in jeder Teststufe vorhanden sein, wenn auch teilweise nur mit sehr geringem Umfang.

2.5.2 Organisation

„A testing organisation is the representation of the relationships between test functions, test facilities, and testing activities aimed at performing a structured test“ Thierry (1973)

Ohne eine ausreichende Organisation scheitert fast jeder Testprozess. Die Beteiligung vieler verschiedener Disziplinen, die Unvorhersehbarkeit des Prozesses, die komplexen Steuerungsaktivitäten, fehlende Erfahrung, und der Zeitdruck machen eine Organisation des Testes unverzichtbar (Koomen und Pol, 1999). Dazu gehört insbesondere eine angemessene Rollen- und Aufgabenverteilung.

2.5.3 Infrastruktur und Werkzeuge

Unter Infrastruktur verstehen Koomen und Pol (1999) alle notwendigen Mittel und Einrichtungen, um den Anforderungen entsprechend testen zu können. Eine Unterscheidung kann hier zwischen Testwerkzeugen und Testumgebung getroffen werden.

Typischerweise sollten drei unterschiedliche Testumgebungen zur Verfügung stehen:

- Die Entwicklungsumgebung für Low-Level-Tests.
- Verwaltbare Systemumgebungen für High-Level-Tests.
- Produktionsnahe Umgebungen, hauptsächlich für Abnahmetests.

Testumgebungen müssen früh im Entwicklungsprozess definiert werden. (Koomen und Pol, 1999)

Testwerkzeuge können in jeder Testphase eingesetzt werden. Fundamental sind hier Werkzeuge zur Planung, Verwaltung, und Fehlerverfolgung. Weiterhin ist es möglich, mit Hilfe von Werkzeugen und Automatisierung die Effizienz von Testaktivitäten zu verbessern. Weitere Werkzeuge können auf Controlling und Planungsebene eingesetzt werden. Auf Ebene der Komponententests existieren beispielsweise Werkzeuge zur automatischen Testfallgenerierung und Ausführung sowie zur deren Code-Überdeckung (siehe Kapitel 2.4.2.2). Auch existieren Werkzeuge zur automatischen statischen Analyse des Quellcodes auf Datenflussanomalien oder so genannter *Code-Smells*. Integrationstests können heutzutage

vollautomatisch mit Hilfe von Programmen zur kontinuierlichen Integration[12] durchgeführt werden.

2.5.4 Testtechniken

Unterschiedliche Tests auf unterschiedlichen Teststufen haben verschiedene Testziele und Testobjekte. Für jede Teststufe muss eine Palette von Testtechniken definiert sein, die die Testziele am jeweiligen Testobjekt erreichen können.
Auf den unteren Teststufen, den Low-Level-Tests sind dies vor allem White-Box-Techniken (Kapitel 2.3.2.2). High-Level-Tests können neben den Black-Box-Techniken (Kapitel 2.3.2.1) je nach Anforderungen auch Techniken zur Messung von beispielsweise Last und Performanz benötigt werden.

2.6 Kritische Faktoren

Die Risiken sowie die potentiellen Widerstände mit denen man bei der Einführung von strukturiertem Testen konfrontiert wird werden laut Koomen und Pol (1999) oft unterschätzt. Die neuen Testaktivitäten erfordern für die Entwickler und natürlich auch für die Tester erstmal einen Mehraufwand dessen Vorteile oft nicht besonders transparent sind. Widerstände kann also man erwarten wenn das Bewusstsein für die Wichtigkeit des Testens nicht ausreichen vorhanden ist. Dies muss in Meetings oder Workshops von den Projekt- oder Abteilungsleitern passend kommuniziert werden. Es muss verdeutlicht werden, dass ein gewisser Aufwand der zur Fehlerverminderung oder Früherkennung von Fehlern beiträgt die Wartungs- sowie Fehlerbehebungskosten deutlich reduzieren kann. Zusätzlich muss nach der Veröffentlichung der Veränderungsmaßnahmen genügend Unterstützung und Information vorhanden sein um die Widerstände zu senken.

Widerstände gegen das Softwaretesten können vielfältigen Ursprung haben. Beispielsweise lassen sich die monetären Auswirkungen des Testens nur schwer hinreichend quantifizieren. Testen verringert zudem lediglich das Risiko eines schwerwiegenden Fehlers, jedoch hat die

[12] Kontinuierliche Integration ist ein ursprünglich aus dem Extreme Programming stammendes Prinzip, nachdem der gesamte Sourcecode in kurzen Intervallen compiliert und gebuilded wird. Dadurch soll das Risiko des Zeitverzugs durch Integrationsprobleme stark gesenkt werden.

Software potentiell schon vorher keinen oder nach dem Testen immer noch einen Fehler dieser Art. Testkosten können höher als die Behebungs- und Ausfallkosten ausfallen. Ähnlich schwer ist die Messung inwiefern eine durch Testen oder *Coding Standards* erreichte höhere Softwarequalität Auswirkungen auf die Wartung, Erweiterung oder Fehlerbehebung hat. Jede dieser Herausforderungen kann Widerstände verursachen und die Einführung eines Testprozesses gefährden.

Dieser Umgang mit den Widerständen ist ein wichtiger zu beachtender Punkt bei der Durchführung des Veränderungsprozesses. Unterstützend ist es wichtig sich über folgende Punkte Gedanken zu machen (Komen und Pol, 1999):

Notwendigkeit

Vorbedingung für eine erfolgreiche Einführung ist das Bewusstsein, dass eine Änderung nötig ist um die Qualität von Software und Testen verbessern zu können. Zudem muss sich das Unternehmen die Frage stellen, welche positiven Effekte die Erhöhung der Qualität für das Unternehmen, die Abteilungen, und die Mitarbeiter haben kann.

Klare Zieldefinitionen

Der zu erreichende Zustand muss klar definiert sein. Es sollte für jeden Beteiligten sichtbar sein, wo rauf die Änderungen abzielen. Dies kann je nach Gruppe der Beteiligten unterschiedlich sein. Für das Management bei BAT muss klar sein, dass die Kosten für Wartung und Fehlerbehebung gesenkt werden sollen. Für die Tester muss ersichtlich werden, dass mit strukturiertem Testen schneller und umfangreicher getestet werden kann und dass die Aufgaben klarer definiert sind, und für die Entwickler muss klar werden, dass die Einführung von den oben genannten Werkzeugen bessere Softwarequalität und weniger Fehler bedeutet weshalb die Rate von *Change*- und *Errorrequests* gesenkt werden kann und der Kunde zufriedener wird. Es muss zudem verdeutlicht werden, dass Tests nicht die Leistung des Entwicklers schmälern soll, sondern ein unterstützendes Werkzeug darstellt welches ihm helfen kann die Qualität des gelieferten Produkts zu steigern.

Unterstützung des Managements

Das Management muss der organisatorischen Veränderung des bestehenden Arbeitsablaufs zustimmen, sprich ausreichend Budget, Zeit, und Personal zur Verfügung stellen. Es muss

jedem Beteiligten deutlich werden, dass das Management hinter der Einführung des Testprozesses steht und diesen ausdrücklich unterstützt.

2.7 Interviews

Mündliche Befragungen von in das zu untersuchende Gebiet eingebundenen Personen sind ein gutes Mittel zur Informationsbeschaffung. Im Gegensatz zu einem Fragebogen lassen sich zusätzlich Informationen aus gezieltem Nachfragen und der spontanen Ausdrucksweise, Stimmlage und Gesichtsausdruck des Interviewten gewinnen. Man sollte jedoch beachten dass persönliche Interviews einen zeitlichen höheren Aufwand bedeuten als schriftliche Befragungen. Folgend werden einige Interviewarten, -strukturen und Fragetechniken erläutert.

2.7.1 Interviewstruktur

Es gibt betreffend des Ablaufs der Fragestellungen und des Interviews 3 Kategorien von Interviews (Schnell, 2005):

Nicht standardisiert
Oder auch „offene“ Interviews. Sie laufen ähnlich ab wie normale Gespräche zu denen ein bestimmtes Thema vorgegeben ist. Der Interviewer benutzt meist nu einen Stichpunktartigen Leitfaden der zu besprechenden Fragestellungen.

Halb standardisiert
Eine etwas strukturiertere Interviewform mit flexiblem Fragekatalog und einer freien Reihenfolge. Auswahl der Fragen und der Reihenfolge richtet sich nach dem Gesprächsverlauf.

Standardisiert
Standardisierte Interviews folgen einer festen Reihenfolge von Fragen mit gleichem Wortlaut für jeden Gesprächspartner. Sie werden normalerweise in Verbindung mit kategorialen (Ja oder Nein) oder skalierten Antwortmöglichkeiten (z.B. 1=Gut bis 5=Schlecht) verwendet. Standardisierte Interviewtechniken werden oft zur quantitativen Analyse eines Sachverhaltes verwendet.

2.7.2 Fragentechniken

Es gibt 2 grundsätzliche Einteilungen von Fragenstellungen (Schnell, 2005):

Offene Fragen

Hierbei wird dem Gegenüber eine freie Assoziation innerhalb seiner Antwort ermöglicht. Der Gesprächspartner soll sich inhaltlich beteiligen damit der Interviewer möglichst viele Informationen gewinnen kann. Beispiel: *„Wie sieht der Testprozess innerhalb ihrer Abteilung aus?“*. Vor allem wichtig, wenn man Informationen zu einem wenig bekannten Sachverhalt ermitteln will.

Geschlossene Fragen

Bei geschlossenen Fragen ist die Antwort auf unterschiedliche Arten begrenzt, entweder durch eine Alternative (z.B. ja oder nein) oder durch eine erwartete Mengenangabe. Beispiel: *„Sind Sie am Testen direkt beteiligt?“*, *„Wie viele Iterationen werden normalerweise benötigt?“*. Sie sind vor allem für eine quantitative bzw. stochastische Auswertung interessant.

2.7.3 Interviewarten

Interviews können auf verschiedene Arten geführt werden. Es macht Sinn sich vorher darüber Gedanken zu machen, was man mit dem Interview erreichen will und entsprechend die passende Interviewart auszuwählen. Bei einer Befragung einer einzelnen Person zur Informationsgewinnung zu einem bestimmten Thema bieten sich 2 unterschiedliche Vorgehensweisen an (Schnell, 2005):

Narratives Interview

Hier wird dem Befragten größtmögliche Freiheit gewährt und der Verlauf des Interviews ist völlig offen. Der Befragte „erzählt“ was ihm zu einer Thematik einfällt. Das narrative Interview eignet sich jedoch nicht um vorher aufgestellt Hypothesen zu prüfen. Die Hypothesen werden erst vom Gesprächspartner im Verlauf des Gesprächs formuliert. Bei narrativen Interviews wird immer unstandardisiert vorgegangen.

Leitfadeninterview

Beim Leitfadeninterview wird vorher ein Fragenkatalog erstellt jedoch kann dem Befragten bei der Beantwortung der Fragen viel Spielraum eingeräumt werden. Er hat die Möglichkeit Fragen unter Umständen zu kommentieren, darf frei berichten und die Ausrichtung des Interviews beeinflussen. Der Interviewer hat jedoch in jedem Fall sicherzustellen dass der Interviewte nicht zu sehr abschweift und man sich an den vorher erstellten Leitfaden hält. Ein Leitfadeninterview ist also halb- oder komplett standardisiert.

2.7.4 Auswahl der Interviewtechnik

Für die Durchführung der Interviews wird ein Leitfadeninterview mit halb standardisiertem Vorgehen und offenen sowie geschlossenen Fragen verwendet. Es handelt sich hier um eine qualitative Analyse der Testsituation, und es soll flexibel auf den Verlauf des Interviews eingegangen werden können. Dies dient dazu, ein besonders detailliertes und aussagekräftiges Bild der vorhandenen Situation zu erhalten.

In vorherigen Meetings der Abteilungen wurde das Vorgehen bei der Analyse des aktuellen Zustandes vorgestellt. Es wurden die durchzuführenden Interviews angekündigt und versucht die Wichtigkeit des Testens als Qualitätssicherungsmaßnahme den Mitarbeitern ins Bewusstsein zu rufen.

Kapitel 3

Analyse und Auswertung

Die vorzuschlagenden Verbesserungen des Testprozesses orientieren sich vor allem an der vorhandenen Situation bei *GSD-CS*. Somit ist eine vorausgehende gründliche Analyse zentraler Bestandteil des Vorgehens. Die Basis dieser Analyse bilden Informationen die aus Interviews gewonnen wurden. Auf deren Basis sowie den Informationen die sich durch die Arbeit im Unternehmen ergeben haben, werden die Abteilung, ihre Projekttypen und ihr Vorgehen bei der Softwareentwicklung beschrieben.

Mit Hilfe dieser Bestandsanalyse werden am Ende des dritten Kapitels identifizierte Probleme aufgezeigt und konzeptionelle Lösungsvorschläge gemacht. Diese Lösungsvorschläge werden im Kapitel 4 vertieft und darauf basierend ein Konzept entworfen.

3.3 Aufgaben der Abteilungen

Wie in den Grundlagen beschrieben wurde die vorliegende Arbeit in Hamburg für die Abteilung *Group Service Delivery – Customer Services (GSD-CS)* erstellt. *GSD-CS* besteht aus 3 Abteilungen: *Applications (GSD-CS-A)*, *Technologies (GSD-CS-T)*, und *Solutions (GSD-CS-S)*. Die Arbeit betrachtet vor allem die Situation in den Abteilungen *Applications* und *Solutions*. Nach Absprache mit dem Management soll der Testprozess und die Testmethoden für diese beiden Gruppen konzipiert werden.

Die Abteilung *Solutions* umfasst etwa 15 interne und externe Mitarbeiter. Sie befasst sich zum einen mit infrastrukturellen und technologischen Anforderungen, d.h. dem Erstellen von Vorlagen wie *Masterpages* in *ASP.NET* und dem Programmieren von Werkzeugen, die die Arbeit der anderen Mitarbeiter erleichtern und oft erst ermöglichen. Zum anderen beschäftigt sie sich mit der Anwendungsentwicklung von Applikationen, die in den verschiedenen

Endmärkten zur systematischen Analyse, Sammlung und Darstellung von Unternehmensdaten verwendet werden können. Dies dient dazu, Mitbewerber oder Marktentwicklungen im Hinblick auf ein gewünschtes Ziel beurteilen zu können, zum Beispiel durch automatisiertes Berichtswesen aus dem existierenden Data-Warehouse. Zusammenfassend lassen sich die Aufgaben der Abteilung mit der Bereitstellung von Business Intelligence Verfahren beschreiben. Aufgrund der unterschiedlichen Anforderungen und Aufgaben wird keine einheitliche Programmiersprache oder ein einheitliches Framework benutzt. In der Abteilung *Solutions* werden die meisten Werkzeuge für die anderen Abteilungen mit C# unter Verwendung von ASP.NET und Windows Forms entwickelt. Jedoch werden auch Programmiersprachen wie MDX und PL/SQL eingesetzt.

Die Abteilung *Applications* umfasst etwa neun interne und externe Mitarbeiter. Sie befasst sich mit dem Testen und dem Second-Level-Support sowie Anwenderschulungen über die im Bereich Business Intelligence eingesetzten Applikationen. Dies umfasst die von *GSD-CS-S* entwickelten Lösungen sowie eingekaufte Fremdsoftware wie der *Reporting*-Applikation *IBM Congos TM1* und der Business Intelligence Plattform *Business Objects XI*.

In den folgenden Abschnitten wird das übliche Vorgehen von dem Festhalten der Anforderungen bis zur Anwendungsentwicklung und den Tests geschildert.

3.3.1 Anforderungen

Anforderungen für die Entwicklung von neuer und der Erweiterung vorhandener Software werden im Allgemeinen nach einer informalen Anfrage und darauffolgender Aufwandsschätzung über einen *Call* im *Trouble-Ticket-System* (TTS) festgehalten. Es existiert jedoch kein standardisiertes Anforderungsdokument. Um Entwicklungen oder Erweiterungen anzustossen werden meist weitere Anforderungen oder Fragen mündlich bzw. per informaler Email geklärt.

