AP 3.1: Effektivität von QS-Maßnahmen Stand der Wissenschaft

Autoren:
Michael Kläs (FhG IESE)
Dr. Taras Beletski (FhG IESE)
Dr. Alex Sarishvili (FhG ITWM)

IESE-Report Nr. 096.07/D
Version 1.0
30. Juni, 2007

Eine Publikation des Fraunhofer IESE

Das Verbundprojekt TestBalance wird vom Bundesministerium für Bildung und Forschung (BMBF) gefördert. Das Förderkennzeichen ist 01 IS F08 D. Die Partner des Verbundprojektes sind die Organisationen:

Fraunhofer IESE (Kaiserslautern),
Fraunhofer ITWM (Kaiserslautern),
imbus AG (Möhrendorf),
SAP AG (Walldorf) und
T-Mobile International AG.

Die Trägerschaft des Projektes liegt bei der Deutschen Gesellschaft für Luft- und Raumfahrt (DLR).

Das Fraunhofer IESE ist ein Institut der Fraunhofer-Gesellschaft.
Das Institut transferiert innovative Software-Entwicklungstechniken, -Methoden und -Werkzeuge in die industrielle Praxis. Es hilft Unternehmen, bedarfsgerechte Software-Kompetenzen aufzubauen und eine wettbewerbsfähige Marktposition zu erlangen.

Das Fraunhofer IESE steht unter der Leitung von
Prof. Dr. Dieter Rombach (geschäftsführend)
Prof. Dr. Peter Liggesmeyer
Fraunhofer-Platz 1
67663 Kaiserslautern

Abstract

Dieses Dokument ist im Rahmen des Arbeitspakets 3 – QS-Effektivität des öffentlichen Projekts TestBalance entstanden. Es beschreibt den Stand der Wissenschaft einerseits hinsichtlich der Effektivität von Qualitätssicherungsmaßnahme, andererseits hinsichtlich Techniken zur Effizienzmodellierung und Vorhersage. Dabei trägt er die aktuellen wissenschaftlichen Ergebnisse zu Effektivität und Effizienz von Inspektionen und Testtechniken zusammen und betrachtet die in den Studien unterschiedenen Fehlerarten. Zudem werden in der Praxis genutzte Klassifikationsverfahren für Fehler vorgestellt und ihre Einsetzbarkeit im TestBalance Ansatz untersucht. Abschließend wird zwei Arten von Vorhersagemodell vorgestellt, einerseits Modell die während der Planungsphase vor der Ausführung einer QS-Maßnahme eingesetzt werden können, andererseits Modelle die es ermöglichen QS-Maßnahmen zu kontrollieren und zu steuern.

Schlagworte: TestBalance, quality assurance, estimation, performance, effectiveness

Inhaltsverzeichnis

1 Einleitung

Das Ziel von TestBalance ist eine wirtschaftlich sinnvolle Lösung für ein Projekt- und QS-Manager zu liefern, d.h. beispielsweise die Gesamtkosten des Projekts minimiert und gleichzeitig die Randbedingungen die durch Anforderungen, Ressourcen und Umgebung vorgegeben sind zu erfüllen. Hierfür sollte die Test-Balance-Lösung eine dem wirtschaftlichen Optimum entsprechende Prüfintensität ermitteln können. Das Hauptziel „Nutzen-Kosten Optimierung von QS-Prozessen“ wird auf dabei auf 4 Teilprobleme herunter gebrochen, die in Abbildung 1 dargestellt sind.

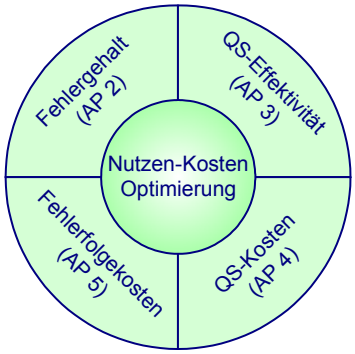

Abbildung 1 Die 4 Teilprobleme des TestBalance-Projekts

Dieses Dokument beschreibt der Stand der Wissenschaft zur Effektivität einer Qualitätssicherungsmaßnahme.

1.1 Definition QS-Effektivität

Generell wird die Effektivität einer Qualitätssicherungsmaßnahme als der Anteil der durch die Qualitätssicherungstechnik gefundenen Fehler von allen findbaren Fehlern definiert. Die Anzahl aller findbaren Fehler kann dabei alternativ definieren werden als (1) Anzahl im Produkt vor der Durchführung der Qualitätssicherungsmaßnahme befindendlichen Fehler *oder* (2) Summe der Anzahl durch Qualitätssicherungstechnik gefundenen Fehler und Anzahl der Restfehler nach Durchführung der Qualitätssicherungstechnik.

Die Abbildung 2 verdeutlich die Definition der Effektivität einer Qualitätssicherungsmaßnahme:

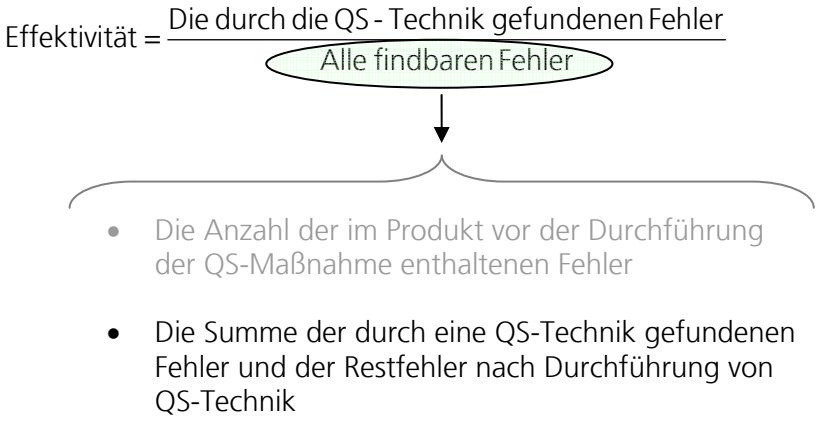

Abbildung 2 Definition der QS-Effektivität

Um die Berechnung von Effektivität von Qualitätssicherungsmaßnahmen zu erleichtern ist es notwendig einige Annahmen über den Fehlerkorrekturprozess zu treffen.

Annahme 1: Es wird angenommen, dass während der Fehlerkorrektur keine neuen Fehler in den Code eingefügt werden.

In einem Projekt wird eine Menge von Qualitätssicherungsmaßnahmen verwendet, wodurch es Abhängigkeiten zwischen den Modellparametern der einzelnen Qualitätssicherungsmaßnahmen gibt.

Definition 1: Wir nehmen an, dass die Reihenfolge der durchgeführten QS-Techniken $T_1, \ldots, T_n$ bekannt ist. Seien F_i die Anzahl der durch QS-Technik T_i gefundenen Fehler und R_i die Anzahl der Restfehler nach Durchführung von QS-Technik T_i, wobei $1 \leq i \leq n$, dann ist die Effektivität von QS-Technik T_i gegeben durch

$$\frac{F_i}{F_i + R_i}.$$

Bemerkung 1: Die Summe der durch die nachfolgenden QS-Techniken $T_{i+1}, \ldots, T_n$ gefundenen Fehler ist dabei kleiner gleich der Anzahl der Restfehler R_i, d.h.

$$R_i \geq \sum_{k=i+1}^{n} F_k.$$

Eine Menge aller findbaren Fehler kann dabei unterschiedlich bestimmt werden. Oft werden dabei zwei Annahmen getroffen:

Annahme A: Es wird angenommen, dass in späteren Phasen gefundene Fehler schon für die QS-Techniken in früheren Phasen findbar waren.

Annahme B: Es wird angenommen, dass nach bestimmtem Zeitpunkt im Wirkbetrieb keine Fehler gefunden werden.

Mit der Annahme A unterschätzt man die Effektivität der QS-Maßnahme in einer Phase. Andererseits wird durch die Annahme B die QS-Effektivität überschätzt. Die Hoffnung ist dass sich die Fehler beider Annahmen gegenseitig aufheben. Die folgende Tabelle beleuchtet diese Aspekt [Hetzel, 1993]. Zudem lässt sich mit der Hilfe dieser Tabelle erkennen, dass die QS-Effektivität von der Reihenfolge der QS-Techniken abhängig ist.

					Walkthrough		*Design Inspections*		*Code Inspections*		*Unit Tests*		*System Tests*	
Step	High Level Design	Low Level Design	Code	Total	Direct	*Age and SRE Adjusted*	Direct	*Age and SRE Adjusted*	Direct	*Age and SRE Adjusted*	Direct	*Age and SRE Adjusted*	Direct	*Age and SRE Adjusted*
Design Walkthrough	40			40	40	*40*								
Design Inspections	9	82		91			91	*91*						
Code Inspections	1	31	66	98					98	*98*				
Unit Tests	0	20	48	68	290	*20*	199	*79*	101	*124*	68	*68*		
System Tests	5	7	11	23							33	*56*	23	*23*
6 Month Operations	2	5	3	10									10	*33*
After 6 Month Operation (Software Reliability Engineering)	3	5	15	23										
Effectiveness					**12%**	***67%***	**31%**	***54%***	**49%**	***44%***	**67%**	***55%***	**70%**	***41%***

Tabelle 1 Berechnung der QS-Effektivität

Tabelle 1 stellt dar, wie sich aus der Anzahl Fehler, die pro Phase eingeführt wurden (Spalte) und pro QS-Aktivität gefunden wurden, die Effektivität der einzelnen QS-Aktivitäten bestimmen ergibt.

- Dabei wird einmal die Effektivität als Anzahl der in der QS-Aktivität gefunden Fehler im Verhältnis zur Gesamtfehlerzahl betrachtet (Direct).
- Zum andern wird die Effektivität als Verhältnis der durch die QS-Aktivität gefunden Fehler zu den zum Zeitpunkt der Durchführung im Produkt vorhandenen Fehlern bestimmt (Age & SRE Adjusted).

Hierbei können sich abhängig von der Effektivitätsdefinition deutlich unterschiedliche Effektivitäten für die einzelnen QS-Aktivitäten ergeben.