3.3.2 Entwicklungszyklus

GSD-CS in Hamburg verfügt über kein definiertes Entwicklungsmodell. Aufgrund des *Trouble-Ticket-Systems* und der Anweisung des Managements an die Mitarbeiter, dass Aufgaben, wie Neuprogrammierung, Fehlerbehebung oder Erweiterung nur noch mit Bezug

auf ein offenes Ticket und zugewiesener Budgetierung zu erledigen sind, hat sich eine gewisse Reihenfolge der Aktivitäten herausgebildet (Abbildung 10).

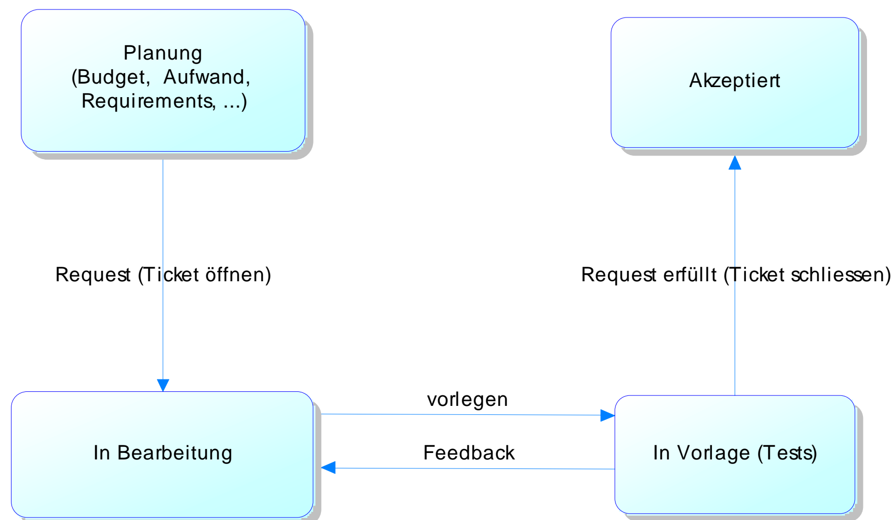


Abbildung 10: Entwicklungsprozess bei *GSD-CS*

Nach einer informalen Anfrage an die Abteilung und dem Äußern von Ideen und Wünschen werden der Aufwand des Projektes, also Personentage, Kosten und Zeitrahmen grob geschätzt. Bei Einverständnis des Kunden wird ein Ticket geöffnet, welches dann von dem entsprechenden Programmierer oder Teamleiter angenommen und bearbeitet wird. Vorläufige Versionen werden den Testern oder direkt dem Kunden vorgelegt und anhand des Feedbacks werden unter Umständen weitere Änderungen vorgenommen. Sobald der Tester bzw. der Kunde alle seine Tests auf dem Programm mit positivem Ergebnis durchführen konnte, wird das Programm für die Produktivumgebung freigegeben und dort veröffentlicht. Das Ticket wird geschlossen (Abbildung 10). Getestet wird hier auf die funktionalen und nicht-funktionalen Anforderungen des Kunden, welche im ersten Schritt informal festgelegt wurden.

Auch wenn es keine exakte Definition der Entwicklungsphasen gibt, allerdings die Phasen sequentiell ausgeführt werden, ist das Vorgehensmodell bei *GSD-CS* wohl am treffendsten mit dem *erweiterten Wasserfallmodell* (Royce, 1970) zu beschreiben. Trotz der sequentiellen Abfolge der Entwicklungsphasen ist ein Rücksprung zur vorherigen Phase zur Verfeinerung des Entwurfs möglich.

Boehm (1982) erweiterte das Wasserfallmodell um elementare Aufgaben der Qualitätssicherung, woraus das allgemeine V-Modell entstanden ist. Wenn man nun die bei GSD-CS eingesetzten Testaktivitäten in den Entwicklungsprozess mit einbezieht und auf diese Modell anwendet, kommt man zu dem in Abbildung 11 dargestellten Ergebnis:

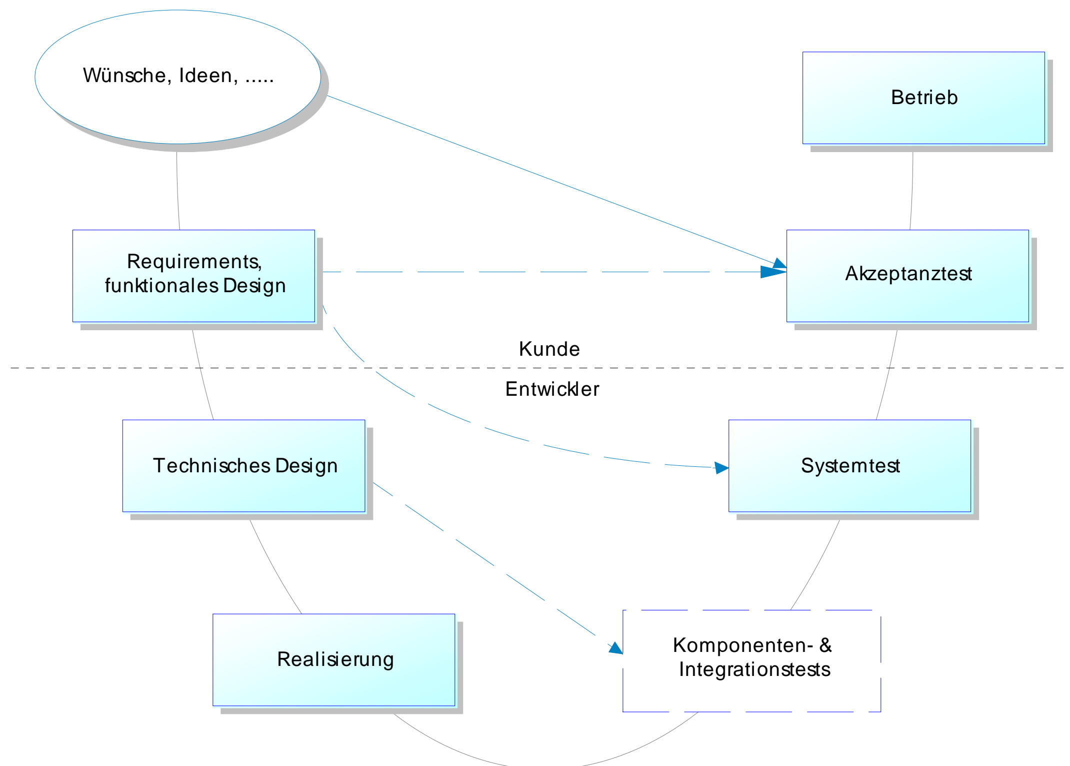


Abbildung 11: Vorgehen mit V-Modell bei GSD-CS

Die gestrichelten Linien drücken aus, dass diese Aktivitäten in einem regulären *V-Modell* vorgesehen sind, jedoch bei *GSD-CS* nicht ausgeführt werden. Gemäß Abbildung 11 werden somit Komponenten- und Integrationstests sowie die systematische Ableitung der Testfälle aus dem technischen und fachlichen Design nicht angewandt.

3.4 Testen

Aufgrund der unterschiedlichen Ausrichtung der Abteilungen *GSD-CS-S* (entwicklungs-orientiert) und *GSD-CS-A* (kundenorientiert) müssen demzufolge Unterscheidungen zwischen den durchzuführenden Tests getroffen werden.

3.4.1 Testmöglichkeiten

Wie bereits in Kapitel 3.3 erläutert, ist *GSD-CS-S* vornehmlich auf die Anwendungsentwicklung ausgerichtet. So hat man dort die Möglichkeit, den entwickelten Quellcode zu testen und auf Qualität zu überprüfen, beispielsweise mit White-Box-Tests (siehe Kapitel 2.3.2.2). Getestet werden kann und sollte somit auf Komponenten- und Integrationsebene. *GSD-CS-A* ist als kundenorientierte Abteilung überwiegend auf das Testen aus Anwendersicht fokussiert. Die Analyse hat ergeben, dass vor allem die Aspekte Benutzbarkeit und Korrektheit wichtig sind. Für das Testen einer GUI-Anwendung unter diesen Qualitätsmerkmalen muss kein Wissen über die technische Repräsentation des Systems vorhanden sein, welches in der Abteilung GSD-CS-A auch nur begrenzt vorhanden ist. Für Testaufgaben dieser Art wird jedoch Wissen über die Endanwendung aus Nutzersicht benötigt, über welches die Mitarbeiter der Abteilung jedoch verfügen. Somit ist dort das Black-Box-Testen die primär eingesetzte Methodik (siehe Kapitel 2.3.2.1). Testaufgaben sind typischerweise System- und Abnahmetest.

3.4.2 Vorgehen

Momentan werden Abnahme- und Systemtests als einzige Testaktivität durchgeführt. Dies geschieht vor der Auslieferung der Software an den Kunden entweder durch das Programmierteam oder durch Mitarbeiter der Abteilung *GSD-CS-A*. Aus den Mitarbeiterinterviews ging hervor, dass meistens „ad-hoc“ und ohne definiertes Vorgehen getestet wird. Testfälle entstehen nicht strukturiert aus dem Anforderungsdokument oder dem technischen Entwurf, sondern ebenfalls spontan. Eine Kategorisierung der Anforderungen in funktionale oder nicht-funktionale Anforderungen wie Benutzbarkeit, Sicherheit oder Korrektheit erfolgt selten. Es gibt wenig Information über die Anforderungsüberdeckung des Systems und dem Anforderungsdokument. Somit ist ein *Requirements Tracing*[13] hier nicht vorhanden. Testwerkzeuge für das Testen von GUI-Anwendungen wie *Capture and Replay*-Werkzeuge[14] werden nicht verwendet. Die Vorgehensweise beim Testen verfolgt hier einen eher pragmatischen Ansatz. In der Literatur wird dies auch mit den Begriffen „Exploratives

[13] Requirements Tracing ist die Möglichkeit den Status eines Requirements im kompletten Systemlebenszyklus zu verfolgen und dokumentieren.

[14] Capture and Replay Werkzeuge automatisieren Testabläufe indem sie mit der grafischen Benutzeroberfläche der zu testenden Anwendung interagieren. Anwendereingaben werden hierbei simuliert.

Testen“ oder „Error Guessing“ d.h. „Raten von Fehlern“ beschrieben (Spillner und Linz, 2005) . Es ist also eine nicht formale Black-Box-Technik.

Sollten Fehler aufgedeckt werden, kommt es darauf an, ob das Programm bereits die Testphase verlassen hat und zur produktiven Arbeit freigegeben wurde. Falls nicht, erstellt der Tester eine informale Nachricht an den Entwickler, welcher die zu testende Komponente übergeben hat und bittet um Nachbesserung. Falls das System bereits produktiv eingesetzt wird, muss ein neues Ticket im *Trouble Ticket System* erstellt werden, da Änderungen an Produktivsystemen nur vorgenommen werden dürfen wenn ein passendes Ticket dafür vorliegt. Die Mittel für eine konstruktive Qualitätssicherung des Testprozesse, beispielsweise durch Testprotokolle oder Vorlagen, sind nur in geringem Umfang vorhanden. Die Systemkonfiguration fließt meistens nicht in die Tests ein.

3.5 Identifikation von Problemen

Die Interviews und die darauf basierende Analyse der aktuellen Situation zeigen, dass das Testen bei GSD-CS einige Unzulänglichkeiten aufweist. Im Folgenden wird der Testprozess unter dem Aspekt der vier Eckpfeiler des strukturierten Testens (Kapitel 2.5, Testen als Prozess) analysiert um Probleme zu identifizieren.

3.5.1 Integration in das Vorgehensmodell

Die Testaktivitäten bei GSD-CS sind nicht in das Vorgehensmodell integriert. Dies ist damit zu begründen, dass das Vorgehensmodell lediglich implizit definiert ist und es somit keine Teststufen gibt. Das Testen findet als monolithischer Block am Ende des Entwicklungsprozesses statt. Durch die fehlende Integration von Testaktivitäten in das Vorgehensmodell fehlen wichtige Teststufen, wie Low-Level-Tests bei GSD-CS. Ein Weglassen dieser Teststufen erhöht den Aufwand im Systemtest durch das fachlich orientierte Personal ungemein (Kapitel 2.3).

Zudem haben viele Personen, die aktuell Tests durchführen, keine ausreichende Kenntnis über die technische Repräsentation des Systems, um aus den auftretenden Fehlern schnell und effizient Rückschlüsse auf die betroffenen Komponenten ziehen zu können. Durch reines

High-Level-Testen entsteht ein übermäßiger Aufwand an Testaktivitäten sowie eine erschwerte Fehlersuche und erhöhter Kommunikationsaufwand zwischen den Abteilungen.

3.5.2 Technik

Bisher wird lediglich der System- und Abnahmetest mittels „explorativem Testen" durchgeführt. Dadurch ist nicht klar, welchen Anteil des Codes und der Anforderungen die Testfälle abdecken und wie detailliert und unter welchen Kriterien getestet wird. Somit ist das Testen nicht transparent und verlässlich. Die Tester bei *GSD-CS-A* haben oft eine hohe Erfahrung im Umgang mit den Applikationen, da sie teilweise direkt aus dem Fachbereich kommen. Jedoch ist selbst ein Tester, der sowohl im Test als auch in der Anwendung der Applikation sehr erfahren ist, keine Sicherheit für eine hohe Fehlerfindungsrate.

Zudem wird wenig dokumentiert. Durch eine umfassendere Dokumentation der Testfälle und des Testablaufs kann im Nachhinein sichergestellt werden, dass die Tests ausreichend umfangreich durchgeführt wurden. Durch ausreichende Dokumentation kann außerdem gewährleistet werden, dass genügend Zeit und Kapazitäten für Änderungen und Fehlerbehebung vorhanden ist. Ein ausführliches Protokoll bietet einen ersten Eindruck von der Qualität des Produkts. (Pol und Koomen, 1999)

Des Weiteren steht das informale Anforderungsdokument nur in eine geringen Zusammenhang mit der Testfallerstellung. Eine systematische Ableitung der Testfälle aus dem fachlichen Design wird nicht durchgeführt. Dies liegt zum Teil in der Informalität der Anforderungsdefinition begründet. Die Anforderungen werden nicht nach Qualitätskriterien kategorisiert, was das Testen dieser Kriterien erschwert.

3.5.3 Infrastruktur und Werkzeuge

Die infrastrukturellen Anforderungen an Testservern und passenden Testdaten werden bei GSD-CS erfüllt. Eine passende Büroumgebung ist auch gegeben, da jeder Mitarbeiter in der Regel einen eigenen Arbeitsplatz hat.

Prozessbegleitend wird ein *Trouble Ticket System* eingesetzt. Für gefundene Fehler oder gewünschte Erweiterungen und Neuentwicklungen können dort unter anderem Tickets

geöffnet und einem Entwickler zugewiesen werden. Es erfüllt somit die grundsätzliche Funktionalität eines *Bug Tracking Systems*. Somit sind die Grundvoraussetzungen an prozessbegleitenden Werkzeugen erfüllt.

Im Falle einer Einführung von Low-Level-Teststufen müssen jedoch weitere unterstützende Werkzeuge vorhanden sein. Diese Tests sind oft automatisierbar oder mit Hilfe von geeigneten Werkzeugen wesentlich effektiver, da das Testobjekt (meistens der Quellcode) in der Regel eine formale Struktur aufweist.

3.5.4 Phasenmodell

Die vorhandene Teststufe ist bisher nicht in einem Phasenmodell definiert. Planung, Ausführung und Bewertung finden spontan statt. Dies hat zur Folge, dass die Testphase unkontrolliert und somit nicht steuerbar ist. Der Tester muss anhand seiner Erfahrung selbst überlegen, welche Planungsaktivitäten er wann und in welchem Umfang anstellt. Auch das Ausmaß der Ausführung ist ihm selbst überlassen. Der Test ist abgeschlossen sobald der Tester dies anhand von Erfahrungswerten beurteilt.