2 Einordnung der QS-Effektivität in ein Rahmenmodell

Wie schon zuvor erwähnt handelt es sich bei der Bestimmung der QS-Effektivität nur um eines der 4 Teilprobleme, die nach ihrer Lösung gemeinsam das Gesamtproblem „Nutzen-Kosten Optimierung von QS-Prozessen" lösen sollen. Im nachfolgenden wird ein mögliches Modell zur Integration der TestBalance Teillösungen vorgestellt und die QS-Effektivität in dieses eingeordnet.

2.1 Das Softwarekostenmodell von Pham

Dem in [Pham, 2000] vorgestellten Software Kostenmodell liegen folgende Annahmen zugrunde:

- Fehlerdebuggingkosten sind niedriger in der Entwicklungsphase als in der operationeller Phase
- Fehlerkorrekturkosten in der Debbugingphase sind konstant
- Fehlerkorrekturkosten in der operationeller Phase sind konstant
- Es existieren verschiedene Arten von Fehlern, die unterschiedliche Korrekturkosten besitzen (z.B. Kritisch, Major, Minor).

In den Formelen wird dabei die folgende Notation verwendet:

T Zeitpunkt des Software Release

C_{i1} Fehlerkorrekturkosten einer Fehlers vom Typ i während der Testphase

C_{i2} Fehlerkorrekturkosten einer Fehlers vom Typ i während der operationellen Phase

C_3 Testkosten in einer Zeiteinheit

$E(T)$ Erwartete Softwarekosten

R_0 Spezifizierte Softwarezuverlässigkeit

Angenommen *t* sei die Zufallsvariable, welche die „Software Lebensspanne" repräsentiert und *g(t)* die Wahrscheinlichkeitsverteilungs-Dichtefunktion dieser Zufallsvariable, dann ist sind die erwarteten Softwarekosten (Ohne Fehlerfolgekosten, die aus Garantieansprüchen resultiert) nach [Pham, 2000]:

$$E(T) = \int_0^T \left[C_3 t + \sum_{i=1}^{3} C_{i1} m_i(t) \right] g(t)dt + \\ + \int_T^{\infty} \left[C_3 T + \sum_{i=1}^{3} C_{i1} m_i(T) + \sum_{i=1}^{3} C_{i2} (m_i(t) - m_i(T)) \right] g(t)dt \quad (1)$$

Wobei m_{it} die Erwartungswertfunktion des hybriden ENHPP (Enhanced Non-Homogeneous Poisson Process) Modells ist, welches im Rahmen der AP2 entwickelt wurde. Die Zuverlässigkeit *R(t)* der Software ist durch die Wahrscheinlichkeit definiert, die für die fehlerfreie Softwarenutzung einen bestimmten Zeitpunkt überschreitet:

$$R(t) = P(T > t) = 1 - F(t) = \int_t^{\infty} f(x)dx$$

Wobei T die Softwareausfallzeit ist, *F(t)* die „Unzuverlässigkeit", und $f(t)$ die Wahrscheinlichkeits-Dichtefunktion eines Zuverlässigkeitswachstumsmodells, z. B. eines ENHPP Modells.

2.2 QS-Effektivität im Modell von Pham

Die Wahrscheinlichkeit, dass ein Fehler innerhalb eines bestimmten Zeitintervalls [t,t+Δt] während des Testprozesses auftritt ist:

$$P(t \le T \le t + \Delta t) = f(t)\Delta t = F(t + \Delta t) - F(t),$$

$$f(t) = \frac{\partial F(t)}{dt}$$

Annahme: Die QS-Effektivität ist proportional zur Fehlerrate. Dies lässt sich in der Schreibweise der Zuverlässigkeitsfunktion wie folgt ausdrücken:

$$P(F_{[t1,t2]}) = \int_{t1}^{t2} f(t)dt = \int_{t1}^{\infty} f(t)dt - \int_{t2}^{\infty} f(t)dt = R(t1) - R(t2)$$

Die Fehlerrate lässt sich dann wie folgt bestimmten:

$$FR = \frac{R(t1) - R(t2)}{(t2 - t1)R(t1)}$$

Die Hazard Funktion ist dabei die Fehlerrate für ein Intervall das gegen Null konvergiert, d.h. die momentane Fehlerrate [Liu, 1995]:

$$h(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)} = \lim_{\Delta t \to 0} \frac{f(t)}{R(t)}$$

Durch die Maximum-Likelihood-Schätzung der Parameter einer Erwartungswertfunktion eines nicht-homogenes Poissonprozesses $m_i(t), \quad i = 1,..,3$ (siehe (1)) schätzt man automatisch auch die Wahrscheinlichkeits-Dichtefunktion des jeweiligen Prozesses und infolgedessen auch die Zuverlässigkeitsfunktion und die Hazard-Funktion. Anhand dieser Funktionen lässt sich auf die Zuverlässigkeit schließen, wie sie im vorherigen Kapitel definiert wurde. Da es sich jedoch um mathematische Funktionen handelt, kann darüber hinaus die Zuverlässigkeit zu jedem Zeitpunkt angegeben werden und nicht nur die Zuverlässigkeit nach Durchführung der QS-Technik.

3 Deskriptive Modelle zur Effektivitätsbestimmung

3.1 Deskriptiven Modellierung

Die Lösungsanforderungen des TestBalance-Ansatz verlangen die Unterscheidung zwischen einer Vielzahl von QS-Effektivitäten. Im Besonderen ist die Effektivität bezogen auf eine bestimmte Fehlerklasse (LA59) und hinsichtlich eines Konstruktionselements oder einer Systemkomponente (LA60) zu bestimmen. Zudem soll neben der QS-Effektivität einzelner QS-Maßnahmen (LA61) auch die QS-Effektivität des Gesamt-QS-Prozess (LA62) betrachtet werden. Hierbei ist unter einem QS-Prozess eine Kombination von QS-Maßnahmen zu verstehen.

Um diese unterschiedlichen Sichten auf das Konstrukt QS-Effektivität mit Hilfe eines deskriptiven Modells zu modellieren ist es Notwenig für einem Fehlern über seine reine Existenz hinausgehende Information zu erfassen, d.h. Fehler müssen hinsichtlich bestimmter Eigenschaften klassifiziert werden.

Die Notwenigkeit zwischen unterschiedlichen Arten von Fehlverhalten (und den mit ihnen in Zusammenhang stehenden Fehlern) zu differenzieren ergibt sich zudem aus den TestBalance-Anforderungen LA23, LA34. Diese fordern eine Unterscheidung von QS-Maßnahmen bzw. der durch sie gefundenen Fehler hinsichtlich der Qualitäts-Eigenschaften des Produktes, die sie beeinflussen. Insbesondere die Bestimmung der erreichen Qualitätszielen (LA67) macht die Erfassung der durch einen Fehler beeinträchtigten Produktqualität notwendig.

Hieraus ergeben sich die zusammengefasst die folgenden Anforderungen an ein Fehler-Klassifikationsschema:

- Um die Effektivität einer QS-Aktivität beurteilen zu können, muss die Anzahl der durch die Aktivität gefundenen Fehler bekannt sein. Das Schema muss daher die Bestimmung der Aktivität (d.h., im Allgemeinen der QS-Maßnahme) ermöglichen, die einen bestimmten Fehler gefunden hat (kurz, *Fehlerfindungsaktivität*), bzw. ob dieser erst im Feld (durch den Kunden) entdeckt wurde.

- Um Beurteilen zu können wie Effizient eine QS-Aktivität ist, muss bekannt sein, ob sich zum Zeitpunkt der Durchführung der QS-Aktivität, der jeweilige Fehler schon im zu prüfenden Produkt befand. Hierzu muss das Schema die Bestimmung der *Fehlerquelle* zulassen.

- Fehler müssen einer bestimmten *Fehlerklasse* zugeordnet werden können. Diese Fehlerklassen sollten möglichst in Beziehung zu bestimmten Arten von QS-Maßnahmen stehen, d.h. gewisse QS-Maßnahmen finden Fehler einer bestimmten Klasse besser oder weniger gut; oder in Beziehung zu den Fehlerfolgekosten, d.h. Fehler bestimmter Klasse erzeugen höhere Fehlerfolgekosten als Fehler anderer Klassen.

- Fehler/Fehlverhalten sollten darüber hinaus hinsichtlich ihres *Ursprungsort* klassifiziert werden können, beispielsweise in welcher Systemkomponente sie entdeckt wurden bzw. aufgetreten sind.

- Letztlich sollten Fehlverhalten/Fehler noch hinsichtlich der durch sie beeinträchtigten *Produktqualität* klassifiziert werden.

Daher stellen wir uns nachfolgend bereits existierende Klassifikationsverfahren vor, die eine Klassifikation von Fehlern hinsichtlich der obigen Kriterien erlauben.

3.2 Fehlerklassifikationsverfahren

In der Literatur existieren zahlreiche vordefinierte Fehlerklassifikationsschemata, die bekanntesten, im Hinblick auf ihre Nennung in Literatur und Information zu ihrer Anwendung in der Industrie sind das *Orthogonal Defect Classificaiton Scheme* [Chillarege, 1992], das *Hewlett-Packard Scheme* [Grady, 1992], und der IEEE Standard zur *Classification for Software Anomalies* [American National Standards Institute, 1995]. Nachfolgend wird untersucht, ob und in wieweit eines dieser Schemata die vom TestBalance Ansatz geforderten Kriterien erfüllt und demnach wieder verwendet werden kann.

Zusätzlich zu diesen drei Schemata mit vordefinierten Attributwerten wird ein am Fraunhofer IESE entwickeltes und in industriellen Projekten erprobtes Verfahren (unter anderem Bosch [Freimut, 2005] und Allianz [Freimut 2000]) zur Entwicklung kontextspezifischer, maßgeschneiderter Fehlerklassifikationsschemata vorgestellt, das *Fehlerstrommodell*.

3.2.1 IBM ODC, Hewlett-Packard und IEEE Schema

Die nachfolgende Tabelle stellt den bei TestBalance benötigten Fehlerklassifikationen, die durch die Fehlerklassifikationsschemata bereitgestellten Fehlerattribute gegenüber.