Dieses Vorgehen hat ähnliche Auswirkungen wie die fehlende Auswahl der Testtechniken. Transparente Qualität des Softwareprodukts kann es nur mit definierten und transparenten Testaktivitäten geben. Dazu gehört unter anderem eine strukturierte Planung.

3.6 Verbesserungsvorschläge

Für die im vorherigen Abschnitt identifizierten Probleme sollen nun in erster Instanz generelle Verbesserungsvorschläge gemacht werden. Diese Vorschläge sind zunächst überwiegend theoretischer Art und werden anschließend im Konzept (Kapitel 4) konkretisiert.

3.6.1 Integration in das Vorgehensmodell

Wie die Analyse des Vorgehens bei GSD-CS ergab, findet die Entwicklung unter einem inkrementellen Entwicklungsprozess statt. Das Testen im allgemeinen V-Modell (siehe Kapitel 2.3, Abbildung 6; Boehm, 1982) kann hier also als Referenzvorgehen betrachtet

werden. Das V-Modell ist abstrakt genug um den Prozess bei GSD-CS abzubilden ohne Veränderungen an diesem vornehmen zu müssen.

Das V-Modell sieht insgesamt vier Teststufen vor:

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

GSD-CS sieht bisher nur zwei dieser vier Stufen in ihrem Entwicklungsmodell vor, und zwar die High-Level-Tests Systemtest und Abnahmetest. Wie in Kapitel 3.5.1 geschildert, führt dies zu einem zu hohen Testaufwand, der mit der Einführung von Low-Level-Tests gesenkt werden könnte.

Mit Komponententests können zwar nur Codier- und Logikfehler gefunden werden, jedoch werden durchgereichte Fehler immer teurer (Kapitel 2.4.1: Low-Level-Tests, Abbildung 7; Boehm, 1982). Zudem können mit Komponententests Datenflussanomalien gefunden werden und Unklarheiten in den Requirements aufgedeckt werden. Dadurch wird sichergestellt, dass die Komponente gemäß ihrer Spezifikation funktioniert.

Die Integration von neuen oder die Änderung alter Komponenten kann oft zu Problemen führen. Somit sind regelmäßige Integrationstests, nicht nur im Entwicklungszyklus ein wichtiger Schritt zur Risikominimierung ungewollter Verzögerungen. Da der Integrationsaufwand stark mit dem Integrationsintervall steigt, senkt man mit kurzen Intervallen die Risiken einer unerwarteten Verzögerung.

Die Einführung der Teststufen Komponenten- und Integrationstest sowie die Verbesserung des Systemtests benötigen jeweils spezielle Techniken, Werkzeuge und Infrastrukturen die in den folgenden Abschnitten erläutert werden.

3.6.2 Technik

In der vorliegenden Arbeit wurde erkannt, dass die Techniken die bei GSD-CS eingesetzt werden nicht definiert sind. Somit ist nicht klar, welche Person und welche Technik in der

Praxis einsetzt wird. Weiterhin wurde festgestellt, dass dies in vielen Fällen der Grund für unzureichendes Testen ist, was sich in einer zu hohen Fehlerrate im Programm nach Freigabe zur Produktion äußert. Somit besteht die Notwendigkeit, dass GSD-CS für jede Teststufe passende Techniken vorgibt, mit denen die Anwendung effizient und nachvollziehbar untersucht werden kann. Dadurch erhöht sich gleichzeitig die Transparenz des Testens wodurch es einfacher ist die Qualität des Softwareprodukts zu beurteilen. Zur Umsetzung dieser Techniken können diverse Werkzeuge und eine passende Infrastruktur wichtig sein. Abschnitt 3.6.3 soll dies genauer beschreiben.

Für die Low-Level-Teststufen ist die Anwendung passender White-Box-Testmethoden besonders wichtig, um den Tests eine ausreichende Aussagekraft zu verleihen. Die Qualität des Testens steht und fällt mit der Qualität der Testfälle. Doch nicht nur die Einführung neuer Techniken für die vorgeschlagenen neuen Teststufen, sondern auch die Verbesserung der aktuellen Tests sind von Bedeutung. Eine Verbesserung zu simplen zufälligen Tests mit „Error Guessing“ stellt die Identifikation von möglichen Schwachstellen dar. Weitere Black-Box-Techniken für den Systemtest sind ergänzend empfohlen. In Kapitel 4 wird darauf näher eingegangen.

3.6.3 Infrastruktur und Werkzeuge

In Kapitel 3.6.1 wurde vorgeschlagen zusätzliche Testphasen in den Entwicklungsprozess zu integrieren. Die beiden Low-Level-Teststufen Komponententest und Integrationstest stehen dabei aktuell im Fokus.

Für Komponententests benötigt man zunächst ein Framework für die eingesetzte Programmierumgebung, welches deren Implementierung und Ausführung unterstützt. Wie in 2.3.2.2 erwähnt ist es sinnvoll, zu diesem White-Box-Testen zu messen welche Teile der Komponente durch den Test abgedeckt werden. Dafür existieren verschiedene Hinlänglichkeitskriterien, die mit Hilfe von weiteren Werkzeugen begleitend zum Komponententest angewendet werden können.

Als weiteren Komponententest kann eine werkzeuggestützte statische Analyse durchgeführt werden. Die in Kapitel 2.4.3.3 (Werkzeuggestützte statische Codeanalyse) genannten Vorteile können entscheidend zu Fehlerminimierung und Qualitätsverbesserung der Software

beitragen. Eine Evaluierung in wie weit ein konkretes Werkzeug für GSD-CS von Nutzen sein kann wird im Laufe von Kapitel 4 durchgeführt.

Vorrausetzung für einen nützlichen Integrationstest bei GSD-CS ist ein geringer Aufwand. Oft empfiehlt sich Automation als Mittel zur Aufwandsreduktion wenn ein nicht sonderlich spezifisches Ergebnis gefordert ist oder man dieselben technischen Gegebenheiten vorfindet. Somit sollte eine potentielle Integrationsumgebung eine Automatisierung des Integrationsprozesses unterstützen.

Weitere infrastrukturelle Änderungen oder Werkzeuge sind für den Systemtest vorerst nicht von Nöten. Durch die zu testende heterogene Anwendungslandschaft, ist der Werkzeugeinsatz wie z.B. von Automatisierungswerkzeugen nur schwer effektiv möglich. Wichtige prozessbegleitende Werkzeuge wie ein *Trouble Ticket System* sind soweit vorhanden. Produktionsnahe Systemumgebungen für die Tests sind ebenfalls verfügbar.

3.7.4 Phasenmodell

Die Organisation der Teststufen in einem Phasenmodell hilft unterstützend dabei die zahlreichen Aktivitäten des Testens strukturiert abzuarbeiten. Der jeweilige Umfang der Phasen der unterschiedlichen Teststufen kann jedoch variieren. Low-Level-Tests erfordern andere Planungsaktivitäten als Systemtests. Selbst in einer Teststufe variieren die Phasen unter den einzelnen Testtechniken. Beispielsweise unterscheidet sich die Vorbereitung für einen Lasttest von dem eines Benutzbarkeitstests. Für jede Teststufe und Testart müssen die jeweiligen Phasen sowie deren Aktivitäten passend definiert sein.

Kapitel 4

Konzept

Das Konzept zur Einführung eines strukturierten Testprozesses bei GSD-CS ist unterteilt in 2 Abschnitte. Im ersten Abschnitt werden anhand der in Kapitel 3 geschilderten Analyse Methoden und Vorgehensweisen praxisnah beschrieben, die die identifizierten Mängel beheben. Dabei liegt die Herausforderung darin, die Maßnahmen auf der einen Seite so konkret zu gestalten, dass der Anwender in der täglichen Arbeit exakte Richtlinien für die Umsetzung der Maßnahmen erhält. Auf der anderen Seite darf sie nicht zu projektspezifisch werden um dem Zusatz im Titel der Arbeit („…für heterogene Projektumgebungen“), die bei GSD-CS auch existieren, gerecht zu werden. Zwischen Anwendbarkeit, Nutzen, und Widerverwendbarkeit muss die richtige Balance gefunden werden. Erklärtes Ziel dieser Beschreibung ist es, den Mitarbeitern eine Hilfestellung zu bieten, damit die vorgeschlagenen Änderungen im Testprozess umgesetzt werden und die Vorteile der Änderungen auch erkennbar sind. Der 2. Abschnitt des Kapitels 4 bildet das Umsetzen dieser identifizierten Maßnahmen in ein konzeptionelles Testhandbuch. Dies soll die Nachhaltigkeit dieser Maßnahmen unterstützen, und eine Grundlage bieten, gemachte Erfahrungen beim Testen und in der Entwicklung zu dokumentieren und weiterzugeben. Das Testhandbuch ist eine Konkretisierung der Testpolitik über alle Projekte, Teststufen, und Testphasen hinweg. Es soll die Risiken und Chancen für die Qualität adressieren, die durch das Testen reduziert werden können und sollen.

Im Idealfall kann der Testleiter das Testhandbuch für die Auswahl der Testaktivitäten für die jeweilige Testphase heranziehen. Eine genauere Beschreibung der Ziele und Inhalte des Testhandbuchs finden sich in im zweiten Hauptabschnitt (Kapitel 4.2: Testhandbuch) dieses Kapitels.

4.1 Maßnahmen

In Kapitel 3 wurden Verbesserungspotentiale identifiziert und konzeptionelle Lösungsmöglichkeiten aufgeführt. Nun ist der nächste Schritt diese Maßnahmen in Hinsicht auf die bei GSD-CS spezifischen Gegebenheiten und Anforderungen zu evaluieren.

Der Autor empfiehlt in den nächsten Abschnitten weitere Testphasen, vor allem für das Low-Level-Testen sowie Verbesserungsmöglichkeiten für die High-Level-Tests. Außerdem werden Werkzeuge evaluiert die diese Phasen unterstützen und Testtechniken vorgeschlagen, die ein effektiveres Testen ermöglichen.

4.1.1 Integration in das Vorgehensmodell

Testaktivitäten sollten grundsätzlich in das Vorgehensmodell integriert sein und nicht erst, wie derzeit bei GSD-CS der Fall ist, am Ende des Entwicklungsprozesses als monolithischer Block stehen. Testaktivitäten werden fast ausschließlich von der fachbereichsnahen Abteilung GSD-CS-A ausgeführt was zu einem zu umfangreichen Systemtest sowie erschwerter Fehlerfindung und hohem Kommunikationsaufwand führt. Im Folgenden werden Teststufen (Abbildung 12) vorgestellt, die auf einfache Weise in das bestehende Vorgehen bei GSD-CS integriert werden können.

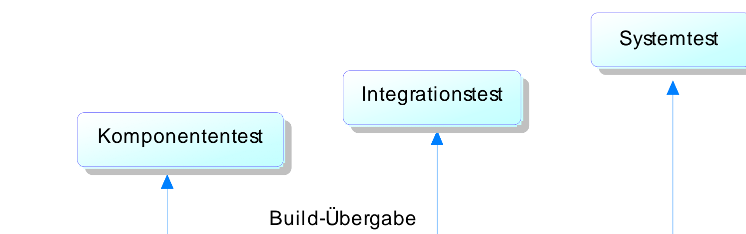


Abbildung 12: Teststufen

4.1.1.1 Komponententest

Das Testen der einzelnen Komponenten durch den Entwickler soll bei GSD-CS den Aufwand des Systemtests reduzieren. Zur Bewertung des Umfangs der durchgeführten Komponententests sollen Hinlänglichkeitskriterien herangezogen werden, welche belegen welcher Anteil der Software von den Testfällen abgedeckt wurde. Komponententests werden typischerweise mit Werkzeugunterstützung ausgeführt. Da die Messung der Überdeckung

ebenfalls vollautomatisch durchgeführt werden kann, ist auch hier der Einsatz von Werkzeugen zu empfehlen.

Die Qualität und Verständlichkeit des Sourcecodes beeinflusst die Wart- und Änderbarkeit sowie Fehleranfälligkeit stark. (Beizer, 1984)

"Good testing works best on good code and good design. And no testing technique can ever change garbage into gold." Beizer (1984)

Somit wird die Durchführung einer statischen Analyse auf Basis des Sourcecodes empfohlen. Eine umfangreichere manuelle Analyse des Codes in Form eines Reviews würde sich bei GSD-CS nur selten lohnen. Der Aufwand wäre im Vergleich zum erzielbaren Nutzen zu hoch. Die in Kapitel 2.3.1.3 vorgestellte werkzeuggestützte statische Codeanalyse würde sich jedoch gut dafür eignen Schwachstellen im Code zu identifizieren. Typischerweise prüfen entsprechende Werkzeuge mindestens auf Konstrukte die:

- Fehler begünstigen
- Qualität/Performanz mindern
- Portabilität verhindern
- Verständlichkeit reduzieren

Fehler werden zum Beispiel durch so genannte *Code-Smells*[15] begünstigt. Beispiele hierfür wären zu große Methoden oder Klassen oder ein falsches Einsetzen von Exceptions. Die Qualität wird unter Anderem durch Datenflussanomalien[16] gemindert. Performanzlastig ist beispielsweise wenn Properties[17] Arrays zurückgeben da durch ständiges Kopieren des Arrays viele Iterationsdurchläufen entstehen. Portabilität wird verhindert, wenn systemspezifische Aufrufe benutzt werden. Vor allem kann eine statische Analyse die Verständlichkeit erhöhen, da auch das syntaktische Design des Codes beispielsweise auf Naming-Konventionen wie

[15] Ein Code-Smell ist ein Konstrukt im Source Code welches vermieden werden sollte. Der Code sollte deshalb refaktorisiert oder neu begutachtet werden. Der Begriff stammt ursprünglich von Kent Beck. Ob ein Code-Smell vorliegt oder nicht, ist jedoch oft subjektiv und hängt unter Anderem vom Programmierer, von der Programmiersprache, und der Entwicklungsmethode ab.

16 Fehlerhafte Sequenz von Attribut- bzw. Objektzugriffen. z.B. der Zugriff auf eine Variable oder ein Attribut bevor es erzeugt wurde (Null-Pointer Exception) oder die zweimalige Wertzuweisung ohne Verwendung des ersten Wertes

[17] Eine Property ist in C# eine Erweiterung von Datenfeldern in Klassen.

Pascal-casing[18] überprüft wird. Auch wird auch auf Abhängigkeiten zwischen den Klassen geachtet. Zudem kann ein Werkzeug zur statischen Analyse oft dupliziertem Code identifizieren sowie den Code unter anderem auf *Lazy Classes* oder *God-Classes*[19] überprüfen.

Zur Fehlerbehebung muss Wissen über den Quelltext vorhanden sein muss, weshalb diese Tests durch entwicklungsnahe Personen, wie z.B. den Programmierer selbst, durchgeführt werden sollten. Da meistens gezielt eine Komponente untersucht wird, bietet sich die Ausführung direkt nach dem Komponententest an. Eine Ausführung vor dem Komponententest wird nicht empfohlen, da die Erstellung und Ausführung der Testfälle der Komponententests parallel zum Entwicklungsprozess beginnen soll. In einem weiteren Schritt könnte somit auch über Test-Driven-Developement[20] nachgedacht werden, wobei eine Ausführung der werkzeuggestützten statischen Analyse vor den Komponententests wenig Sinn machen würde. Zudem sollte nach Kent Beck die Optimierung des Sourcecodes erst am Ende der Entwicklung der Komponente stattfinden.

"Make it run, make it right, make it fast." (Kent Beck, 2000)

Obgleich es aufgrund mangelnder Dokumentation der Tests keine, für diese Arbeit, geeigneten Zahlen gibt die belegen könnten wie viel Prozent der Fehler mit einer werkzeuggestützten statischen Analyse durchschnittlich aufgedeckt werden könnten, schätzt der Autor Werkzeuge dieser Art für GSD-CS als sehr hilfreich ein. Im Laufe der Arbeit konnte durch Begutachtung des Codes im *Repository* festgestellt werden, dass durchaus teilweise *Code-Smells* und fehlerbegünstigende Konstrukte im hinterlegten Sourcecode vorhanden sind. Außerdem werden Komponenten oft von einzelnen Entwicklern geschrieben, was ein Hinwegsehen über qualitative Mängel des Codes fördern kann (Kapitel 2.2, Psychologie des Softwaretestens). Zudem haben die Interviews ergeben, dass einzelne Entwickler schon aus Eigenmotivation entsprechende Werkzeuge einsetzen und es als sinnvolle Maßnahme erachten.