Aspekt	IBM ODC	HP Schema	IEEE Schema
Fehlerfindungsaktivität	Defect Removal Activities	-/-	Project Activity, Project Phase
Fehlerquelle	-/-	Origin	Source
Fehlerklasse	Defect Type, Qualifier, Triggers	Type, Mode	Type, Resolution
Ursprungsort	*Source*	-/-	*Suspected Cause*
Beeinträchtigte Produktqualität	Impact	-/-	*Type, Product Status*

Fehlerattribute, die in der Tabelle *kursiv* dargestellt sind, erfassen den durch TestBalance abzudeckenden Aspekt nur teilweise. Die unterstrichen Fehlerattribute erscheinen im Vergleich mit den entsprechenden Attributen aus den anderen Klassifikationsschema am besten geeignet für die Verwendung im TestBalance Ansatz und sollten bei der Erstellung eines Klassifikationsschemas daher näher betrachtet werden. Im Anhang ist eine Auflistung dieser Attribute inklusive Beschreibung der einzelnen Attributwerte zu finden.

Wie sich zeigt bei Betrachtung der Tabelle zeigt, deckt keines dieser Schemata alle in den TestBalance Anforderungen geforderten Aspekte ab. Die ODC von IBM enthält beispielsweise keine Klassifizierung für die Fehlerquelle. Das HP Schema ermöglicht weder die Klassifikation des Ursprungsorts noch der Fehlerfindungsaktivität oder der beinträchtigen Produktqualität. Die durch das IEEE Schema angebotenen Klassifizierungen für Ursprungsort und der beeinträchtigen Qualität sind für unsere Zwecke unzureichend.

3.2.2 Das Fehlerstrommodell

Ein Fehlerstrommodell (FSM) ist ein Messinstrument, das die QS-relevanten Eigenschaften von Softwarefehlern auf einer geeigneten Abstraktionsebene erfasst und eine Hilfestellung zu deren Aufbereitung und Interpretation bereitstellt. Hierzu werden Fehlerzahlen und Informationen zu Fehlereigenschaften gesammelt.

Als Basis werden dabei für jeden Fehler die folgenden Informationen erfasst:

Fehlerquelle: Wo wurde der Fehler in den Entwicklungsprozess eingebracht?

Fehlersenke: Wo wurde der Fehler im Entwicklungsprozess gefunden?

Fehlertyp: Was musste korrigiert werden, um den Fehler zu beheben?

Diese können durch weitere Aspekte (Attribute) erweitert werden, wobei erprobte Prinzipien zur Entwicklung von Fehlerklassifikationen zum Einsatz kommen sollten, wie sie in [Freimut, 2001] vorgestellt werden.

Die erfassten Informationen liefern Aussagen über die Qualität des Softwareentwicklungsprozesses im Allgemeinen und über die Effektivität der verwendeten QS-Maßnahmen im Besonderen (z.B. über die Effektivität eingesetzter Inspektionstechniken oder Komponententests bezüglich bestimmter Qualitätseigenschaften oder Fehlertypen).

Der Prozess der Definition und Einführung eines solchen Schemas wird in [Freimut, 2005] anhand einer Fallstudie bei Bosch vorgestellt, er gliedert sich dabei grob in 3 Phasen:

- Entwicklung des Basismodells (Fehlerquelle und –senke)
- Erweiterung zum detaillieren Modell (Erweitung um Attribute)
- Einführung (Training und Pilotierung)

Neben der Bestimmung der QS-Effektivität einzelner Maßnahmen sowie der QS-Effektivität der QS-Strategie erlauben die FSM-Daten Rückschlüsse hinsichtlich Prozessverbesserungs-Potentiale. Ziel dieser Auswertung ist insbesondere die Optimierung der QS-Prozesse und QS-Strategien. Typische Fragen sind dabei:

- **Verweildauer:** Wie lange dauert es von der Fehlerinjektion bis zur Fehlerentdeckung? Wie viel teuerer wurde die Fehlerbehebung hierdurch?
- **Filterwirkung der QS-Prozesse:** Gibt es eine Häufung von Fehlern bzw. Fehlertypen, die durch bestimmte QS-Aktivitäten „hindurchrutschen"? Finden die QS-Aktivität, die Fehlertypen, die sie finden sollten, werden z.B. alle Schnittstellenfehler im Integrationstest entdeckt?
- **Fehlerfolgekosten:** Gibt es bestimmte Fehlerklassen, die im Feld häufig auftreten und / oder hohe Kosten verursachen?

4 Vorhersage für Projektkontrolle und Steuerung

Während der Durchführung der QS-Maßnahmen ist es für den QS-Verantwortlichen beziehungsweise den Projektmanager wichtig zu bestimmen, ob die durchgeführte QS-Maßnahme die vorhergesagte Effektivität erreicht hat beziehungsweise zu welchem Zeitpunkt sie eingestellt werden kann. Bei Inspektionen kann dies die Entscheidung zu einer erneuten Inspektion desselben Artefakts beeinflussen, beim der Durchführung von Tests kann der Zeitpunkt abgeschätzt werden, ab dem die Tests eine hinreichende Anzahl von Fehlern gefunden haben (d.h. eine hinreichende Effektivität erreicht haben).

4.1 Kontrolle und Steuerung von Inspektionen

In der Literatur werden zur Kontrolle und Steuerung von Inspektionen zahlreiche Ansätze beschrieben. Teilweise benötigen die Ansatze historische Daten (z.B. Kontrollcharts), teilweise sind aber auch die während der aktuellen Inspektion gesammelten Daten ausreichend (z.B. Capture-Recapture-Modelle). Im Folgenden stellen wir die unterschiedlichen Klassen von Modellen vor.

4.1.1 Kontrollcharts basierend auf historischen Daten

Basierend auf Messungen aus historischen Projekten, die zum laufenden Projekt ähnlich sind, lassen sich Werte hinsichtlich der erwateten Anzahl von zu findenden Fehlern ableiten [Weller, 1994], [Ebenau, 1994], [Laitenberger, 1999]. Darstellen lassen sich die Zahlen gefundener Fehler in der bisherigen Projekten beispielsweise mit der Hilfe von Boxplots.

Der Nachteil eines solchen Vorgehens ist, dass ein sehr gutes Produkt unnötigerweise zweimal inspiziert würde, da in ihm bei der ersten Inspektion weniger als die erwarteten Fehler gefunden wurde. Zudem würde ein stark fehlerbehaftetes Produkt nach einer schlecht durchgeführten Inspektion nicht erneut inspiziert, da bereits in der ersten Inspektion hinreichend viele Fehler gefunden wurden.

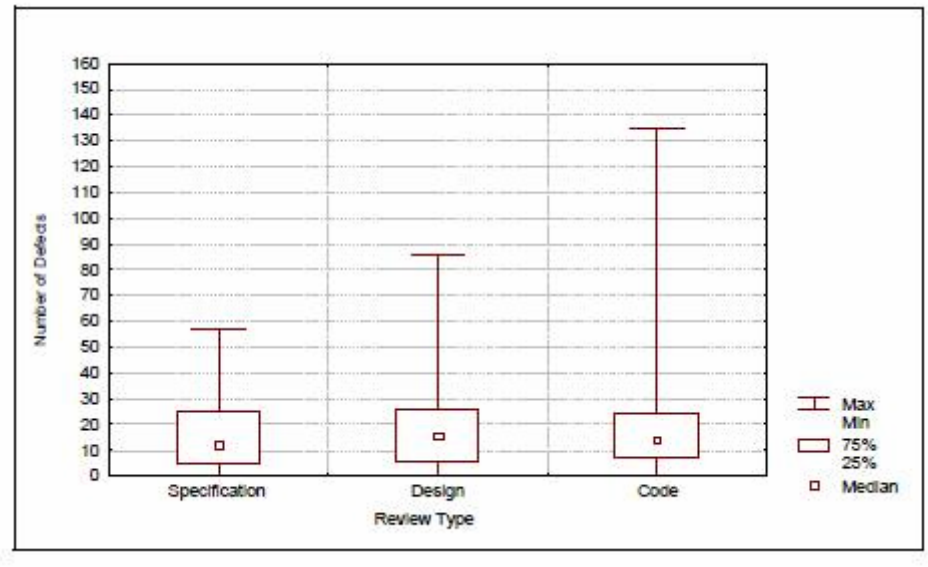

Abbildung 3 Boxplot mit gefundenen Fehlern der Reviews in Phasen aus [Laitenberger, 1999]

4.1.2 Kosten-Nutzen-Modelle

Adam [Adam, 1999] schlägt vor die Entscheidung über eine erneute Inspektion anhand einer Kosten-Nutzen-Abwägung zu treffen. Um diese Entscheidung zu treffen benötigt er Informationen über die in der ersten Inspektion gefundenen Fehler, die veranschlagten Kosten für die Inspektion, die erwartete Effektivität der Inspektion, sowie die durchschnittlichen Kosten, die entstehen wenn ein Fehler erst zu einem späteren Zeitpunkt gefunden wird. Dieser Ansatz erfordert jedoch einen ausgereiften und stabilen Inspektionsprozess, so dass die Effektivität einer Inspektion als weitgehend konstant angenommen werden kann. Dabei wird das in 4.1.1 skizzierte Problem nicht gelöst, aber umgangen, indem von einer konstanten Effektivität ausgegangen wird. Bezüglich Effektivität einer typischen Inspektion sowie Möglichkeiten die Effektivität des Inspektionsprozesses zu stabilisieren verweiß er auf [Gilb, 1996].

4.1.3 Capture-Recapture-Modelle

Capture-Recapture Modelle stammen ursprünglich aus der Biologie. Dort werden sie dazu verwendet, um beispielsweise aus mehreren (unvollständigen) Stichproben auf die Gesamtheit des Bestands zu schließen. Im Software Engineering werden sie verwendet, um aus den durch unterschiedliche Inspektoren gefundenen Fehlern auf die Gesamtheit der im Dokument vorhandenen Fehler zu schließen (Defect Content Estimation). Mit Hilfe dieser Schätzung und der Gesamtzahl der durch die Inspektoren gefundenen (unterschiedlichen) Fehler

kann dann die Effektivität der Inspektion bzw. die erwartete Restfehleranzahl berechnet werden. Die dabei getroffene Annahme ist, dass wenn die Mehrzahl der Fehler durch mehrere Inspektoren gefunden wurde, die Anzahl der unentdeckten Fehler gering ist, wohingegen eine Vielzahl von Fehlern die nur durch einen Inspektor gefunden wurde auf eine hohe Anzahl unentdeckter Fehler hindeutet [Eick, 1992].

Beispiele für die Anwendung und den Vergleich solcher Modelle untereinander sind unter anderem in [Freimut, 1997], [Briand, 1997], [Miller, 1999], [Briand, 2000], [Freimut, 2001] zu finden. Zusammenfassend lässt sich sagen [Petersson, 2004]:

- Die meisten Schätzer *unterschätzen* die Anzahl verbleibender Fehler
- Die besten Vorhersagen liefert Modelle mit *Mh-JK*[1] Schätzer
- Gute Ergebnisse liefern diese *ab 4 Inspektoren*
- Capture-Recapture-Modelle lassen sich auch bei Inspektionen mit Perspektiven-basierten Lesetechniken anwenden

4.1.4 Kurvenanpassungs-Modelle

Eine mit den Capture-Recapture-Modellen verwandte Art von Modellen sind die Kurvenanpassungsmodelle (Curve-Fitting), wie sie in [Wohlin, 1998] für Inspektionen vorgeschlagen werden. Sie haben Ähnlichkeiten zu den Zuverlässigkeitswachstumsmodellen, wie sie in den späteren Phasen des Softwareentwicklungsprozesses beim Testen angewendet werden [Lyu, 1996], [Musa, 1998].

Die gesammelten Fehlerdaten der Inspektoren werden bei dieser Art von Modellen in einem ersten Schritt nach bestimmten Kriterien sortiert (und teilweise aufkumuliert) und in einem Diagram aufgetragen, dann wird in einen gegebene Verteilung (z.B. eine Exponentialverteilung), an die Daten angepasst.

Es wurden unterschiedliche Modelle und Verbesserungen für diese Art der Modelle vorgestellte, jedoch konnte keines der Modelle die Genauigkeit eines Capture-Recapture-Modells mit Mh-JK Schätzer erreichen [Petersson, 2004].

[1] Das Modell Mh geht davon aus, dass Fehler eine unterschiedlich hohe Wahrscheinlichkeit besitzen entdeckt zu werden, als Schätzer wird der Jackknife-Ansatz (JK) verwendet.

4.1.5 Subjektive Schätzungen

In [El Emam, 2000] wird ein Ansatz zur erfahrungsbasierten subjektiven Schätzung der verbleibenden Fehler vorgestellt. Dieser Ansatz wurde von [Biffl, 2000] und [Thelin, 2002] weiter untersucht und hinsichtlich seiner Genauigkeit mit den Capture-Recapture-Modellen verglichen. Dabei stellte Biffl bei subjektiven Schätzungen einen geringeren Bias (Abweichung im Mittel), jedoch eine größere Varianz verglichen mit (quantitativen) CR-Modellen fest. Thelins Ergebnisse hingegen zeigen eine höhere Genauigkeit der CR-Modelle hinsichtlich Punkt- wie auch Konfidenzintervallschätzungen [Petersson, 2004], und somit eine geringere Abweichung im Mittel und eine geringere Varianz.

4.1.6 Fehlerklassifikation

Eine weitere Möglichkeit, die Effektivität einer Design- und Code-Inspektionen zu überprüfen, ist die Verwendung der ODC Trigger und Defect Type Attributen. Der Vergleich der Verteilung der in einer QS-Aktivität gefundenen Fehler bezüglich ihrer Trigger und Defect Types [Chaar, 1993] mit historischen oder erwarteten Verteilungen hilft Abweichungen zu erkennen. Gegebenenfalls kann auf dieser Grundlage eine weite Inspektion angeordnet werden.

4.2 Kontrolle und Steuerung von Testaktivitäten

Normalerweise im Laufe der Testaktivitäten wird relative große Menge an Daten gesammelt, unter anderem Fehlerverlaufsdaten. In der Literatur wird zeitliche Verlauf der Fehlermeldungen meistens mit eine große Klasse von Zuverlässigkeitswachstums-Modellen modelliert. Mit Hilfe von diesen Modellen ist es möglich den Fehlerverlauf zu modellieren, voraussagen und analysieren für Kontroll- und Steuerungszwecken.

4.2.1 Zuverlässigkeitswachstums-Modelle

Die Möglichkeit, das vom Benutzer wahrgenommene Ausfallverhalten der Software über die beobachteten Ausfälle während des Testens zu modellieren und vorherzusagen, wurde in den frühen siebziger Jahren erkannt [Jeske, 2001]. Eine ausführliche Übersicht zu den Software-Zuverlässigkeitsmodellen ist in [Lyu, 1995] enthalten. Eine gute Kurzübersicht liefern [Liggesmeyer, 2002] und [Lyu, 2002].

Software-Zuverlässigkeitsmodelle beschreiben das Zuverlässigkeitswachstum während des Testens. Beim Testen erfährt die Software durch Fehlerkorrektur einen Zuwachs an Zuverlässigkeit. Die unmittelbare und perfekte Behebung der Fehler ist dabei eine zentrale Annahme der Zuverlässigkeitsmodelle. Ebenso wie

die Annahme dass die Software funktional anhand Ihres operationalen Profils getestet wird. Seit den frühen siebziger Jahren wurden zahlreiche Software-Zuverlässigkeitsmodelle entwickelt. [Musa, 1983] beschreibt ein Klassifikationsschema, das Modelle nach den folgenden fünf Attributen klassifiziert:

1. *time domain*: Wie wird die Zeit erfasst? In *calendar time* oder *execution time*?

2. *category*: Haben wir ein Modell mit einer endlichen oder einer unendlichen Anzahl von Fehlern?

3. *type:* Typ der Verteilung der bis zum Zeitpunkt *t* beobachteten Ausfälle

4. *class* (nur bei Modelle mit endlicher Anzahl an Ausfällen): Funktionale Form der Ausfallintensität bzgl. der Zeit

5. *familiy* (nur Modelle mit unendlicher Anzahl an Ausfällen): Funktionale Form der Ausfallintensität bzgl. beobachteten Ausfälle

So unterscheidet die *type*-Klassifikation beispielsweise Binomiale Modelle von den Poisson Modellen, die annehmen, dass das Ausfallverhalten der Software durch einen nicht-homogenen Poisson Prozess (NHPP) beschrieben werden kann. Die *class*-Klassifikation unterscheidet dann innerhalb der Poisson Modelle zwischen Modellen mit einer exponentiellen und einer Weibull Ausfallintensität. Ein bekannter Vertreter der Binomialen Modelle ist das Jelinski-Moranda Modell [Jelinski, 1972] welches asymptotisch äquivalent zu dem NHPP Modell von Goel-Okumoto ist [Goel, 1979]. Genauere Informationen zu jeweiligen Zuverlässigkeitsmodellen können in dem TestBalance Report zu AP2.1 (Stand der Wissenschaft zur Modellierung des Fehlergehalts) nachgelesen werden.

5 Vorhersage für Projektplanung

Zur Planung einer projektspezifischen QS-Strategie ist es notwendig die Effektivität der eingesetzten QS-Maßnahmen zu kennen. Denn nur bei Kenntnis der Effektivität der eingesetzten QS-Maßnahmen lässt sich die Menge und Schwere der zu erwartenden Restfehler bestimmen. Diese Information ist schon zu Projektbeginn notwenig, um die QS-Strategie auf das Projekt auszurichten und damit Aufwand und Ressourcen frühzeitig einzuplanen zu können.

Aus diesem Grund sollte die Testbalance Methode eine Möglichkeit bereitstellen, die Effektivität der eingesetzten QS-Maßnahmen schon vor deren Ausführung zu bestimmen, dass heißt schon zu Projektbeginn eine Vorhersage hinsichtlich der Effektivität der einzelnen QS-Maßnahmen treffen. Die Effektivität einer geplanten QS-Maßnahme ist dabei aber nicht nur von der eingesetzten QS-Technik sondern auch von weiteren Einflussfaktoren (z. B. der Intensität der Prüfung) abhängig.

Nachfolgend betrachtet wird deshalb, welche Einflussfaktoren in der Literatur bekannt und empirisch belegt sind und welche Vorhersagemodelle bisher zur Bestimmung der QS-Effektivität eingesetzt wurden. Dabei unterteilen wir die QS-Maßnahmen in Inspektionen und Testsaktivitäten.

5.1 Effektivitätsvorhersage zur Planung von Inspektionen

5.1.1 Die wichtigsten Einflussfaktoren

Die folgenden Faktoren werden im Allgemeinen als wichtige Einflussgrößen hinsichtlich der Effektivität von Inspektionen genannt und wurden als solche schon in mindestens einer empirischen Studie nachgewiesen:

- **Aufwand** (im besonderen Vorbereitungsaufwand)
  Studien: u.a. [Laitenberger, 2000], [Christenson, 1990]

- **Eigenschaften des untersuchten Produkts** (z.B. Größe, Komplexität)
  Studien: u.a. [Laitenberger, 2002], [Ebenau, 1994], [Christenson, 1990]

- **Größe und Variabilität des Inspektionsteams**
  Studien: u.a. [Laitenberger, 2002]

- **Qualifikation der Inspektoren** (insbesondere Domänenwissen)
  Studien: u.a. [Laitenberger, 2000], [Porter, 1996]

- **Systematik des Inspektionsprozess** (z.B. verwendete Lesetechnik)
  Studien: u.a. [Laitenberger, 1997]

5.1.2 Vorhersagemodell zur Inspektionsplanung

Zur Modellierung und Vorhersage der Effektivität von Inspektionen wurden bisher schon zahlreiche Modellierungstechniken verwendet. Unter anderem wurden Model-Fitting-Methoden [Christenson, 1987], Regressionsanalyse [Ebenau, 1994], Pfadanalyse [Laitenberger, 2002], Diskriminanzanalyse [Yaung, 1994], Neutrale Netzwerke [Raz, 1994] sowie Multiple-Adaptive-Regressions-Splines (MARS) [Freimut, 2004] angewendet.