---

[18] Bei der Pascal-cased Notation beginnen Namen mit einem Großbuchstaben und werden logisch mit weiteren Großbuchstaben getrennt

[19] Eine Lazy Class ist eine Klasse die zu wenig Funktionalität bietet um den Aufwand einer Klassenerstellung zu rechtfertigen. Das Gegenteil ist eine God Class, die fast die komplette Funktionalität des Programms übernimmt.

[20] Beim Test-Driven-Developement werden die Tests konsequent vor der zu entwickelnden Komponente oder Klasse geschrieben. Das Vorgehen stammt ursprünglich aus dem Extreme Programming.

Durch die vollständige Automatisierung der Analyse ist der Aufwand des Einsatzes eines solchen Werkzeuges sehr gering. Die Kosten für die Behebung gefundener Fehler würden in späteren Entwicklungsstadien sehr viel höher ausfallen. (Kapitel 2.4.1 Low-Level-Tests, Abbildung 7; Boehm, 1982). Passende Werkzeuge für den Komponententest, die Codeüberdeckung, sowie die automatisierte statische Codeanalyse werden in Kapitel 4.1.3 evaluiert. Abbildung 13 visualisiert die geplante Abfolge der einzelnen Phasen des Komponententests.

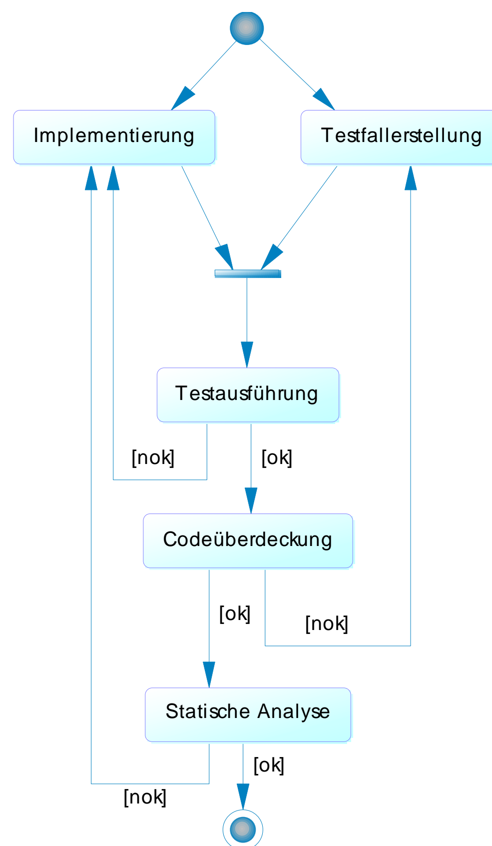


Abbildung 13: Flussdiagramm Komponententest

4.1.1.2 Integrationstest

Der Integrationstest soll bestehende Probleme im Zusammenspiel mit einzelnen Komponenten aufdecken. Integrationen sind bei GSD-CS selten sehr aufwändig da die Systeme und Anwendungen meistens nur aus wenigen Komponenten bestehen. Daher ist eine Big-Bang-Integration nicht wirklich riskant. Vorrausetzung für einen nützlichen Integrationstest bei GSD-CS wäre ein geringer Aufwand. Automation empfiehlt sich als

Mittel zur Aufwandsreduktion wenn ein nicht sonderlich spezifisches Ergebnis gefordert ist oder wenn oft dieselben technischen Gegebenheiten vorzufinden sind. Automatisierte Integrationstests können mit Hilfe von kontinuierlicher Integration durchgeführt werden. Von fertigen Systemeinheiten werden täglich, potentiell mehrmals, *Builds* erzeugt. Die Voraussetzung des *Repository Servers* ist bei GSD-CS vorhanden. Aus diesem Server wird der Quellcode abgerufen, kompiliert und optional *deployed* (siehe Abbildung 14). Die Buildroutine startet dabei zeitabhängig oder ereignisabhängig. Das heißt, dass der Prozess entweder zu einem bestimmten Zeitpunkt, oder wenn der Programmierer zum Beispiel eine neue Version in den *Repository Server* eincheckt stattfindet, was jedoch relativ aufwändig ist. Im ersten Schritt ist ein *Nightly Build*, sprich ein zeitabhängiger Buildprozess über Nacht zu empfehlen. Falls Fehler auftreten, wird die entsprechende Datei identifiziert und der Entwickler benachrichtigt. Somit werden Fehler in einer neuen oder geänderten Komponente schnell durch die Integration mit dem restlichen System aufgedeckt. Einmal konfiguriert, läuft der *Compile*-, *Build*-, und Testprozess in einer kontinuierliche Integrationsumgebung automatisch ab weshalb nur geringer Aufwand entsteht und somit die Vorrausetzungen bei GSD-CS erfüllt. Die Evaluierung passender Umgebungen ist Inhalt von Kapitel 4.1.3.

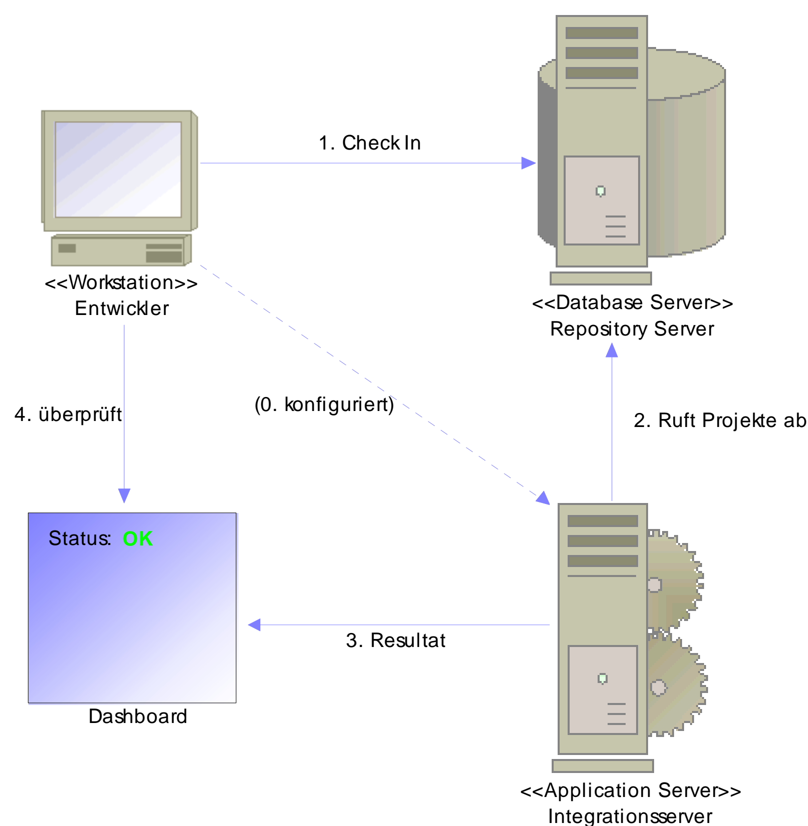


Abbildung 14: Interaktion zwischen Repository, Entwickler und Integrationsserver

4.1.1.3 Systemtest

Der Systemtest ist die umfangreichste und die letzte Teststufe vor dem Abnahmetest. Die fachlichen Anforderungen an das Gesamtsystem sollen getestet werden.
Aktuell wird bei GSD-CS erfahrungsbasiert getestet. Aufgrund der Anwendungslandschaft ist hier eine Testautomation nicht zu empfehlen. Der Konfigurationsaufwand wäre verglichen mit dem Gewinn zu hoch. Bei erfahrungsbasiertem Testen ist es wichtig dass der Tester Erfahrung im Umgang mit dem Programm hat. Dies ist im Allgemeinen bei GSD-CS gegeben. Dennoch ist das kein Garant für aussagekräftige Testfälle. Die bisherige erfahrungsbasierte Erstellung der Testfälle ist grundsätzlich und vielfach ausreichend für den Systemtest. Jedoch sollte man versuchen mit der Erstellung strukturierter und systematischer vorzugehen um mit weniger Testfällen eine höhere Anzahl Fehler zu finden, beziehungsweise mehr fehlerträchtige Stellen abzudecken. Anhand von Orientierungspunkten können Testfälle erstellt werden, die auf unterschiedliche Qualitätsmerkmale und Softwareaspekte abzielen. Die Analyse hat ergeben, dass für GSD-CS die funktionalen Qualitätsmerkmale Korrektheit und Robustheit und das nicht-funktionale Qualitätsmerkmal Bedienbarkeit am wichtigsten sind. Wie auf diese Qualitätsmerkmale getestet werden kann, ist im nächsten Kapitel geschildert.

Zudem fehlt eine passende Testberichterstattung. Die Testresultate müssen protokolliert und veröffentlicht werden. Der erste Schritt zu einer Testberichterstattung ist dass überhaupt dokumentiert wird (Pol und Koomen, 1999). Die Anzahl der Fehler unterteilt in gelöste und ungelöste Defekte müssen parallel zum Softwarelebenszyklus aufgeführt werden. Die Berichterstattung muss regelmäßig und frühzeitig stattfinden damit das Projekt genügend Spielraum für Änderungen und Nachbesserungen hat.

Die Dokumentation der Testfälle garantiert ein effektiveres Vorgehen bei möglichen Fehlernachttests. Eine hohe Widerverwendbarkeit der Testfälle wird angestrebt. Somit kann auch nachvollzogen werden inwiefern die Testfälle den Anforderungen entsprechen. Das schließt auch eine weitere potentielle Verbesserungsmöglichkeit ein: die systematische Ableitung der Testfälle aus den fachlichen Anforderungen. Dazu ist jedoch eine formalere Form des Anforderungsanalyse und des Definieren der funktionalen und nicht-funktionalen Anforderungen nötig, was ein nächster Schritt für GSD-CS wäre, der jedoch im Rahmen dieser Arbeit nicht weiter berücksichtigt wird.

4.1.2 Technik

Vorbedingung für einen aussagekräftigen Test ist die Auswahl einer geeigneten Testtechnik. Der vorgeschlagene Komponententest ist codebasiert und schließt grob zwei Aktivitäten ein: das „Unittesten" und die automatisierte statische Analyse. Für das „Unittesten" müssen passende White-Box-Techniken bekannt sein um aussagekräftige Testfälle erstellt zu können, die zudem eine angemessene Codeüberdeckung erreichen. Die statische Analyse läuft werkzeuggestützt und automatisch. Hier ist somit als Technik nur die Bedienung des Werkzeuges wichtig.

Der Integrationstest funktioniert ebenfalls automatisiert. Somit ist es nur notwendig, dass Projekt initial für die Integrationsumgebung konfigurieren zu können. Der Build- und Integrationsprozess läuft anschließend selbstständig.

Aufgrund der Art der Systeme wird der Systemtest weiterhin erfahrungsbasiert ausgeführt. Jedoch sollte er mehr Struktur als in seiner bisherigen Ausführung haben. Die drei Qualitätsmerkmale, Korrektheit, Robustheit und Bedienbarkeit müssen dabei isoliert betrachtet werden. Beim Test auf Korrektheit ist besonders drauf zu achten, dass passende Eingabedaten ausgewählt werden. Aufgrund der Erfahrung der Tester im Umgang mit den Softwaresystemen und der Art der Daten (oftmals Auswertung von Absatz- oder Lieferzahlen durch die BI-Systeme) kann hierbei der erfahrungsbasierte Ansatz beibehalten werden. Verbessern kann man einen solchen Ansatz jedoch, in dem versucht wird die Schwachstellen der Software zu identifizieren um dort ein genaueres Testen zu ermöglichen. Schwachstellen sind hier typischerweise Teile des Systems, die häufig geändert wurden oder in denen schon viele Fehler gefunden wurden. Also Schwachstellen, als Konsequenz von Änderungen in den Anforderungen, von Fehlerkorrekturen und Fehlerhäufung. Neben diesen Regressionstests und Fehlernachtests kann eine Äquivalenzklassen- und Grenzwertanalyse dazu dienen aussagekräftige Testfälle zu konzipieren. Wie hierbei vorgegangen werden kann ist im Testhandbuch erläutert.

Ein Test auf Robustheit sollte auch Teil dieser Teststufe sein. Die Fehlerbehandlung soll überprüft werden, beispielsweise durch Abbruch des Programms oder des Vorgangs zu einem unerwarteten Zeitpunkt oder Fehlbedienung wie die Eingabe ungültiger Zeichen oder das Einlesen eines ungültigen Datenformats. Zum anderen sollte überprüft werden, wie die Software auf ungültige, aber definierte Werte reagiert. Typische Beispiele wären hier:

negative Werte, zu hohe oder zu niedrige Werte, Null, oder Leerstrings. Im Testhandbuch sind weitere potentielle Schwachstellen die überprüft werden sollten aufgeführt. Die Auswahl liegt hier aber auch wieder in der Verantwortung des Testers.

Der Test auf Bedienbarkeit ist fast nur mit Hilfe einer Einschätzung und Kommunikation mit den Anwendern zu vollziehen. Eine unabhängige Maßzahl die die Benutzbarkeit in einem gewissen Maße bewerten kann, ist die Anzahl der Eingaben die für die Durchführung eines Geschäftsprozesses zu tätigen hat. Anhand einer Dokumentation sowie Priorisierung der Geschäftsprozesse und Arbeitsabläufe kann man anschließend die Bedienungseffizienz des Systems abzuschätzen. Folgend kann man versuchen in der Entwicklung die Anzahl der Eingaben für besonders oft vorkommende Abläufe zu minimieren.

Wie oben erwähnt, wird für den Systemtest im Testhandbuch ein Ablauf vorgeschlagen, bei dem die Tests der drei Qualitätsmerkmale unterschieden werden. Sie sollten separat hintereinander ausgeführt werden mit Hilfe der oben und im Testhandbuch aufgeführten Vorgehensweisen. Dadurch soll gewährleistet werden, dass möglichst Aspekte der Software betrachtet werden. Zudem sollten, wie in Kapitel 4.1.1.3 erläutert, die Testfälle und die Testdurchführung dokumentiert werden um die Widerverwendbarkeit der Tests zu gewährleisten. Auch soll dadurch überprüft werden können, ob die Testfälle den funktionalen und nicht-funktionalen Anforderungen entsprechen und diese tatsächlich geprüft wurden.

4.1.3 Infrastruktur und Werkzeuge

Im Rahmen der bisherigen Arbeit wurden mehrere Möglichkeiten zur werkzeugbasierten Unterstützung der jeweiligen Testphasen identifiziert. Die Einführung von Werkzeugen ist jedoch nur dann sinnvoll, wenn einzelne Testaktivitäten tatsächlich effektiver durchgeführt werden können. Das jeweilige Werkzeug muss also entweder Zeit sparen, oder den Umfang bzw. die Qualität des Ergebnisses erhöhen. Manche Aktivitäten werden auch erst durch den Einsatz von Testwerkzeugen ermöglicht, da ohne (Teil-)Automatisierung die Kosten den Nutzen übersteigen würden. Jedoch kann eine Einführung eines ineffizienten oder unpassenden Werkzeugs kontraproduktiv sein und mehr Zeit oder Ressourcen kosten oder eine Prozessänderung erzwingen die den gewonnenen positiven Effekt wieder zunichte macht. Deshalb müssen bei der Einführung von Werkzeugen folgende Punkte bedacht werden:

- Identifikation und Quantifizierung der Ziele
- Betrachtung möglicher Alternativen
- Kosten-Nutzen-Analyse
- Identifikation von Einschränkungen

Durch die vorgeschlagenen Maßnahmen zur Testprozessverbesserung werden vier Arten von Werkzeugen benötigt:

- Automatische Statische Analyse
- Komponententestwerkzeug
- Code-Überdeckung
- Kontinuierliche Integration

Folgend wird eine konkrete Implementierung die in einer Vorabanalyse recherchiert wurden evaluiert.

4.1.3.1 Werkzeuggestützte statische Analyse

Für das .NET Framework existiert zur automatisierten statischen Analyse das Werkzeug *fxCop* von Microsoft. Es wird die zur Zeit (3.6.2008) aktuelle Version 1.36 betrachtet. *FxCop* führt eine Analyse anhand der *managed Code Assemblies*[21] durch und hat somit einen großen Vorteil gegenüber den meisten ähnlichen Werkzeugen: Eine rein textbasierte Analyse des vom Programmierer verfassten Codes bleibt aus. Aufgrund der umfangreichen Informationen die im Assembly gespeichert werden ist es für fxCop möglich sehr gute Ergebnisse zu liefern. FxCop besteht prinzipiell aus den Elementen Ziele, Nachrichten und Regeln. Ziele stellen sämtliche Strukturen des Assemblies dar, die einer Analyse unterzogen werden. Beispielsweise Namensraume, Methoden, Konstruktoren, und Ressourcen. Nachrichten sind die Rückmeldungen der Regeln. Diese informieren den Benutzer über Regelverletzungen. Sie geben detaillierte Informationen über Herkunft und Hintergrund der Regel sowie Empfehlungen und Tipps, wie diese eingehalten werden kann. Die Ziele sollen die Regeln einhalten. Es gibt sechs Kategorien von Regeln:

---

[21] In der .NET-Umgebung wird die Zusammenstellung übersetzter Programmklassen, welche anschließend in der .NET-Laufzeitumgebung ausgeführt werden als *managed Code Assemblies* bezeichnet. Diese sind beispielsweise vergleichbar mit den .JAR-Dateien in Java.

- *Com Rules* beziehen sich auf COM-Programmierung. Beispielsweise wird geprüft, ob COM-Typen einen öffentlichen Standardkonstruktor besitzen.
- *Design Rules* prüfen das Softwaredesign. Unter anderem auf, wie Exceptions richtig eingesetzt werden – das Abfangen von „System.Exception“ wird hier sofort moniert.
- *Naming Rules* prüfen Namenskonventionen. Ein Beispiel ist das Pascal-Casing von Methoden.
- *Performance Rules* prüfen die Performanz.. So sollen beispielsweise wie oben angemerkt Properties keine Arrays zurückliefern, da durch ständiges Kopieren von Arrays Performanzprobleme entstehen.
- *Security Rules* beschreiben präventive Maßnahmen gegen Sicherheitslücken.
- *Globalization Rules* unterstützen die Erstellung von lokalisierbaren Anwendungen. Hierzu zählt die lokalisierte Ausgabe von Zahlen oder Datumsangaben.
- *Usage Rules* beugen fehlerhaftem Code vor, der als solcher nicht direkt sichtbar ist. Beispielsweise sollen Konstruktoren keine virtuellen Methoden aufrufen.

FxCop beinhaltet GUI und Command-Line Versionen und unterstützt .NET 1.x, .NET 2.0 und .NET 3.x.

Ziele

Die Möglichkeiten von solchen Werkzeugen reichen von der Sicherstellung von einfachen Coding-Standards, und der Aufdeckung von *Code Smells* (wie z.B. dupliziertem Code), hin zu der Prüfung von Typumwandlungen oder Bereichsgrenzen. Neben dem Auffinden von semantischen Fehlern können somit auch qualitative Mängel aufgedeckt werden. Für weitere Vorteile ist auf Kapitel 2.3.1.3 (Werkzeuggestützte statische Codeanalyse) sowie Kapitel 4.1.1.1 (Komponententest) verwiesen, in dem die Vorteile einer werkzeuggestützen statischen Analyse bei GSD-CS erläutert werden.

Zusammenfassend ist die Ausführung von fxCop ein einfach durchführbares Mittel zur frühzeitigen präventiven Fehlerfindung und dem Identifizieren von fehlerträchtigen Stellen.

Kosten-Nutzen-Vergleich

Da fxCop auch für kommerzielle Unternehmen kostenlos einzusetzen ist, fallen keine Anschaffungskosten an. Das letzte Update wurde am 10.10.2007 herausgegeben. Es ist anzunehmen, dass zu neueren .NET-Versionen auch fxCop passend erweitert wird.

Folgekosten sind kaum vorhanden. Die Hardware muss nicht erweitert werden und eine Einarbeitung ist mit einem Aufwand von etwa 1-2 Stunden zu bemessen.

Der Nutzwert äußert sich in einer höheren Softwarequalität und Lesbarkeit. Somit sinken die Wartungs- und Fehlerbehebungskosten. Weiterhin wird durch früh gefundene Fehler der Systemtest entlastet.

Einschränkungen

Im Falle komplexer Software ist ein zusätzliches manuelles Review nicht zu ersetzen. Jedoch kann eine werkzeuggestützte statische Analyse schon vielfältige Anhaltspunkte für das manuelle Review geben und dieses entlasten.

Bewertung

Aufgrund der geringen Kosten und unkomplizierten Einführung ist die Verwendung von fxCop sehr zu empfehlen. Die in den Grundlagen und im Konzept genannten Vorteile der werkzeuggestützten statischen Analyse können durch den Einsatz von fxCop erreicht werden.

„FxCop sollte in jedem Projekt eingesetzt werden, um (…) robusteren Code in Produktion zu geben. Die Erfahrung hat gezeigt, dass der Einsatz des Tools den besonders positiven Nebeneffekt hat, dass Entwickler jeder Erfahrungsstufe Ihren eigenen Stil verbessern und mehr über die Funktionsweise des .NET Frameworks gelernt haben – somit kann das Know-how des Teams und des Einzelnen gesteigert werden“ (Hüttl, 2004).

4.1.3.2 Komponententests und Code-Abdeckung

Für fast jede gängige Programmiersprache sind Frameworks für einen Komponententest vorhanden. Exemplarisch wird hier jedoch nur eine konkrete Implementierungen für .NET die in einer Vorabanalyse recherchiert wurde evaluiert. Ergänzend dazu wird ein Werkzeug für die Codeabdeckung der Komponententests vorgestellt. Diese beiden Werkzeuge werden gemeinsam betrachtet, da die Code-Abdeckung direkt auf dem Komponententest basiert.

Eine Vorhabrecherche hat *NUnit* als Marktführer identifiziert. *Nunit* ist eigentlich eine Standalone-Applikation, die sich jedoch mit dem *Visual Studio* Add-In *testdriven.NET* gut in die bei GSD-CS eingesetzte Entwicklungsumgebung integrieren lässt. Außerdem liefert das Add-In das Werkzeug *NCover* mit, was eine Analyse bezüglich Pfadabdeckung,

Sequenzpunkten, und Funktionspunkten ermöglicht. Dadurch erlangt man einen Einblick in die Aussagekraft der Testfälle und man kann eine Hinlänglichkeitskriterium definieren, wieviel Prozent des Codes durch Testfälle abgedeckt werden müssen.

Ziele

Die Ziele und Vorteile der Komponententests sind in den Grundlagen (Kapitel 2.4.1, Low-Level-Tests) sowie im Konzept (Kapitel 4.1.1.1, Komponententest) zu finden.

Alternativen

Für die Programmierplattform *Visual Studio 2008 Professional* existieren mehrere weitere Lösungen für Komponententests, wie z.B. ein bereits mitgeliefertes Framework von Microsoft. Dies bietet einen automatischen Teststubgenerator und einen Wizard für eine Testsuiteerstellung sowie natürlich das *Visual Studio Look and Feel*. Außerdem ist das Testen von privaten Membern[22] möglich. Aufgrund des größeren Funktionsumfangs ist aktuell der Defakto-Standard in der Industrie jedoch wie oben angegeben die Open-Source-Lösung *NUnit.*

Die *Visual Studio 2008 Versionen Team Edition* und *Test Edition* bieten einen ähnlichen Umfang wie *NUnit* mit *testdriven.NET* und *NCover*. Da diese Version vom Visual Studio aber ein vielfaches mehr kostet als die aktuell eingesetzte Professional-Version und weit aus mehr als die vorgeschlagene Software, ist der Einsatz anderer Programmierplattformen keine Alternative.

Kosten-Nutzen-Vergleich

Anschaffungskosten entstehen für *NUnit* nicht, da es Open-Source ist. Jedoch kostet das Add-In *testdriven.NET* mit dem Code-Abdeckungswerkzeug *Ncover* 90 USD pro Einzelplatzlizenz. Der Einsatz dieser Werkzeuge erhöht jedoch die Effizienz von NUnit ungemein.

Folgekosten entstehen hier auf der Seite der Einarbeitung. Das Werkzeug ist intuitiv zu bedienen, jedoch müssen für effektives Komponententesten die Mitarbeiter im Einsatz der

22 In der .NET-Umgebung werden Instanzmethoden oder -variablen als Member bezeichnet. Das Schlüsselwort privat ist ein Zugriffsmodifizierer. Auf private Member kann nur im Body der Klasse oder der Struktur zugegriffen werden.

spezifischen Techniken geschult sein. Trotz zusätzlicher Kosten durch die Erstellung der Testfälle, können sie helfen die Gesamtkosten des Testens zu reduzieren. Erstellt sind ihre Ergebnisse einfach zu analysieren und wiederzuverwenden. Vor allem durch Softwareänderungen und –erweiterungen bedingte Regressionstests lassen sich somit sehr kostengünstig durchführen.

Einschränkungen

Mit Komponententests können nur Codier- und Logikfehler gefunden werden. Das schmälert jedoch ihren Nutzen nicht, da durch das frühzeitige Aufdecken dieser Fehler die späteren Testphasen entlastet werden können.

Bewertung

In Anbetracht der unkomplizierten Implementierung könnten die Komponententests mit NUnit sehr von Nutzen sein. Auch wenn aufgrund der heterogenen Umgebung sie nicht überall verwendet werden können, stellen sie doch für die Eigenentwicklungen mit ASP.NET und Windows Forms eine sehr empfehlenswerte Maßnahme zur Qualitätsverbesserung dar. Für weitere hier eingesetzte Programmiersprachen wie PL/SQL existieren auch äquivalente Frameworks (für PL/SQL beispielsweise utPLSQL) welche jedoch noch evaluiert werden müssten. Zudem muss abgewogen werden, wie intensiv die Komponententests durchgeführt werden dürfen, um das Ziel nicht zu verfehlen. Bereits in Kapitel 2.1 (Ziele des Softwaretestens) wurde gezeigt, dass die Schwierigkeit darin liegt, für jede Testsufe das Minimum der Summe der geschätzen Fehlerbehebungs- und Testkosten zu treffen. Beim Verzicht auf Komponententests kann ein durchgereichter Fehler jedoch hohe Kosten verursachen. (Abbildung 7; Boehm, 1982)

Die Messung der Codeabdeckung mittels *NCover* ist ebenfalls sehr hilfreich, da auf diese Weise die Qualität der Testfälle zumindest hinsichtlich der Vollständigkeit gewährleistet werden kann.

„In this way, you will insure that the quality of your software is not undermined by inadequate testing.” (web: Gupta, 2006)

Da dies bei *Visual Studio* in diesem Umfang nur mit der sehr viel teureren *Team Suite* oder *Test Edition* möglich wäre, bietet sich hier *testdriven.NET* zusammen mit *NUnit* für *GSD-CS*

gut an. Prinzipiell lassen sich für die meisten Programmiersprachen, wie beispielsweise PL/SQL Frameworks für Komponententests mit geringem Aufwand in den Entwicklungsprozess integrieren.

4.1.3.4 Kontinuierliche Integration

Für die .NET Umgebung hat sich *CruiseControl.NET* als führendes Werkzeug herauskristallisiert. Es läuft optimalerweise auf einem separaten Server als Windowsdienst und lässt sich unkompliziert an bestehende Versionsverwaltungsprogramme wie das bei *BAT GSD-CS* eingesetzte *Microsoft SourceSafe* anbinden und unterstützt auch das automatisierte Ausführen von *NUnit* und *fxCop*. Das Resultat des Prozesses lässt sich über das integrierte Werkzeug *Dashboard* online über einen Webbrowser einsehen. Der vorgeschlagene Arbeitsablauf des Integrationstests ist in Abbildung 15 dargestellt.

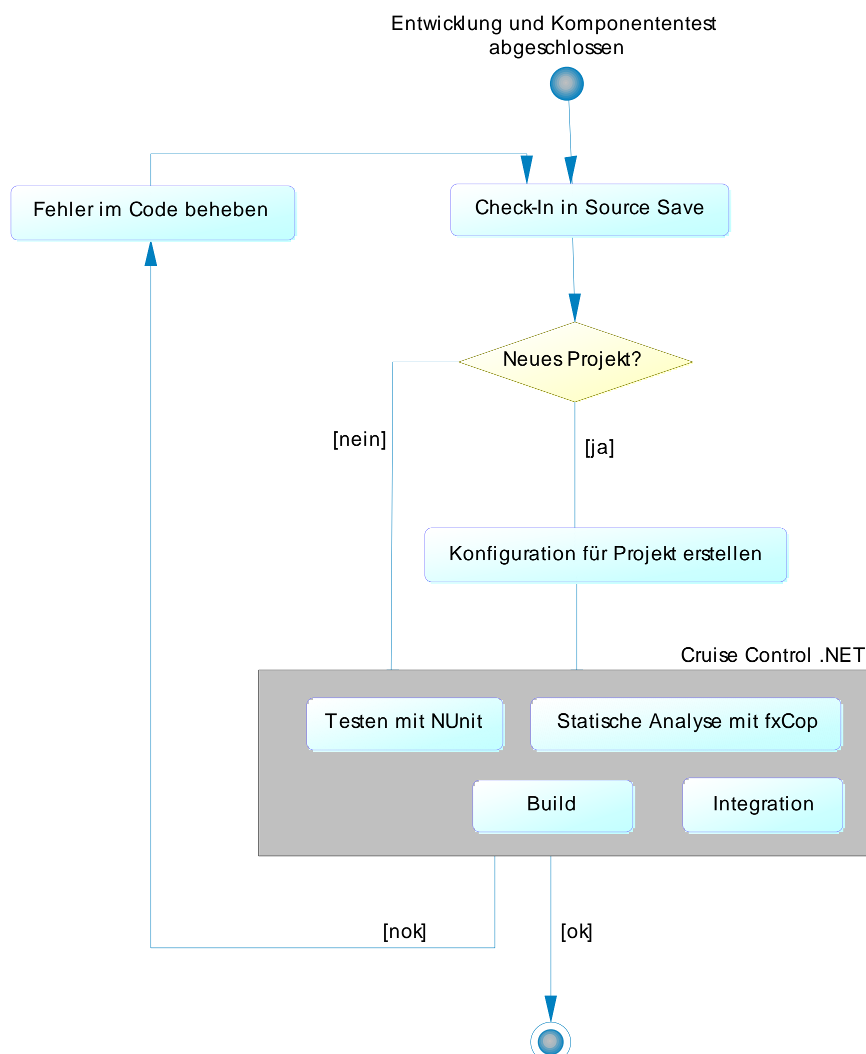


Abbildung 15: Arbeitsablauf des Integrationstests bei GSD-CS

Ziele

Ziele eines Werkzeugs für kontinuierliche Integration sind die automatisierte Integration von Softwarekomponenten, das frühzeitige Finden von Integrationsproblemen. Folglich sollen das Risiko einer unerwarteten Verzögerung minimiert und hohen Fehlerbehebungskosten vermieden werden.

Alternativen

CruiseControl.NET ist einer der Marktführer für die .NET-Umgebung. Jedoch sind die weiteren Konkurrenten nicht Open-Source und haben hohe AnschaffungKosten, und sind somit nicht für einen Einsatz bei GSD-CS geeignet.

*Kosten-Nutzen-*Vergleich

Es entstehen nur marginale Anschaffungskosten. CruiseControl.NET ist Open-Source, läuft jedoch optimalerweise auf einem separaten Server.