Die wenigsten der Modelle wurden jedoch zum Zweck der Planung von Inspektionen eingesetzt. Meist wurde vorgeschlagen, sie zur Kontrolle und Steuerung durchgeführter Inspektionen zu verwenden, also zur Entscheidungsfindung, ob eine erneute Inspektion des Produktes notwendig ist. Teilweise wurden sie auch eingesetzt, um die Auswirkungen von Einflussfaktoren zu quantifizieren.

Eine der Ausnahmen ist das Modell von [Briand, 1997c], welches explizit die Planung von Inspektionen unterstützen sollte. Das Modell basierte auf Daten von 150 Inspektionen und einer Lineareren-Regressionsanalyse. Die dabei angenommene Beziehung der gemessenen Größen war:

ln(Fehlerdichte) = a * ln(Vorbereitungsaufwand) + b * ln(Produktgröße)

Eine weitere Modellierungstechnik, die eingesetzt wurde um eine Planungshilfe für Inspektionen bereitzustellen ist MARS. In [Freimut, 2006] konnte anhand von drei Fallstudien gezeigt werden, dass die mit Hilfe des Modells abgeleiteten Planungshilfen sinnvoll waren und die Modellierungstechnik insbesondere bei einer hohen Anzahl von Datenpunkten eine höhere Genauigkeit lieferte als der Einsatz der klassischen Regressionsanalyse.

5.2 Effektivitätsvorhersage zur Planung von Testaktivitäten

5.2.1 Die wichtigsten Einflussfaktoren

In [Andersson, 2006] sind die Ergebnisse mehrer empirischen Untersuchen gesammelt, die teilweisen in der Appendix B beschrieben sind. Bei einigen dieser Arbeiten hat man die Rolle der Einflussfaktoren auf die Effektivität der Qualitätssicherungsmaßnahmen statistisch untersucht. Leider widersprechen sich die

Ergebnisse der Studien teilweise. Man kann jedoch feststellen, dass die folgenden Einflussfaktoren bei einigen Situationen eine große Rolle für die Effektivität einer Qualitätssicherungsmaßnahme spielen:

- Erfahrung mit der Programmiersprache, Testmethode usw.
- Programmierungssprache, Architektur, Programmierungsaufgabe
- Testtechnik selbst
- Persönliche individuelle Unterschiede
- Fehlertyp des Fehlers (Unterschiede zwischen Effektivitätsanalysen verschiedener Fehlertypen)

5.2.2 Vorhersagemodell zur Testplanung

Die wichtigste Klasse von Vorhersagemodellen sind die Zuverlässigkeitswachstums-Modelle (siehe Kapitel 4.2.1). Um diese Modelle anwenden zu können, müssen ihre Modellparameter kalibriert werden. Die am häufigsten angewendete Methode ist dabei die so genannte Maximum-Likelihood-Schätzung. Die Maximum-Likelihood-Methode ist ein Verfahren zur möglichst genauen Schätzung der Populationskennwerte auf Grundlage von Stichprobenwerten. Man versucht, seine Schätzer dabei so zu definieren, dass die Wahrscheinlichkeit, dass eben dieser geschätzte Kennwert die beobachteten Ergebnisse in der Stichprobe verursacht hat, maximiert wird.

Diese Maximum-Likelihood-Schätzung kann dabei entweder auf Daten aus dem aktuellen Projekt basieren oder (während der Planungsphase) auf die historischen Fehlerverlaufsdaten. Bei einem historischen Schätzungsverfahren ist man jedoch gezwungen die Annahme zu treffen, dass im neuen Projekt die gleichen Prozesse zur Anwendung gelangen und zu vergleichbaren Effekten im neuen Projekt führen.

6 Empirische Daten zur QS-Effektivität

6.1 Empirisches Wissen zu Inspektionen

In ihrer Arbeit fassen [Aurum, 2002] Resultate empirischer Untersuchungen von Software Inspektionen aus den letzten 25 Jahren zusammen. Die Autoren betrachten mehr als 30 empirische Studien und die sich unter anderem mit dem Effekt unterschiedlicher Lesetechniken und Team-Größen beschäftigen.

[Elberzhager, 2005] stellt die Resultate von unterschiedenen Untersuchungen hinsichtlich QS-Effektivität vor. Die wichtigsten Ergebnisse können dabei in den folgenden Punkten zusammengefasst werden:

- Systematische Lesetechniken sind effektiver als ad hoc Lesetechnik.
- Die Leistung der (systematischen) Benutzungsbasierten-Lesetechnik kann durch ihre Überarbeitung in zukünftigen Experimenten verbessert werden.
- Generell scheinen Perspektivenbasierte Lesetechniken effizienter als Checklistenbasierte Lesetechniken zu sein. Jedenfalls sollten die Einflussfaktore der besseren Leitung der Perspektivenbasierten Lesetechniken identifiziert werden.
- In meisten Fällen scheint der Einsatz einer Lesetechnik eine effektive Fehlererkennungstechnik zu sein.

6.2 Empirisches Wissen zu Testtechniken

[Juristo, 2002] untersuchen empirische Indizien über der Effektivität unterschiedener Test-Techniken. Die Autoren betrachten dabei mehr als 20 Studien und vergleichen die Test-Techniken sowohl innerhalb einer Familie von Techniken als auch die Technikfamilien miteinander. Die Schussfolgerung der Autoren war, dass unser Wissen über der Effektivität der Test-Techniken trotz der großen Anzahl der Untersuchungen sehr beschränkt ist. Zwei Untersuchungen sollen exemplarisch heraus gegriffen werden.

[Kim, 2000] analysierte die Leistung der Regressionstest-Techniken und stellte fest, dass relative Effektivität (bezieht auf der Anzahl der mit Testsuite entdeckbaren Fehler) maximal für Safe Techniken und retest-all ist, während mehr als 90% Effektivität für Random Testing erreicht wurde. Hinsichtlich der absoluten Effektivität (bezieht auf der Anzahl aller Fehler) stellten sie fest, dass je mehr Fehler im Programm sind, desto uneffektive sind alle der eingesetzten Techniken.

[Basili, 1987] stellte fest, dass erfahrene Benutzer im Vergleich zu strukturbasiertem Testen mehr Fehler mit funktionalen Testtechniken fanden. Für unerfahrene Benutzer gab es in einem Fall keinen Unterschied zwischen der Effektivität der strukturbasierten und funktionalen Testtechniken, und in einem anderen Fall entdeckte man mehr Fehler mit den funktionalen Testtechniken.

Hinsichtlich der unterschiedlichen Arten der gefundenen Fehler entdeckten Grenzwertanalysen (engl. boundary-value analyses; Überprüfung von Randwerten von Äquivalenzklassen) mehr Kontrollflussfehler als eine Überprüfung mit dem Kriterium der Satzüberdeckung, während für andere Fehlerarten keinen Unterschied zwischen diesen Techniken feststellbar waren. Demgegenüber stellten [Kamsties, 1995] fest, dass die Wirksamkeit der Technik nicht von der Wahl zwischen den funktionalen und strukturellen Testtechniken aber vom Programm selbst abhängt. Nach ihrer Ansicht, scheinen funktionalen und strukturellen Testtechniken unterschiedliche Arten von Defekten nicht zu finden.

Für die Testtechniken stellte [Elberzhager, 2005] die folgenden Beobachtungen zusammen:

- Die Regressionstesttechniken scheinen effektiver als „Random Testing" zu sein.
- Testtechniken die auf dem Datenfluss basierenden scheinen effektiver zu sein als Techniken die auf dem Kontrollfluss basieren.

6.3 Effektivität hinsichtlich unterschiedlicher Fehlerarten

In folgenden Unterkapiteln wird die Effektivität einer Qualitätssicherungsmaßnahme in der Abhängigkeit von Fehlertyp betrachtet. Drei Fehlertypen werden betrachtet: Anforderungsfehler, Designfehler und Programmierungsfehler.

6.3.1 Anforderungsfehler

Hinsichtlich Anforderungsfehler scheint die Wahl zwischen Anforderungsinspektionen und Systemtests ziemlich klar: Mehr Aufwand am Anfang des Entwicklungsprozesses zu investieren um einen guten Satzes von Anforderungen herzustellen ist sinnvoller als ein System, das auf falschen Anforderungen basiert zu entwickeln und anschließend zu überarbeiten. Die Fallstudie von [Berling, 2005] stützt diese Annahme.

6.3.2 Designfehler

Hinsichtlich Designfehler vergleicht nur ein Experiment die Fehlerentdeckung mit Design-Inspektionen gegenüber funktionalen Tests. Die Inspektionen waren in diesem Experiment deutlich effektiver und effizienter als die Testtechniken [Andersson, 2003].

Die Resultate dieser Studie sind, dass Effektivität und Effizienz der Inspektionen höher sind und Testtechniken mehr Zeit für Einarbeiten erfordern. Die Variationen zwischen den experimentellen Gruppen waren groß. Mehr als Hälfte der Fehler wurden mit Inspektionen (53,5%) gefunden während etwas weniger beim Testen (41,8%) gefunden wurden. Die Effizienz war bei den Inspektionen 5 Fehler pro Stunde und weniger als 3 Fehler pro Stunde beim Testen. Basierend auf diesen Daten sind Design Inspektionen zu bevorzugen.

Obgleich Aufwände für die Überarbeitung des Designs nicht in Betracht gezogen wurde (d.h. die Studie beinhaltete nur Fehlverhaltenbeobachtung und Fehlerlokalisierung), waren Inspektionen effektiver und effizienter. Ein Fehler, der während der Designinspektionen entdeckt wird, ist preiswerter zu beheben als ein Fehler, der während des funktionalen Testens gefunden wird, da im letzten Fall eine Überarbeitung des Designs und des Codes notwendig ist. Wenn Überarbeitung betrachtet wird, würde der Unterschied bezüglich der Effizienz daher vermutlich noch größer sein, da die Korrektur von Fehlern, die durch Inspektion ermittelt werden, früh im Entwicklungsprozess stattfindet. Logischerweise sollte eine Kombination von Inspektionen und Testen bevorzugt werden, wenn die Techniken unterschiedliche Fehler entdecken.

Die im Experiment erzielen Resultat wurde durch zwei Fallstudien bestätigt [Berling, 2005], [Conradi, 1999].