Als Einarbeitungsaufwand werden 1-2 Stunden veranschlagt um sich mit den Konfigurationsdateien auseinanderzusetzen, die für jedes Projekt angelegt werden müssen. Einmal erstellt, dürfte der zusätzliche Aufwand für ein neues Projekt jedoch nur wenige Minuten betragen. Weiterhin ermöglicht das Werkzeug *CCNetConfig* eine GUI-basierte Erstellung der Konfigurationsdateien. Das Werkzeug bietet eine Profilierung von Standardeinstellungen an, die die Projektinitiierung durch den Entwickler erleichtert.

Einschränkungen

Das Werkzeug hat bezüglich der benötigten Anforderungen keinerlei Einschränkung.

Bewertung

Geringer Initialaufwand sowie geringe Einarbeitungskosten machen CruiseControl.NET zu einem sehr empfehlenswerten Werkzeug für das Integrationstesten. Zusätzlich gibt es noch die Möglichkeit Komponententests und automatische statische Analyse durchzuführen und die Ergebnisse online einzusehen.

„CruiseControl bringt eine deutliche Qualitätsverbesserung (…). Durch die ständige Integration der Arbeiten aller Entwickler werden mögliche Fehler früh entdeckt. Durch kontinuierliches erfolgreiches Bauen eines Projekts kann außerdem die Moral eines

Entwicklungsteams nachhaltig gestärkt werden. Der initiale Einführungsaufwand für CruiseControl ist für (…) Projekte, die bereits auf CVS (…) basieren, ausgesprochen gering und wird von den zuvor genannten Vorteilen mehr als aufgewogen.“ (web: Schluff, 2003)

4.1.4 Phasenmodell

Jede Teststufe benötigt einen definierten Prozess mit den jeweils durchzuführenden Testaktivitäten. Dabei folgt jede Testphase der selben Prozessstruktur, jedoch sind die Aktivitäten jeweils unterschiedlich. Hierfür wird eine Anleitung, beispielsweise in Form eines Handbuchs benötigt, in dem zu jeder Phase die geeignete Abfolge der Aktivitäten dokumentiert ist. Der genaue Inhalt leitet sich von den konzentionierten Maßnahmen ab und wird im Testhandbuch konkretisiert.

4.2 Konzept des Testhandbuchs

Das Testhandbuch bildet das Fundament der Teststrategie. In ihm sollen dem Vorgehensmodell entsprechend alle Teststufen dokumentiert sein. Für jede Teststufe, müssten die vier Eckpfeiler des strukturierten Testens, nämlich Organisation, Techniken, Infrastruktur und Werkzeuge sowie das Phasenmodell aufgeführt sein.

Die Interviews haben zudem ergeben, dass keine einheitliche Begriffsterminologie verwendet oder bekannt ist. Somit ist ein Glossar am Ende des Testhandbuchs vorhanden.

4.2.1 Ziele für Tester und Entwickler

Das Testhandbuch versucht folgende Ziele und Vorteile für die direkt am Testen beteiligten Personen zu erreichen:

- Es müssen aus dem Testhandbuch die zu verwendenden Methoden für jede Teststufe ersichtlich sein.
- Es soll eine Hilfestellung für neue Mitarbeiter sein.
- Es muss klar sein, welche organisatorischen Maßnahmen vor dem Testen für jede Teststufe durchgeführt werden müssen.
- Es muss klar sein, welche Infrastruktur und welche Werkzeuge für die Teststufen benötigt werden, und ob eine Vorbereitung dieser Teststufen nötig ist.

- Es muss der Ablauf jeder Teststufe strukturiert erkennbar sein.
- Das Testhandbuch soll und muss um die beim Testen gemachten „Best Practices“ regelmäßig erweitert werden, um auf diese Weise eine nachhaltige und aktuelle Grundlage für die Testaktivitäten bei GSD-CS darstellen zu können.

Das Testhandbuch lebt zudem von der Akzeptanz durch den Anwender, also der am Test beteiligten Personen. Es bedarf regelmäßigen Feedbacks und kontinuierlicher Verbesserung. Somit kann das Testhandbuch als wichtiges Instrument im Wissenstransfer zu nachfolgenden Projekten dienen. (Spillner und Linz, 2005).

4.2.1 Ziele für das Management

Für das Management hat das Konzept des Testhandbuchs folgende Ziele:

- Der Testprozess wird mess- und steuerbar.
- Der Testprozess wird kostentransparent.
- Durch standardisiertes Vorgehen beim Testen können die Ergebnisse des Testens Rückschlüsse auf die Qualität der Software geben.
- Der Testprozess kann schneller an sich ändernde Gegebenheiten angepasst werden.
- Durch die Erweiterung des Testhandbuchs mittels weiterer Methoden oder Stufen, lässt sich der Testprozess unkompliziert auf eine höhere Qualitätsstufe heben.
- Anhand des Testhandbuchs lassen sich Ausgaben und Kosten einfacher zerlegen und Potential zur Effektivitätssteigerung erkennen.
- Anhand des definierten Testprozesses lassen sich Ressourcen einfacher und genauer planen.

Nach DeMarco (1982) muss gemessen und überwacht werden, um zu kontrollieren. Ein untransparentes Gebilde lässt sich demnach nicht effektiv steuern. Weiterhin bedeutet eine Effizienzsteigerung durch den Einsatz der Testmethoden und Testwerkzeugen sowie durch eine höhere Codequalität ein geringeres Risiko schwerwiegender Fehler. Dies äußert sich in höherer Kundenzufriedenheit und geringerem Wartungs- und Fehlerbehebungsaufwand.

4.2.3 Struktur

Das Testhandbuch ist in 4 Abschnitte unterteilt: Inhalt, Einleitung, Teststufen und Glossar. In Abschnitt 2 sind Sinn und Zweck sowie die Form des Testhandbuchs aufgeführt. Dann folgt gemäß Abbildung 16 eine Beschreibung des konzipierten Testprozesses und der Phasenorientierung der Teststufen. Dem Leser soll dadurch ein Überblick was von diese Dokument zu erwarten ist und welche Informationen an der entsprechenden Stelle zu finden sind.

2. Einleitung **3**
2.1 Beschreibung *3*
2.2 Testprozess *4*

Abbildung 16: Testhandbuch: Kapitel 2

Im Kapitel 3 folgt die genaue Beschreibung des Ablaufs der einzelnen Phasen sowie deren Ein- und Ausgangsbedingungen. Zu jeder Phase sind zudem die vier Eckpfeiler des strukturierten Testens beschrieben: Organisation, Infrastruktur, Techniken, und Phasen. (Abbildung 17).

3. Testphasen **6**
3.1 Komponententest *6*
3.1.1 Eingangsbedingung 7
3.1.2 Organisation 7
3.1.3 Infrastruktur und Werkzeuge 8
3.1.4 Techniken 8
3.1.5 Phasen 8
3.1.6 Ausgangsbedingung 8
3.2 Integrationstest *9*
3.2.1 Eingangsbedingung 11
3.2.2 Organisation 11
3.2.3 Infrastruktur und Werkzeuge 11
3.2.4 Techniken 11
3.2.5 Phasen 14
3.2.6 Ausgangsbedingung 15
3.3 Systemtest *15*
3.3.1 Eingangsbedingung 16
3.3.2 Organisation 16
3.3.3 Infrastruktur und Werkzeuge 17
3.3.4 Techniken 17
3.3.5 Phasen 17
3.3.6 Ausgangsbedingung 18

Abbildung 17: Testhandbuch: Kapitel 3

Das letzte Kapitel bildet das Glossar. Eine einheitliche Begriffsterminologie ist das Fundament einer effektiven und guten Zusammenarbeit aller Beteiligter. In der Analyse wurde klar, dass diese bisher nicht vorhanden ist. Ein Nachschlagewerk für die wichtigsten Begriffe und Fundament für stetige Erweiterung ist zudem eine sinnvolle Ergänzung zur Beschreibung des Testprozesses und dessen Methoden und sollte ebenfalls regelmäßig aktualisiert und erweitert werden.

4.2.4 Beispiel: Integrationstest

Exemplarisch für die anderen Kapitel, wird im Anhang B das Kapitel 3.2: Integrationstest des Testhandbuchs aufgeführt. Für die vollständige Einsicht in den konzeptionellen Draft des Testhandbuchs wird auf die digitalen Anhänge auf der CD verwiesen.

Kapitel 5

Zusammenfassung und Ausblick

Kapitel 5 soll dem Leser einen kompletten Überblick über die Arbeit bieten. Zu Beginn erfolgt eine Zusammenfassung in der Zielsetzung, Vorgehensweise, Inhalte, Ergebnisse und Schlussfolgerungen der Arbeit in kurzer Form dargestellt werden. Der Ausblick erläutert, welches Vorgehen der Autor für die Umsetzung des Konzepts bei GSD-CS empfiehlt und wo er weiteres Verbesserungspotential sieht. Im Fazit werden die Inhalte und Ergebnisse der Arbeit diskutiert und die besonderen Herausforderungen geschildert.

5.1 Zusammenfassung

Ziel dieser Arbeit war, den Testprozess bei GSD-CS durch die Entwicklung einer strukturierten Vorgehensweise effizienter zu gestalten. Die vorgeschlagenen Maßnahmen sollten mit dem angebrachten Aufwand umsetzbar sein. Am Anfang stand die genaue Analyse des aktuellen Zustandes und Vorgehens. Dazu wurden Interviews mit den am Testen beteiligten Personen durchgeführt. Neben dem Testprozess wurde gleichzeitig der Entwicklungsprozess untersucht, da dieser die Testaktivitäten wesentlich beeinflusst. Diese Untersuchung zeigte, an welchen Stellen Verbesserungspotential vorhanden ist. Daraufhin folgen Vorschläge für Maßnahmen zur Mängelbeseitigung. In Kapitel 4 wurden diese aufgeführten Maßnahmen konkretisiert, der aktuellen Situation angepasst und bewertet.

Das Ergebnis ist eine Erweiterung des aktuellen Vorgehensmodells um weitere Testphasen sowie eine klare Strukturierung in der bisherigen Testphase mit dem Einsatz passender Testtechniken. Dazu wurden die vier Eckpfeiler des strukturierten Testens (Techniken, Organisation, Infrastruktur und Werkzeuge, Phasenmodell) jeder Phase betrachtet. Diese Maßnahmen erlaubten die Abfassung eines konzeptionellen Testhandbuches, welches den Bogen zwischen Theorie und Umsetzung spannt. Das Testhandbuch soll ein Leitfaden für Tester und Entwickler sein und darlegen, welche Techniken und Werkzeuge im Testprozess

eingesetzt werden sollen. Es definiert zugleich - parallel zum Entwicklungsprozess – den vorgeschlagenen Testprozess und dessen Phasen-Ein- und –Ausgangs-Bedingungen.

5.2 Ausblick

In der vorliegenden Arbeit wurde ein mit dem Softwareentwicklungsprozesses verbundener phasenorientierter Testprozess konzipiert. Dadurch wird eine wichtige Grundlage für weitere Maßnahmen bereit gestellt. Das Testhandbuch bedarf noch einiger Erweiterung welche jedoch nur in Zusammenarbeit mit Vertretern der unterschiedlichen Abteilungen zu leisten ist. Die in der Arbeit konzipierten Maßnahmen müssen diskutiert werden und weiter auf die spezifischen Gegebenheiten angepasst werden. Dadurch wird auch das Risiko eines Scheiterns der Einführung stark gesenkt, da damit das Vorantreiben der Umsetzung nicht nur von einer Person getragen wird. Vorbedingung ist jedoch die Entscheidung des Managements diese Änderungen zu unterstützen. Dies bedeutet nicht nur die Bereitstellung von finanziellen Mitteln sondern auch eine entsprechende Aufgabenverteilung. Erst wenn diese Fragen geklärt sind kann das Konzept sinnvoll erweitert und implementiert werden. Anschließend empfiehlt sich folgendes Vorgehen:

- Verfeinern und Konkretisieren des Konzepts in Zusammenarbeit mit Vertretern der beteiligten Rollen (Tester, Testleiter, Management, Programmierer, Anwender)
- Anwendung in einem konkreten Projekt
- Evaluierung der Ergebnisse
- Gegebenenfalls Nachbesserung oder Erweiterung
- Kommunizieren der geplanten Veränderungsmaßnahmen in Abteilungsmeetings

Das aktuell vorliegende konzeptionelle Testhandbuch sollte mit Hilfe dieser Ergebnisse vervollständigt werden. Sollte sich das resultierende Testhandbuch und der Testprozess als Fundament des Testens bewähren, ist es möglich auf deren Basis weitere Verbesserungsmaßnahmen zu planen. Denkbar wäre hier der nächste im V-Modell vorgesehene Schritt der bei GSD-CS nicht ausgeführt wird, nämlich die systematische Ableitung der Testfälle aus den Anforderungen. Dazu existieren verschiedene Modelle zur Anwendungsfallmodellierung (z.B. UML 2) sowie Anforderungsspezifizierung (z.B. SRS). Die Qualität der Software könnte nun auch gezielt unter ausgewählten weiteren Qualitätsmerkmalen, wie zum Beispiel Zuverlässigkeit, untersucht werden. Generell lassen

sich weitere Testaktivitäten durch das Testhandbuch einfach in den Testprozess integrieren. Dadurch stehen die Änderungen direkt allen Mitarbeitern zur Verfügung.

5.3 Fazit

Eine besondere Herausforderung dieser Arbeit war, die Theorie des Testens mit der Akzeptanz der Mitarbeiter, der Art von entwickelter Software, und der Vorgabe von einfach umsetzbaren Maßnahmen zu verbinden. Es musste verständlich gemacht werden wie die vorgeschlagenen Maßnahmen unterstützen können die Ziele des Testens effizienter zu erreichen. Auch durften die Maßnahmen nicht allein in einem theoretischen Konzept niedergeschrieben werden, da sonst die Chancen für eine Umsetzung sehr gering gewesen wären. Es musste begründet und konkretisiert werden, welche Testaktivitäten zu welchem Zeitpunkt des Softwarelebenszyklusses und mit welchen Werkzeugen und Techniken durchgeführt werden sollen.

Eine weitere Herausforderung ergab sich dadurch, dass die Arbeit in einem Unternehmen durchgeführt wurde. Hierbei mussten Theorie und Praxis gleichermaßen in der Ausarbeitung der Verbesserungsvorschläge miteinander verzahnt werden. Die theoretischen Grundlagen und das daraus resultierende Konzept mussten in jeder Phase anhand der Praxis beurteilt werden.

Durch die Arbeit vor Ort wurde klar, dass das in der Fachtheorie oft empfohlene Vorgehen für die Praxis nicht geeignet sein muss, da die Umsetzung oft zu schwierig oder der Aufwand zu hoch wären. Es müssen zusätzlich zur Theorie noch viele weitere Faktoren wie Psychologie, Trends, Unternehmensphilosophie- und Entwicklung, Vorkenntnis der Mitarbeiter und bisheriges Vorgehen beachtet werden. Deshalb existiert kaum eine theoretische Basis die ohne genau Evaluierung und Anpassung auf die vorgefundenen Gegebenheiten anwendbar ist.

Am Anfang der Arbeit wurde neben verschiedenen Reifegradmodellen das Modell *Test Process Improvement* (TPI) evaluiert. Das so genannte *Assessement* zur Ermittlung der aktuellen Situation sowie die darauf basierende Analyse um Verbesserungspotential zu identifizieren ist zwar einfach durchzuführen, jedoch wurde schnell klar dass die Ergebnisse nicht die erwarteten Ansprüche erfüllten. Die Maßnahmen wären nur mit hohem Aufwand

umzusetzen gewesen und setzen eine Organisation sowie einen Qualitätsanspruch voraus dem viele Unternehmen nicht gerecht werden können. Trotzdem wurden die ersten Ergebnisse dieses Verfahrens in Abteilungsmeetings vorgestellt und diskutiert. Auch hierbei stellte sich heraus, dass die Akzeptanz für die Maßnahmen sehr gering gewesen ist und eine Prozessveränderung nicht durchsetzbar gewesen wäre. Der Autor hat festgestellt, dass die Thematik und die Risiken einer Prozessveränderung im Testumfeld so komplex sind, dass man in unstrukturierten Umgebungen wie bei GSD-CS nur selten Modelle zur Prozessverbesserung einsetzen kann. Ein Modell wie TPI benötigt bereits einen fundamentalen Reifegrad sowie einen sehr hohen Qualitätsanspruch.