Die Fallstudie von [Berling, 2005] untersuchte fünf inkrementelle Projektiterationen und ermittelte eine durchschnittliche Fehlerentdeckungs-Effizenz von 0,68 Fehlern pro Stunde für Inspektionen und von 0,10 Fehlern pro Stunde für das Testen, wobei erwähnt werden muss, dass durchgeführte Tests Fehler nicht mehr finden können, die bereits in der Inspektion entdeckt wurden, was bei der in der Fallstudie erhobenen Daten berücksichtigt werden muss. Die Anzahl der Fehler, die in den Code im Laufe des Fehlerkorrekturprozesses eingeführt werden, wurde auf etwa 0,13 Fehler pro Stunde geschätzt. Die Fallstudie berichtet auch über eine etwas höhere Effektivität der Inspektionen, aber die Unterschiede sind eher gering: 86,5% Effektivität für Inspektionen und 80% Effektivität für die Testtechniken. Die verhältnismäßig hohe Effektivität im Vergleich mit anderer Literatur wird durch die unterschiedliche Definition der Effektivität verursacht. In [Berling, 2003] basiert die Definition der Effektivität auf der geschätzten Zahl der Fehler, die durch die Technik vielleicht gefunden werden konnten; nicht auf der Anzahl aller Fehler, die in den Dokumenten bestehen.

Ähnliche Resultate für Effizienz werden in der Fallstudie von [Conradi, 1992] berichtet. Hier wurden 0,82 Fehler pro Stunde in der Designinspektion gefunden, während nur 0,013 Fehler pro Stunde im funktionalen Testen ermittelt wurden. Diese empirischen Daten unterstützen die Tatsache, dass Designinspektionen das effizienteste Mittel für die Aufdeckung von Designfehler sind.

6.3.3 Programmierungsfehler

Hinsichtlich Programmierungsfehler ist eine relative große Anzahl von Experimenten der Frage nachgegangen ob Inspektionen oder Testtechniken effektiver sind. Jedoch gibt es keine klare Antwort, ob Codeinspektionen oder funktionale bzw. Strukturbasierende Testtechniken für die Entdeckung von Programmierungsfehler vorzuziehen sind. Die Resultate tendieren zu der Aussage, dass Testen effektiver und effizienter ist, obgleich sekundäre Faktoren wie der Informationsausbreitungseffekt von Inspektionen, der Wert der Modultestautomatisierung, die Kosten der verwendeten Testsuite oder der Nutzen von Testgetriebenen Design nicht in Betracht gezogen wird [Andersson, 2006].

Einige der Experimente sind dabei Replikationen früherer Experimente, die jedoch nicht die Ergebnisse der ursprünglichen Experimente bekräftigten. Es scheint, dass Faktoren existieren, die nicht gemessen oder kontrolliert werden, aber die die Leistung der Methoden trotzdem beeinflussen. Daher kann es nützlich sein andere „weiche“ Faktoren wie z.B. Motivation und Zufriedenheit zu erforschen [Höst, 2005] und Methoden insbesondere in der Praxis zu anwenden und zu beobachten.

6.3.4 Zusammenfassung

Die Tabelle 2 fast die Ergebnisse empirischer Untersuchungen hinsichtlich der Effektivität einer Qualitätssicherungsmaßnahme bezogen auf eine Fehlerklasse zusammen.

	Inspektionen	**Testen**
Anforderungsfehler	+++	− − −
Designfehler	+++	+
Programmierungsfehler	+	++

Tabelle 2 Zusammenfassung: Tauglichkeit der QS-Techniken für die Suche nach Fehler bestimmter Fehlerklassen

Literatur

[Adams, 1999] Adams, Tom: A formula for the re-inspection decision (24). In: Software Engineering Notes, USA, Nr. 3, S. 80. 1999.

[American National Standards Institute, 1995] IEEE Std 1044.1-1995, 1995: IEEE Guide to Classification for Software Anomalies.

[Andersson, 2003] Andersson C.; Thelin T.; Runeson, P.; Dzamashvili, N.: An Experimental Evaluation of Inspection and Testing for Detection of Design Faults. Proceedings of the 2nd International Symposium on Empirical Software Engineering, S. 174-184: 2003.

[Andersson, 2006] Andersson C.: Managing Software Quality through Empirical Analysis of Fault Detection. Doctoral thesis, Lund University, Department of Communication Systems: 2005.

[Aurum, 2002] Aurum A.; Petersson H.; Wohlin C.: State-of-the-Art: Software Inspections after 25 Years. Software Testing, Verification and Reliability, 12(3), S. 133-154: 2002.

[Basili, 1987] Basili V.R.; Selby R.: Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering 12(12), S. 1278-1296: 1987.

[Berling, 2003] Berling T.; Thelin T.: An Industrial Case Study of the Verification and Validation Activities. Proceedings 9th International Software Metrics Symposium, S. 226-238: 2003.

[Biffl, 2000] Biffl, Stefan; Grechening, Thomas; Köhle Monika: Evaluation of inspectors' defect estimation accuracy for a requirements document after individual inspection. In: Proceedings Seventh Asia-Pacific Software Engineering Conference. ASPEC 2000, Singapore, 5-8 Dec. 2000, S. 100–109. 2000.

[Briand, 1997] Briand, Lionel C.; El Emam, Khaled; Freimut, Bernd; Laitenberger, Oliver: Quantitative evaluation of capture-recapture models to control software inspections. In: Proceedings The Eighth International Symposium on Software Reliability Engineering, Albuquerque, NM, USA, 2-5 Nov. 1997, S. 234–244. 1997.

[Briand, 1997] Briand, Lionel C.; Laitenberger, Oliver; Wieczorek, Isabella: Building resource and quality management models for software inspec-

tions. Proceedings of the International Software Consulting Network and International Software Engineering Network. 1997. (ISERN 97-06).

[Briand, 2000] Briand, Lionel C.; El Emam, Khaled; Freimut, Bernd; Laitenberger, Oliver: A comprehensive evaluation of capture-recapture models for estimating software defect content (26). In: IEEE Transactions on Software Engineering, USA, Nr. 6, S. 518–540. 2000.

[Briand, 2004] Briand, Lionel C.; Freimut, Bernd; Vollei, Ferdinand: Using multiple adaptive regression splines to support decision making in code inspections (73). In: Journal of Systems and Software, USA, Nr. 2, S. 205–217. 2004.

[Chaar, 1993] Chaar, Jarir K.; Halliday, Michael J.; Bhandari, Inderpal S.; Chillarege, Ram: In-process evaluation for software inspection and test (19). In: IEEE Transactions on Software Engineering, USA, Nr. 11, S. 1055–1070. 1993.

[Chillarege, 1992] Chillarege, Ram; Bhandari, Inderpal S.; Chaar, Jarir K.; Halliday, Michael J.; Moebus, Diane S.; Ray, K. Bonnie; Wong, Man-Yuen: Orthogonal defect classification - A concept for in-process measurements (18). In: IEEE Transactions on Software Engineering, Nr. 11, S. 943–956. 1992.

[Christenson, 1987] Christenson, Dennis A.; Huang, Steel T.: Code inspection Management using statistical control limits (41). In: Proc. Nat. Commun. Forum, S. 1095–1100. 1987.

[Christenson, 1990] Christenson, Dennis A.; Huang, Steel T.; Lamperez, Alfred J.: Statistical quality control applied to code inspections (8). In: IEEE Journal on Selected Areas in Communications, USA, Nr. 2, S. 196–200. 1990.

[Conradi, 1999] Conradi R.; Marjara A.S.; Skåtevik B.: An Empirical Study of Inspection and Testing Data at Ericsson, Norway. Proceedings 24th NASA Software Engineering Workshop, Greenbelt/Washington, USA: 1999.

[Dybå, 2005] Dybå T.; Kitchenham B. A.; Jørgensen M.: Evidence-Based Software Engineering for Practitioners. IEEE Software 22(1), S. 58-65: 2005.

[Ebenau, 1994] Ebenau, Robert G.; Strauss, Susan H. 1994: Software inspection process. New York: McGraw-Hill (McGraw-Hill systems design & implementation series).

[Eick, 1992] Eick, Stephen G.; Loader, Clive R.; Long, M. David; Votta, Lawrence G.; Wiel, Scott Vander: Estimating software fault content before coding. In: Proceedings of the 14th international conference on Software engineering ICSE '92, S. 59–65. 1992.

[El Emam, 2000] El Emam, Khaled; Laitenberger, Oliver; Harbich, Thomas: The application of subjective estimates of effectiveness to controlling software inspections (54). In: Journal of Systems and Software, USA, Nr. 2, S. 119–136. 2000.

[Elberzhager, 2005] Elberzhager F.: Analysis of Empirical Findings on Quality Assurance Techniques. Master's thesis, TU Kaiserslautern: 2005.

[Freimut, 1997] Freimut, Bernd 1997: Capture-recapture models to estimate software fault content. Master's Thesis. Kaiserslautern, Germany. University of Kaiserslautern.

[Freimut, 2000] Freimut, Bernd; Klein, Brigitte; Laitenberger, Oliver; Ruhe, Günther 2000: Measurable software quality improvement through innovative software inspection technologies at Allianz Life Assurance. Fraunhofer IESE. Kaiserslautern. (IESE-Report, 014.00/E).

[Freimut, 2001a] Freimut, Bernd 2001a: Developing and using defect classification schemes. Kaiserslautern: Fraunhofer IESE (IESE-Report, 072.01/E).

[Freimut, 2001b] Freimut, Bernd; Laitenberger, Oliver; Biffl, Stefan: Investigating the impact of reading techniques on the accuracy of different defect content estimation techniques. In: Proceedings Seventh International Software Metrics Symposium. METRICS 2001, London, UK 4-6 April 2001, S. 51–62. Online verfügbar unter http://dx.doi.org/10.1109/METRIC.2001.915515. 2001b.

[Freimut, 2005] Freimut, Bernd; Denger, Christian; Ketterer, Markus: An industrial case study of implementing and validating defect classification for process improvement and quality management. In: 11th International Symposium on Software Metrics, Como, Italy, 19-22 Sept. 2005, S. 10 pp. 2005.

[Freimut, 2006] Freimut, Bernd 2006: MAGIC. A hybrid modeling approach for optimizing inspection cost-effectiveness. Stuttgart: Fraunhofer-IRB-Verl. (PhD theses in experimental software engineering, 20).

[Gilb, 1996] Gilb, Tom; Graham, Dorothy; Finzi, Susannah 1996: Software inspection. Reprinted. Harlow, England: Addison-Wesley.

[Goel, 1979] Goel A.L.; Okumoto K.: Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures. IEEE Transactions on Reliability 28(3), S. 206-211: 1979.

[Grady, 1992] Grady, Robert B. 1992: Practical software metrics for project management and process improvement. Englewood Cliffs, NJ: Prentice Hall.

[Hetzel, 1976] Hetzel W.C.: An Experimental Analysis of Program Verification Methods. Ph.D. dissertation, University of North Carolina, Chapel Hill: 1976.

[Hetzel, 1993] Hetzel B.: Making Software Measurement Work: Building an Effective Program. John Wiley & Sons: 1976.

[Höst, 2005] Höst M.; Wohlin C.; Thelin T.: Experimental Context Classification: Incentives and Experience of Subjects. Proceedings of the 27th International Conference on Software Engineering, S. 470-478: 2005.

[IBM, 2002] IBM 2002: Orthogonal Defect Classification. Online verfügbar unter http://www.research.ibm.com/softeng/ODC.

[Jelinski, 1972] Jelinski Z.; Moranda P.; Freiberger W.: Software reliability research. Academic Press: 1972.

[Jeske, 2001] Jeske D.R.; Pham H.: On the Maximum Likelihood Estimates for the Goel-Okumoto Software reliability Model. The American Statistician 55(3), S. 219-222: 2001.

[Juristo, 2002] Juristo N.; Moreno A.M.; Vegas S.: A Survey on Testing Technique Empirical Studies: How Limited is Our Knowledge. Proceedings of the 1st International Symposium on Empirical Software Engineering, S. 161-172: 2002.

[Juristo, 2003] Juristo N.; Vegas S.: Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect? In Empirical Methods and Studies in Software Engineering, R. Conradi and A. I. Wang (Eds.), Springer, S. 208-232: 2003.

[Kamsties, 1995] Kamsties E.; Lott C.M.: An Empirical Evaluation of Three Defect-Detection Techniques. Proceedings of the 5th European Software Engineering Conference, S. 362-383: 1995.

[Kim, 2000] Kim J.M.; Porter A.; Rothermel G.: An Empirical Study of Regression Test Application Frequency. Proceedings of the 22nd International

Conference on Software Engineering, Limerick, Ireland, IEEE Computer Society Press, S. 126-135: 2000.

[Laitenberger, 1997] Laitenberger, Oliver; DeBaud, Jean-Marc: Perspective-based reading of code documents at Robert Bosch GmbH (39). In: Information and Software Technology, Netherlands, Nr. 11, S. 781–791. 1997.

[Laitenberger, 1998] Laitenberger O.: Studying the Effects of Code Inspection and Structural Testing on Software Quality. Proceedings of the 9th International Symposium on Software Reliability Engineering, S. 237-246: 1998.

[Laitenberger, 1999] Laitenberger, Oliver; Leszak, Marek; Stoll, Dieter; El Emam, Khaled: Quantitative modeling of software reviews in an industrial setting. In: Proceedings of METRICS '99: Sixth International Symposium on Software Metrics, Boca Raton, FL, USA, S. 312–322. 1999.

[Laitenberger, 2000] Laitenberger, Oliver; DeBaud, Jean-Marc: An encompassing life cycle centric survey of software inspection (50). In: Journal of Systems and Software, USA, Nr. 1, S. 5–31. 2000.

[Laitenberger, 2002] Laitenberger, Oliver; Beil, Thomas; Schwinn, Thilo: An industrial case study to examine a non-traditional inspection implementation for requirements specifications. In: Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ont., Canada, 4-7 June 2002, S. 97–106. 2002.

[Lyu, 1995] Lyu M.R.: Handbook of Software Reliability Engineering. McGraw-Hill: 1995.

[Lyu, 1996] Lyu, Michael R. 1996: Handbook of software reliability engineering. Los Alamitos, New York: IEEE Computer Society (McGraw-Hill).

[Lyu, 2002] Lyu M.R.: Encyclopedia of Software Engineering. John Wiley & Sons, chapter Software Reliabiliy Theory: 2002.

[Miller, 1999] Miller, James: Estimating the number of remaining defects after inspection (9). In: Software Testing, Verification and Reliability, UK, Nr. 3, S. 167–189. 1999.

[Musa, 1998] Musa, John D. 1998: Software reliability engineering. More reliable software, faster development and testing. New York: McGraw-Hill.

[Myers, 1978] Myers G.J.: A Controlled Experiment in Program Testing and

Code Walkthroughs/Inspections. Communications of the ACM 21(9), S. 760-768: 1978.

[Petersson, 2004] Petersson, Hakan; Thelin, Thomas; Runeson, Per; Wohlin, Claes: Capture-recapture in software inspections after 10 years research. theory, evaluation and application (72). In: Journal of Systems and Software, USA, Nr. 2, S. 249–264. 2004.

[Pham, 2000] Pham H.: Software reliability. Springer Singapore: 2000.

[Porter, 1996] Porter, Adam A.; Siy, Harvey P.; Votta, Larry G.: A review of software inspections. In: Advances in Computers. 1996.

[Roper, 1997] Roper M.; Wood M.; Miller J.: An Empirical Evaluation of Defect Detection Techniques. Information and Software Technology 39(11), S. 763-775: 1997.

[Runeson, 2003] Runeson P.; Andrews A.: Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. Proceedings of the 14th International Symposium on Software Reliability Engineering, S. 3-13: 2003.

[So, 2002] So S.S.; Cha S.D.; Shimeall T.S.; Kwon Y.R.: An Empirical Evaluation of Six Methods to Detect Faults in Software. Software Testing, Verification and Reliability 12(3), S. 155-171: 2002.

[Thelin, 2002] Thelin, Thomas; Petersson, Hakan; Runeson, Per: Confidence intervals for capture–recapture estimations in software inspections (44). In: Information and Software Technology, Netherlands, Nr. 12, S. 683–702. 2002.

[Weller, 1994] Weller, Edward F.: Using metrics to manage software projects (27). In: Computer, USA, Nr. 9, S. 27–33. Online verfügbar unter http://dx.doi.org/10.1109/2.312035. 1994.

[Wohlin, 1998] Wohlin, Claes; Runeson, Per: Defect content estimations from review data. In: Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, 19-25 April 1998, S. 400–409. 1998.

[Yaung, 1994] Yaung, Alan T.; Raz, Tzvi: A predictive model for identifying inspections with escaped defects. In: Proceedings Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 94), Taipei, Taiwan, 9-11 Nov. 1994, S. 287–292. 1994.

Appendix A – Fehlerklassifikation

IEEE Schema – Project Activity (Fehlerfindungsaktivität)

Klassifikationswert	Beschreibung
Analysis	The examination of code, algorithms, documents, or test results to determine their correctness with respect to their intended uses, or to determine their operational characteristics. Some examples of analytical techniques are performance modeling, mathematical analysis, comparison to previous research, and prototyping.
Review	A meeting during which some part of the product software or documentation is reviewed in order to detect and remedy any deficiencies that could affect its fitness for use or the environmental aspects of the product, process, or service. The review may also identify potential improvements of performance, safety, and economic aspects. The users, customers, or other interested persons are often included for their comments and approval.
Audit	An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria.
Inspection	A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include design and code inspections.
Code/CompileAssmble	The process of translating a user language program into ist relocatable or absolute machine code equivalent.
Testing	The process of exercising a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.
Validation/Qualification Testing	Formal testing, usually conducted by the developer for the customer, to demonstrate that the software meets ist specified requirements.
Support/Operational	The software has been delivered and accepted and is being used for its intended purpose in its intended environment. The software is supported and maintained by the developer or the using agency.

Walk-through	A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems.

HP Schema – Origin (Fehlerquelle) in [Grady, 1992]

Klassifikationswert	Beschreibung
Specifications/ Requirements	-/-
Design	-/-
Code	-/-
Environmental Support	-/-
Documentation	-/-
Other	-/-

HP Schema – Type (Fehlerklasse) in [Grady, 1992]

Beim HP Schema sind die Werte der Fehlerklasse (Type) abhängig von der zuvor bestimmten Fehlerquelle (Origin).

Fehlerquelle	Mögliche Klassifikationswert für Type
Specifications/ Requirements	Requirements or Specifications
	Functionality
Specifications/ Requirements AND Design	HW Interface
	SW Interface
	User Interface
	Functional Description

Design	Proc. (Interproc.) Communications
	Data Definition
	Module Design
	Logic Description
	Error Checking
	Standards
Code	Logic
	Computation
	Data Handling
	Module Interface/Implementation
	Standards
Environmental Support	Test SW
	Test HW
	Development Tools
	Integration SW

HP Schema – Mode (Fehlerquelle) in [Grady, 1992]

Klassifikationswert	Beschreibung
Missing	-/-
Unclear	-/-
Wrong	-/-
Changed	-/-
Better Way	-/-

IBM ODC – Defect Type (Fehlertyp) at [IBM, 2002]

Klassifikationswert	Beschreibung
Assign/init	Value(s) assigned incorrectly or not assigned at all; but note that a fix involving multiple assignment corrections may be of type Algorithm.
Checking	Errors caused by missing or incorrect validation of parameters or data in conditional statements. It might be expected that a consequence of checking for a value would require additional code such as a do while loop or branch. If the missing or incorrect check is the critical error, checking would still be the type chosen.
Alg/Method	Efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change. Problem in the procedure, template, or overloaded function that describes a service offered by an object.
Func/Class/Object	The error should require a formal design change, as it affects significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure(s); The error occurred when implementing the state and capabilities of a real or an abstract entity.
Timing/Serial	Necessary serialization of shared resource was missing, the wrong resource was serialized, or the wrong serialization technique was employed.
Interface/O-O Messages	Communication problems between: 1) modules, 2) components, 3) device drivers, 4) objects, 5) functions via 1) macros, 2) call statements, 3)control blocks, 4) parameter lists
Relationship	Problems related to associations among procedures, data structures and objects. Such associations may be conditional.