Die Gefahr des Scheiterns einer Einführung von einem strukturierten Testprozess ist vergleichsweise hoch, da es viele Risiken gibt die oft nur schwer zu beseitigen sind. Jedoch ist selbst ein passendes, durchdachtes und ausführliches Konzept kein Garant für das Gelingen. Fehlende Akzeptanz, gestrichenes Budget, strategische oder politische Managemententscheidungen können das Scheitern der Einführung eines neuen Prozesses verursachen.

Der Autor ist zuversichtlich, dass die herausgearbeiteten Werkzeuge und die vorgeschlagenen Maßnahmen die oben aufgeführten Ansprüche erfüllen können. Die im Testhandbuch festgehaltenen Maßnahmen haben sich in der Praxis bereits bewährt und lassen sich zudem mit wenig Aufwand umsetzen.

Der Autor verspricht sich von den Vorschlägen zusammenfassend folgende Effekte für GSD-CS: durch *Coding-Standards* eine höhere Codequalität und Verbesserung des individuellen Stils, was die Fehlerträchtigkeit verringert und die Wartbarkeit und Robustheit des Codes erhöht. Durch Komponententests wird eine frühe, günstige Fehlerfindung und Entlastung des Systemtests erreicht. Durch die kontinuierliche Integration ergibt sich eine durchgängige hohe Codequalität im *Repository* sowie konstant verfügbare lauffähige *Builds*. Eine höhere Effizienz und konstante Testqualität durch den Einsatz der richtigen Testmethoden ist ein weiterer positiver Effekt. Dies äußert sich in einem geringeren Fehlerrisiko nach Produktivfreigabe. Zusätzlich erhält der Testprozess eine höhere Transparenz durch die Messbarkeit einzelner Aktivitäten und durch die festgelegte Struktur. Durch diese Effekte erhält der Testprozess die in den Anforderungen geforderten Eigenschaften: Strukturiertheit, Messbarkeit und Effizienz bezogen auf geringeren Aufwand und höhere Testtiefe.

Durch das Testhandbuch wird ein Fundament für die Wissensübermittlung von alten zu neuen Mitarbeitern sowie zwischen den Projekten geschaffen. Zudem schafft es eine praxisnahe Orientierung, die als Fundament für die Einführung der Maßnahmen dienen kann. Der in der Arbeit konzipierte Testprozess bietet zudem eine gute und bewährte Grundlage für weitere Verbesserungen des Testens oder des Softwareentwicklungsprozesses.

Abschließend betrachtet, konnten auch die vorgegebenen nicht-funktionalen Anforderungen an das Konzept erfüllt werden: Einfachheit durch die praxisnahe Formulierung im Testhandbuch und der Auswahl passender Methoden, geringe Kosten durch den Einsatz von einfach zu bedienenden Open-Source-Werkzeugen, Akzeptanz durch Rücksprache mit den beteiligten Personen und regelmäßige Präsentation der Ergebnisse in Meetings und Nachhaltigkeit durch Dokumentation in Form des Testhandbuchs.

Wenn sich der vom Autor von ersten Rückmeldungen erlangte Eindruck bestätigt, werden die vorgeschlagenen Maßnahmen als positiv wahrgenommen, was eine wichtige Vorbedingung für eine gelungene Einführung eines strukturierten Testprozesses ist. Zudem sollte jedoch auf die weiteren in der Arbeit dargestellten Risiken geachtet werden. Präventive Maßnahmen, wie erhöhte Kommunikation und Information zum Beispiel in Meetings erhöhen deutlich die Chancen einer erfolgreichen Einführung.

Letztendlich hatte die vorliegende Arbeit auch den zufriedenen Kunden im Blickfeld und dies nicht nur als „Nebeneffekt“ einer besseren Softwarequalität unter den genannten Anforderungen. Die Ergebnisse und die erwarteten Effekte passen somit in die Strategie der *Group Service Delivery* (GSD) als globale Überorganisation der Abteilung *Customer Solutions.* Thorsten Broese (Leiter der *IT Services*) schreibt auf der Intranetseite der Gruppe: “*The ultimate goal is to satisfy our customers. We want to focus on quality because it inspires employees, instils a culture and an attitude, creates an image in the market and community, and reduces costs in the long term.”*

Abkürzungsverzeichnis

ASP.NET	*Active Server Pages .NET*
BAT	*British American Tobacco*
CS	*Customer Solutions*
CVS	*Concurrent Versions System*
GUI	*Graphical user interface*
GSD	*Global Service Delivery*
IEEE	*Institute of Electrical and Electronics Engineers*
MDX	*Multidimensional Expressions*
UML	*Unified Modelling Language*
PL/SQL	*Procedural Language/Structured Query Language*
SRS	*Software Requirements Specification*
TPI	*Test Process Improvement*
TTS	*Trouble Ticket System*

Literaturverzeichnis

Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000 (ISBN: 0201616416)

Black, R.: *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*, 1. Auflage. Wiley & Sons, Hoboken/NJ, 2007 (ISBN: 0470127902)

Boehm, B.W.: *Software Engineering Economics.* Prentice Hall PTR, Upper Saddle River/NJ,1982 (ISBN 0138221227)

Copeland, L.: *A Practitioner's Guide to Software Test Design.* Artech House Computing Library, Norwood/MA, 2003 (ISBN: 158053791X)

DeMarco, Tom: *Controlling Software Projects: Management, Measurement and Estimation*. Yourdon Press, New York/NY, 1982 (ISBN 0131717111)

Dahl, O.-J., E. W. Dijkstra und C.A.R. Hoare: *Structured Programming*. Academic Press, London, 1972 (ISBN 0122005503)

Dijkstra, E.W.: *The Humble Programmer*. Commun. ACM **15**: 859-866 (1972)

Farooq, A., H. Hegewald und R. R. Dumke: *A Critical Analysis of Testing Maturity Model*. Metrics News. Journal of the Software Metrics Community **12**: 35-40 (2007)

Holzer, B. und T. Mark: *Das TPI-Modell verbessert den Testprozess*. Computerwoche **10** (2007)

Hüttl, R.: *Sittenwächter - Erhöhte Codequalität durch den Einsatz des Analysetools FxCop*. Dot.NET Magazin 4/2004

IEEE Computer Society: *1028-1997 IEEE standard for software reviews*. Software Engineering Standards Committee of the IEEE Computer Society, USA 4. März 1998 (ISBN 1559379871)

Juran, J.M. und F.M. Gryna (Hrsg.): *Juran's Quality Control Handbook*. 4. Auflage. McGraw-Hill, New York, 1988 (ISBN: 0070331766)

Kaner, C., J. Bach, und B. Pettichord: *Lessons Learned in Software Testing: A Context-Driven Approach*. 4. Auflage. Wiley & Sons, Hoboken/NJ, 2002 (ISBN 9780471081128)

Kaner, C., J. Falk und H.Q. Nguyen: *Testing Computer Software.* 2. Auflage. Wiley & Sons, Hoboken/NJ, 1993 (ISBN 0471358460)

Koomen, T. und M. Pol: *Test Process Improvement. A Step-by-step Guide to Structured Testing.* Addison-Wesley, Cambridge/MA, 1999 (ISBN 0201596245)

Pol, M., T. Koomen und A. Spillner: *Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap.* , dpunkt-Verlag, Heidelberg, 2002 (ISBN 3898641562)

Royce, W.W.: *Managing the development of large software systems.* In: Proceedings of the 9th International Conference on Software Engineering 1987 (IEEE Computer Society), pp.328-338. IEEE Computer Society Press, Los Alamitos/CA 1987

Spillner, A. und T. Linz: *Basiswissen Softwaretest*. 3. Auflage. dpunkt-Verlag, Heidelberg, 2005 (ISBN 3898643581)

Veenendaal, E. van, und M. Pol (Hrsg.): *A Test Management approach for structured testing. Archiving Software Product Qualitity*, UTN Publishers, Den Bosch, Netherlands, 1997

Veenendaal, E. van, und Ron Swinkels: *Guidelines for Testing Maturity*. Professional Tester **3**: 1-5 (2002)

Walter, W. und I. Wünsche: *Einführung in die moderne Kostenrechnung: Grundlagen, Methoden, neue Ansätze* Gabler Verlag, 2005 (ISBN: 3409322469)

Whittaker, J.A.: *What is software testing? And why is it so hard?* IEEE Software Magazine **17**: 70-79 (2000)

Wiener, L.R.: *Digitales Verhängnis, Gefahren der Abhängigkeit von Computern und Programmen*. Addison-Wesley, München, 1994

Young, M. und R. N. Taylor: *Rethinking the taxonomy of fault detection techniques.* In: Proceedings of the 11th International Conference on Software Engineering 1989 (IEEE Computer Society), pp. 53–62. ACM Press, New York/NY, 1989 (ISBN 0818619414)

Zhu, H., A.V. Bach und J.H.R. May: *Software unit test coverage and adequacy.* ACM Computing Surveys **29**: 366-427 (1997)

Internetquellen

Ambler, S: *Why Agile Software Development Techniques Work*
http://www.ambysoft.com/essays/whyAgileWorksFeedback.html (03.08.2008)

BAT, British American Tobacco Internetauftritt: http://www.bat.com (20.05.2008)

Gupta, S. C.: *What lies beneath - Discovering untested code* IBM Developer Works, 2006. - http://www.ibm.com/developerworks/rational/library/06/0124_gupta/ (22.08.2008)

GSD, Group Service Delivery Intranetseite (05.07.2008)

IIT, Illinois Institute of Technologie, Computer Science: *Testing Maturity Model Project* - http://www.cs.iit.edu/research/tmm.html (20.04.2008)

ISTQB: *Standard Glossary of Terms used in Software Testing V1.3,* 31. Mai 2007. - http://www.istqb.org (30.4.2008)

BS, British Computer Society - Specialist Interest Group in Software Testing: *Software Component Testing Standard,* Working Draft 3.4, April 2001
http://www.testingstandards.co.uk/bs_7925-2.htm (08.04.2008)

Schluff,S : *Continuous Integration mit CruiseControl* , Orientation in Objects GmbH -
http://www.oio.de/cruisecontrol.htm (03.06.2008)

V-Modell XT, *Dokumentation V-Modell XT Version 1.2*, Bundesrepublik Deutschland 2004
http://www.kbst.bund.de/ (24.8.2008)

Anhang

Anhang A: Fragenkatalog

Anhang B: Beispielkapitel Testhandbuch: Integrationstest

Anhang A: Fragenkatalog

Folgender Fragenkatalog wurde im Rahmen der Untersuchung verwendet. Dabei wurde, wie in Kapitel 2.7.4 beschrieben, ein Leitfadeninterview mit offenen und geschlossenen Fragen verwendet. Auf den Interviewverlauf wurde flexibel reagiert. Manche Fragen wurden gestrichen und andere spontan hinzugefügt. Es sollte ein besonders genaues Bild der Situation entstehen, vor allem unter dem Aspekt, welche Maßnahmen akzeptiert werden könnten.

Projekte

- Mit und an welcher Art von Softwaresystemen arbeiten Sie?
- In welcher Art von Projekten sind Sie hauptsächlich involviert?
- Wie würden Sie ihre Rolle bzw. Tätigkeit dabei beschreiben?

Softwareentwicklung

- Gibt es ein (einheitliches) Vorgehensmodell?
- Gibt es Artefakte z.B. in Form von Dokumenten nach jeder Phase?
- In welcher Form werden die Anforderungen (Requirements) erhoben? Woher kommen diese?
- In welcher Form werden Programmteile vor der Implementierung geplant?

Testen

- Wie werden hier normalerweise Tests durchgeführt?
- Gibt es einheitliche Testprozesse oder definierte Phasen?
- Welche Rolle nehmen Sie dort ein?
- Inwiefern sind hier die Grundkonzepte des Testens bekannt?
  - Gibt es eine einheitliche Begriffswelt?
- Gab es Schulungen/Seminare zu Softwarequalität- oder Testen?
- Wieviel Zeit wird für das Testen prozentual im Vergleich zu der Entwicklung aufgewendet?

- Gibt es einen Unterschied zwischen dem Testen von neu entwickeltem Code und der Überprüfung eines behobenen Fehlers?
- Gibt es oft Seiteneffekte oder Folgefehler einer Änderung?
- Könnte man Ihrer Meinung nach mit einem strukturierteren Testvorgehen etwas verbessern?
- Wann ist der erste Zeitpunkt in einem Entwicklungsprozess in dem getestet wird?
- Werden Aufwandschätzungen bzgl. Kosten und Zeit beim Testen betrieben?
- Wird das Testen in irgend einer Form mit Metriken bewertet?
- Werden Werkzeuge für die Defektadministration oder weitere mit dem Testen in Bezug stehende Werkzeuge eingesetzt?
- Gibt es eine definierte Testumgebung? Wer trägt die Verantwortung dafür?
- Gibt es einen speziellen Arbeitsplatz für das Testen? Wie sieht dieser aus?
- Gibt es festgelegte Rollen für das Testen, sprich z.B. Testmanager und Tester?
- Wie werden die Ergebnisse der Tests kommuniziert?
- Wird vor dem Testen ein Testplan formuliert, der besagt was wie wann und von wem getestet werden soll? Wird dieser Testplan während des Testens verifiziert?

Anhang B: Beispielkapitel Testhandbuch - Integrationstest

3.2 Integrationstest

Der Kompontentest bildet die Grundlage für den nachfolgenden Integrationstest. Die zuvor getesteten einzelnen Module werden nun zu einem System zusammengeschlossen um Integrationsprobleme aufzudecken. Dies soll automatisch mit der Umgebung CruiseControl.NET geschehen. Abbildung 4 zeigt den Arbeitsablauf.

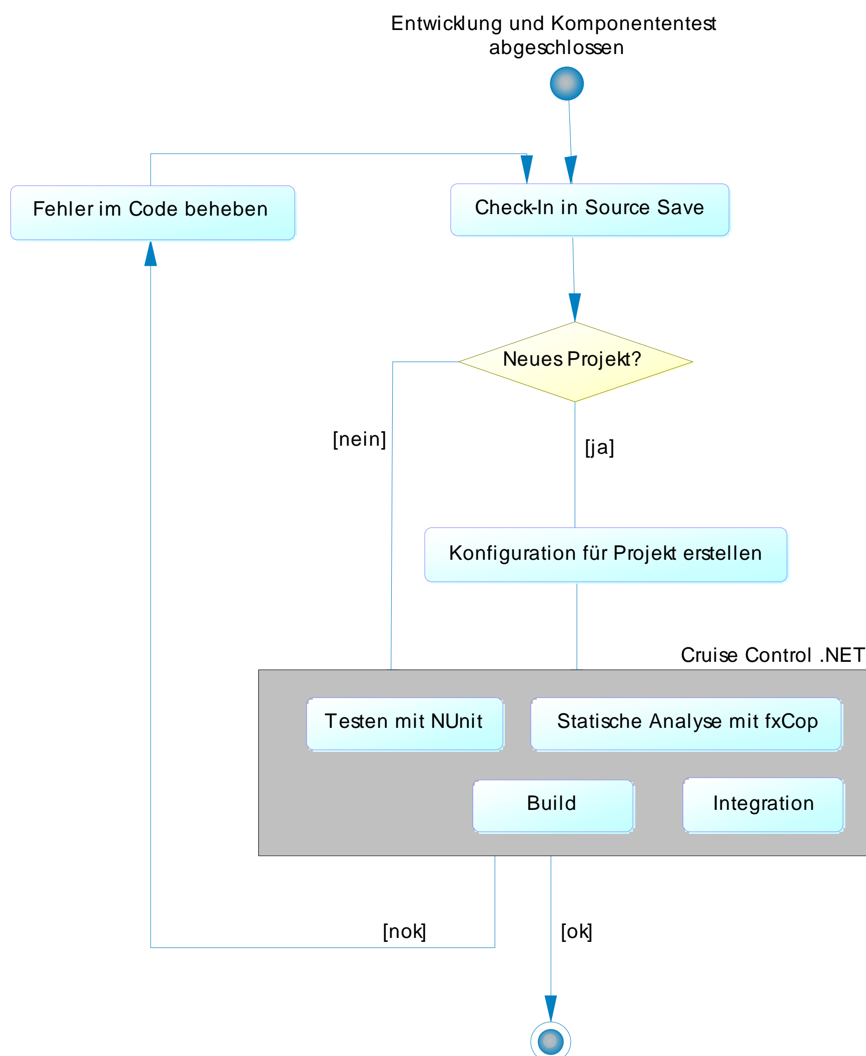


Abbildung 4: Arbeitsablauf Integrationstest

Nachdem ein neues Projekt in das Repository (*Microsoft SourceSave*) eingecheckt ist, muss das Projekt für CruiseControl.NET konfiguriert werden. (Siehe 3.2.4)

Der Integrationsserver führt anschließend ereignissgesteuert einen Build sowie die statische Analyse mit fxCop und den Komponententest mit NUnit durch. Der Entwickler muss nun auf dem Dashboard überprüfen, ob die Tests positiv abgeschlossen sind. Dadurch ist gewährleistet, dass die Software buildfähig ist, die Komponenten gemäß ihrer technischen Spezifikation funktionieren sowie die Codequalität sowie der Programmierstil einem gewissen Standard genügt. Die Aussagekraft des Komponententests hängt jedoch von der Qualität der Testfälle ab. Hier für muss in der vorherigen Teststufe Komponententest gesorgt werden.

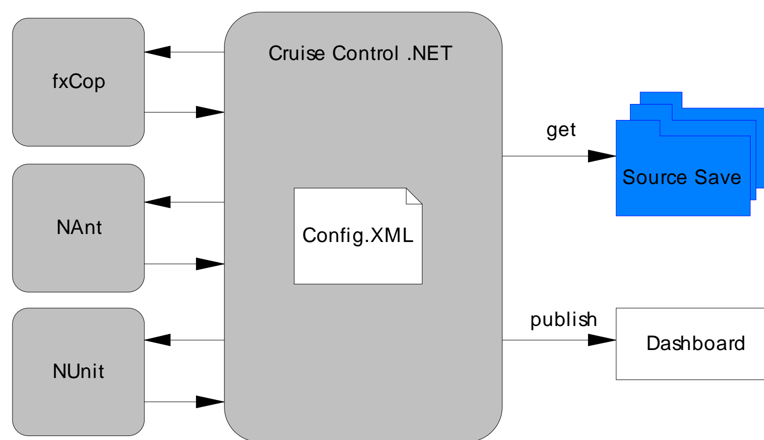


Abbildung 5: Interaktion CruiseControl.NET

3.2.1 Eingangsbedingung

Der Komponententest ist hinlänglich abgeschlossen und es exisitiert eine buildfähige Version.

3.2.2 Organisation

Organisatorisch gibt es beim Integrationstest nur die Rollenverteilung: Testleiter und Tester ist hier der Entwickler.

3.2.3 Infrastruktur und Werkzeuge

- Der Integrationstest findet mit CruiseControl.NET statt.
- Zu dem Testobjekt muss die passende Konfigurationsdatei vorhanden sein oder erstellt werden
- Ein Repository Server muss aufgesetzt sein (Microsoft Source Save)
- Der Integrationsserver mit CruiseControl.NET muss vorhanden sein.

- NAnt [web NAnt] als Buildwerkzeug muss konfiguriert sein
- NUnit und fxCop sollten vorhanden und benutzbar sein, was aber schon im Komponententest festgelegt ist.

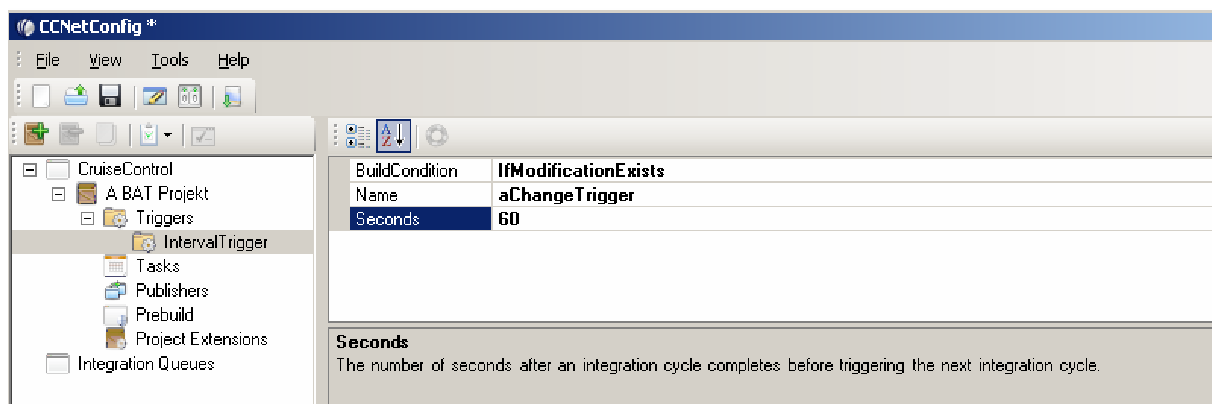
3.2.4 Techniken

Da wir hier eine automatische Integration durchführen, ist an Techniken nur die Bedienung der Umgebung CruiseControl.NET wichtig. Vorbedingung für eine Durchführung der Teststufe Integrationstest ist ein funktionsfähige Instanz dieser Umgebung somit beschäftigt sich dieser Absatz nur mit der Konfiguration eines einzelnen Projektes.

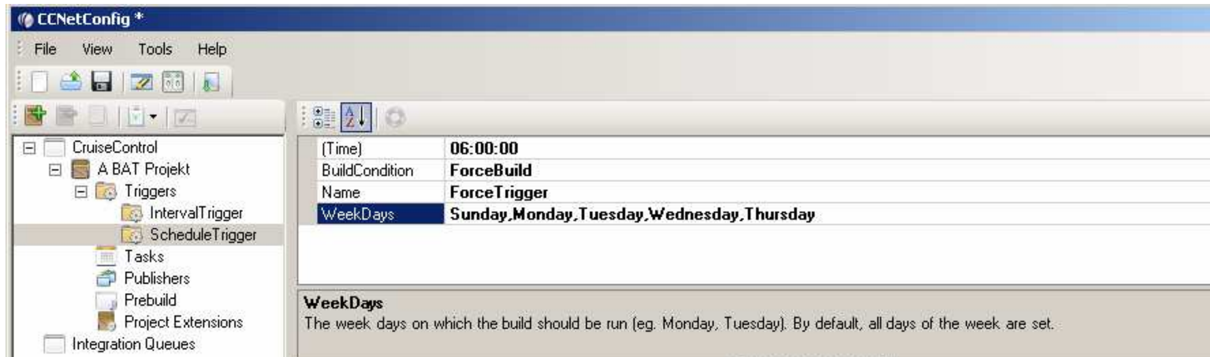
Für jedes neue Projekt muss eine XML-Datei angelegt werden, die beschreibt wo der Compiler, der Repositoryserver, NUnit, fxCop, und die Solution liegen. Diese kann manuell angelegt werden. Hier ist auf die umfangreiche Dokumentation unter [web CC.NET]
oder die zahlreichen How-To's im Internet zu verweisen.

Konfortabler geht die Konfiguration über das Werkzeug CCNetConfig , mit dem man mit Hilfe einer grafischen Oberfläche die Einstellungen vornehmen kann. [web CCCFG]
Damit lassen sich Trigger, Tasks, und Publishers konfigurieren.

Trigger
Mit Hilfe der Trigger, lässt sich bestimmen wann und zu welchem Event der Integrationsprozess gestartet werden soll. Interessant sind hierbei vor allem die Intervalltrigger und die Scheduletrigger. Der Intervalltrigger legt fest, dass in einem bestimmen Zeitintervall das Repository auf Änderungen oder andere Events überprüft wird:



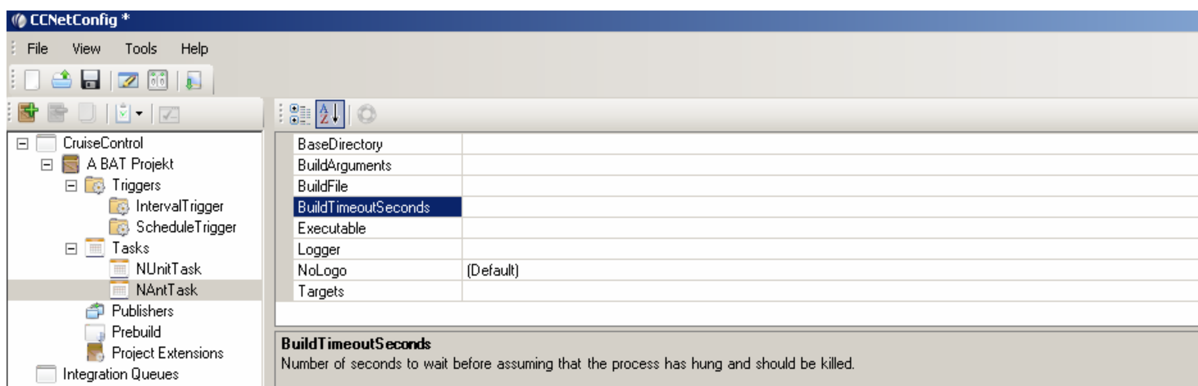
Der Scheduletrigger legt den Zeitpunkt fest, an dem der Prozess in jeder Woche gestartet wird.



Im Beispiel jeden Tag außer Freitag und Samstag um 6:00 Uhr.

Tasks

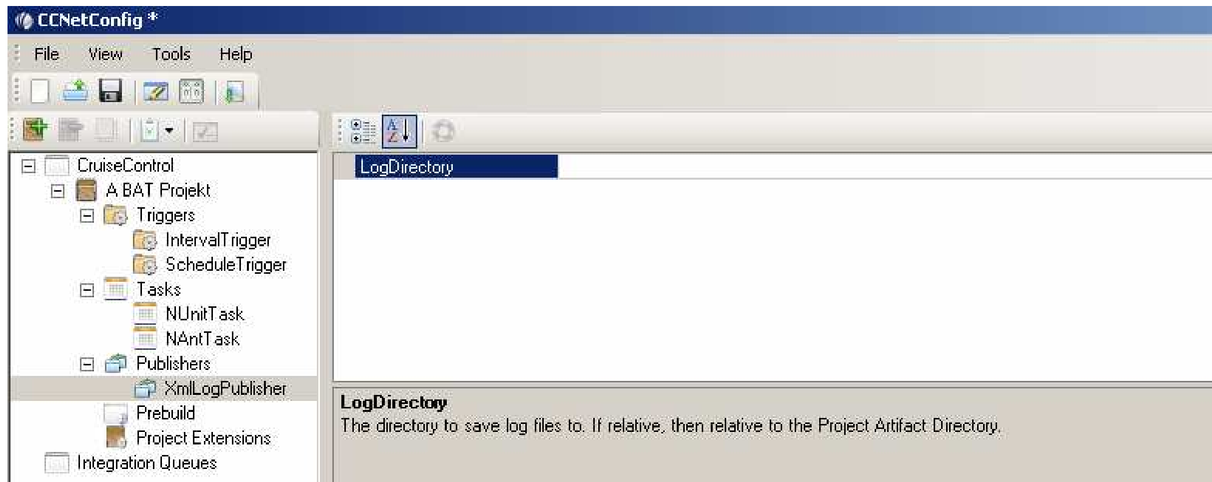
In den Tasks wird beschrieben, welche Aktionen durchgeführt werden sollen. Hierbei sind vor allem der NUnit-Task und der NAnt-Task (Compiler) wichtig:



Im NAnt Task wird unter anderem auch die Verwendung von fxCop eingestellt. Eine genaue Anleitung findet man dazu unter *Run FxCop as part of your integration* [web: CC.NET]

Publishers

Mit den Publishern wird der zu liefernde Output von CruiseControl.NET festgelegt. Da wir das Dashboard nutzen wollen, müssen wir das Ergebnis des Build- und Testprozesses als XMLLog festhalten. Also wird ein XMLLog-Publisher angelegt.



Damit ist die Konfiguration abgeschlossen und die Integration sollte nun nach Eintreten des Events des IntervalTriggers automatisch ablaufen.

3.2.5 Phasen

Vorbereitung

Falls das Projekt noch nicht in CruiseControl.NET angelegt ist, muss die XML-Konfigurationsdatei für das Projekt angelegt werden. (Schritt 0.) (Siehe Techniken 3.5.4)

Durchführung

Die Projekte werden vom Integrationsserver abgerufen und die Integration findet statt. Das Ergebnis wird auf dem Dashboard dargestellt (Abbildung 3, Schritt 2+3). Schritt 2 wird entweder zeitgesteuert gestartet, oder falls neue Versionen in den Repository Server eingecheckt werden (Abbildung 3, Schritt 2).

Abschluss

Das Ergebniss des Tests wird über das Dashboard überprüft (Abbildung 3, Schritt 4).

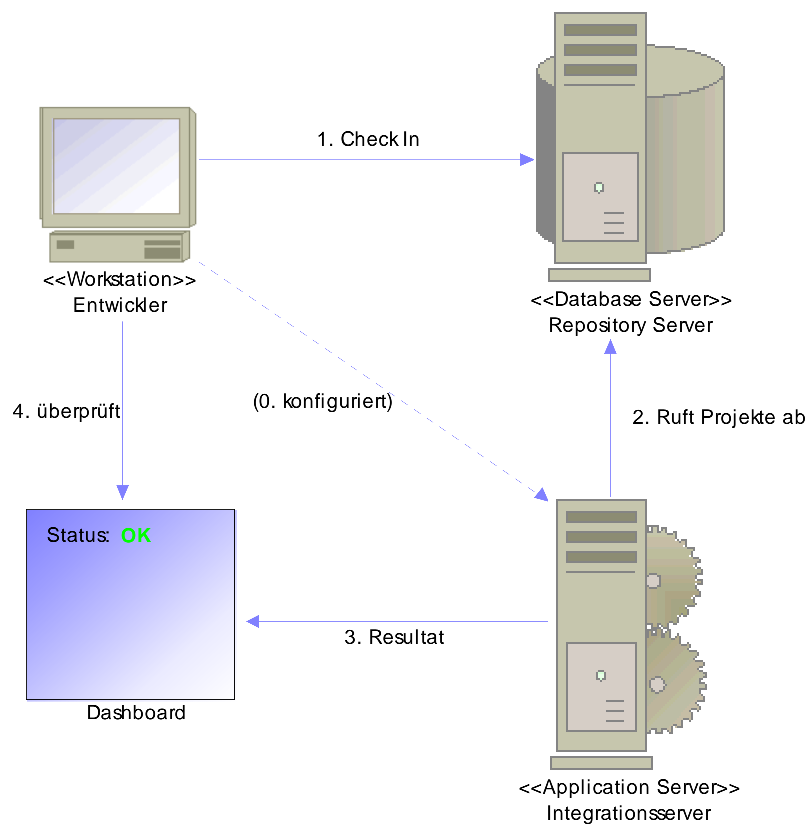


Abbildung 5: Interaktion zwischen Repository, Entwickler, und Integrationsserver

3.2.6 Ausgangsbedingung

Der Integrationstest weist bei der Überprüfung auf dem Dashboard keine Fehler auf.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Ich erkläre mich damit einverstanden, dass ein Exemplar meiner Bachelorarbeit in die Bibliothek des Fachbereichs aufgenommen wird; Rechte Dritter werden dadurch nicht verletzt.

Hamburg, den ………………. ……………………………

Daniel Löffelholz