IBM ODC – Qualifier (Fehlertyp) at [IBM, 2002]

Klassifikationswert	Beschreibung
Missing	The defect was to due to an omission (e.g.) an assignment statement was missing.
Incorrect	The defect was to due to a commission (e.g.) a checking statement used the incorrect values.

Extraneous	The defect was to due to something not relevant or pertinent to the document or code (e.g.) there is a section of the design document which is not pertinent to the current product and should be removed.

IBM ODC – Triggers (Fehlertyp) at [IBM, 2002]

Klassifikationswert	Beschreibung
Design Conformance	The document reviewer or the code inspector detects the defect while comparing the design element or code segment being inspected with its specification in the preceding stage(s). This would include design documents, code, development practices and standards, or to ensure design requirements aren't missing or ambiguous.
Logic/ Flow	The inspector uses knowledge of basic programming practices and standards to examine the flow of logic or data to ensure they are correct and complete.
Backward Compatibility	The inspector uses extensive product/component experience to identify an incompatibility between the function described by the design document or the code, and that of earlier versions of the same product or component. From a field perspective, the customer's application, which ran successfully on the prior release, fails on the current release.
Lateral Compatibility	The inspector with broad-based experience, detects an incompatibility between the function described by the design document or the code, and the other systems, products, services, components, or modules with which it must interface.
Concurrency	The inspector is considering the serialization necessary for controlling a shared resource when the defect is discovered. This would include the serialization of multiple functions, threads, processes, or kernel contexts as well as obtaining and releasing locks.
Internal Document	There is incorrect information, inconsistency, or incompleteness within internal documentation. Prologues and code comments represent some examples of documentation which would fall under this category.
Language Dependency	The developer detects the defect while checking the language specific details of the implementation of a component or a function. Language standards, compilation concerns, and language specific efficiencies are examples of potential areas of concern.

Side Effect	The inspector uses extensive experience or product knowledge to foresee some system, product, function, or component behavior which may result from the design or code under review. The side effects would be characterized as a result of common usage or configurations, but outside of the scope of the component or function with which the design or code under review is associated.
Rare Situations	The inspector uses extensive experience or product knowledge to foresee some system behavior which is not considered or addressed by the documented design or code under review, and would typically be associated with unusual configurations or usage. Missing or incomplete error recovery would NOT, in general, be classified with a trigger of Rare Situation, but would most likely fall under Design Conformance if detected during Review/Inspection.
Simple Path	The test case was motivated by the knowledge of specific branches in the code and not by the external knowledge of the functionality. This trigger would not typically be selected for field reported defects, unless the customer is very knowledgeable of the code and design internals, and is specifically invoking a specific path (as is sometimes the case when the customer is a business partner or vendor).
Complex Path	In White/Gray Box testing, the test case that found the defect was executing some contrived combinations of code paths. In other words, the tester attempted to invoke execution of several branches under several different conditions. This trigger would only be selected for field reported defects under the same circumstances as those described under Simple Path.
Coverage	During Black Box testing, the test case that found the defect was a straightforward attempt to exercise code for a single function, using no parameters or a single set of parameters.
Variation	During Black Box testing, the test case that found the defect was a straightforward attempt to exercise code for a single function but using a variety of inputs and parameters. These might include invalid parameters, extreme values, boundary conditions, and combinations of parameters.
Sequencing	During Black Box testing, the test case that found the defect executed multiple functions in a very specific sequence. This trigger is only chosen when each function executes successfully when run independently, but fails in this specific sequence. It may also be possible to execute a different sequence successfully.
Interaction	During Black Box testing, the test case that found the defect initiated an interaction among two or more bodies of

	code. This trigger is only chosen when each function executes successfully when run independently, but fails in this specific combination. The interaction was more involved than a simple serial sequence of the executions.
Workload/Stress	The system is operating at or near some resource limit, either upper or lower. These resource limits can be created by means of a variety of mechanisms, including running small or large loads, running a few or many products at a time, letting the system run for an extended period of time.
Recovery/Exception	The system is being tested with the intent of invoking an exception handler or some type of recovery code. The defect would not have surfaced if some earlier exception had not caused exception or recovery processing to be invoked. From a field perspective, this trigger would be selected if the defect is in the system's or product's ability to recover from a failure, not the failure itself.
Startup/Restart	The system or subsystem was being initialized or restarted following some earlier shutdown or complete system or subsystem failure.
Hardware Configuration	The system is being tested to ensure functions execute correctly under specific hardware configurations.
Software Configuration	The system is being tested to ensure functions execute correctly under specific software configurations.
Blocked Test (previously Normal Mode)	The product is operating well within resource limits and the defect surfaced while attempting to execute a system test scenario. This trigger would be used when the scenarios could not be run because there are basic problems which prevent their execution. This trigger must not be used in customer reported defects.

IBM ODC – Impact (Beeinträchtigte Produktqualität) at [IBM, 2002]

Klassifikationswert	Beschreibung
Installability	The ability of the customer to prepare and place the software in position for use. (Does not include Usability).
Integrity/Security	The protection of systems, programs, and data from inadvertent or malicious destruction, alteration, or disclosure.
Performance	The speed of the software as perceived by the customer and the customer's end users, in terms of their ability to perform their tasks.

Maintenance	The ease of applying preventive or corrective fixes to the software. An example would be that the fixes can not be applied due to a bad medium. Another example might be that the application of maintenance requires a great deal of manual effort, or is calling many pre- or co-requisite requisite maintenance.
Serviceability	The ability to diagnose failures easily and quickly, with minimal impact to the customer
Migration	The ease of upgrading to a current release, particularly in terms of the impact on existing customer data and operations. This would include planning for migration, where a lack of adequate documentation makes this task difficult. It would also apply in those situations where a new release of an existing product introduces changes effecting the external interfaces between the product and the customer's applications.
Documentation	The degree to which the publication aids provided for understanding the structure and intended uses of the software are correct and complete.
Usability	The degree to which the software and publication aids enable the product to be easily understood and conveniently employed by its end user.
Standards	The degree to which the software complies with established pertinent standards.
Reliability	The ability of the software to consistently perform its intended function without unplanned interruption. Severe interruptions, such as ABEND and WAIT would always be considered reliability.
Requirements	A customer expectation, with regard to capability, which was not known, understood, or prioritized as a requirement for the current product or release. This value should be chosen during development for additions to the plan of record. It should also be selected, after a product is made generally available, when customers report that the capability of the function or product does not meet their expectation.
Accessibility	Ensuring that successful access to information and use of information technology is provided to people who have disabilities.
Capability	The ability of the software to perform its intended functions, and satisfy KNOWN requirements, where the customer is not impacted in any of the previous categories.

Appendix B – Empirische Untersuchungen: Vergleich Inspektion und Testen

[Hetzel,1976]

Typ: Experiment
Artefakt: Code (3 PL/I Programme: 64, 164 and 170 Anweisungen)
Aktor: 39 Studenten (3 zufällige Gruppe von 13)
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 37,3[2] (disciplined code reading)
Testen (funktionales; Strukturbasierendes), %: 47,7; 46,7 (specification testing; selective testing)

[Myers, 1978]

Typ: Experiment
Artefakt: Code (PL/I Program: 63 Anweisungen)
Aktor: 59 Experte
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 38,0 (Walkthroughs / Inspektionen)
Testen (funktionales; Strukturbasierendes), %: 30,0; 36,0 (functional testing; Structural testing)

[Basili, 1987]

Typ: Experiment
Artefakt: Code (erste und zweite Phase: Simple-T; dritte Phase: Fortran; P1: 22 Anweisungen, P2: 95 Anweisungen, P3: 48 Anweisungen, P4: 144 Anweisungen)
Aktor: 32 Experte + 42 fortgeschrittene Studenten (erste Phase (Univ. Md): 20 junior + 9 intermediate = *29*; zweite Phase (Univ. Md): 9 junior + 4 intermediate = *13*; dritte Phase (NASA/CSC): 13 junior + 11 intermediate + 8 advanced = *32*; Insgesamt: *74*)
Effektivität

[2] Farbnotationen:
Rot – statistisch besser
Grün – statistisch schlechter
Orange – kein statistischer Unterschied
Schwarz – nicht analysiert

Inspektion (Code Reading; Fagan Inspektionen), %: 54,1 (code reading by stepwise abstraction)
Testen (funktionales; Strukturbasierendes), %: 54,6; 41,2 (functional testing using equivalence partitioning and boundary value analysis; structural testing using 100% statement coverage criteria)

[Kamsties, 1995]

Typ: Experiment
Artefakt: Code (C Programme; 193 + 18 = *211* (ntree), 219 + 29 = *248* (cmdline) und 207 + 23 = *230* (nametbl) nonblank, non-comment lines)
Aktor: 27 Studenten (erste Durchführung): skills with C: *subjective scale* – min 0.0, median 1.5 and max 4.0 out of 5.0; *objective scale* – min 0.0, mean 1.5 and max 6.0 years of working with C; 15 students (zweite Durchführung): skills with C: *subjective scale* – min 0.0, median 1.0 and max 3.0 out of 5.0; *objective scale* – min 0.0, mean 1.5 and max 6.0 years of working with C
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 43,5 (1st replication), 50,3 (2nd replication) (code reading by stepwise abstraction)
Testen (funktionales; Strukturbasierendes), %: 47,5; 47,4 (1st replication), 60,7; 52,8 (2nd replication) (functional (black-box) testing using equivalence classes and boundary values; structural (white-box) testing with the goal of 100% coverage of branches, multiply conditions, loops and relational operators)

[Roper, 1997]

Typ: Experiment
Artefakt: Code (C Programme; 193 + 18 = *211* (ntree), 219 + 29 = *248* (cmdline) und 207 + 23 = *230* (nametbl) nonblank, non-comment lines)
Aktors: 47 Studenten
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 32,1 (code reading by stepwise abstraction)
Testen (funktionales; Strukturbasierendes), %: 55,2; 57,5 (functional testing using equivalence partitioning and boundary value analysis; structural testing using as close to 100% branch coverage as possible)

[Laitenberger, 1998]

Typ: Experiment
Artefakt: Code (C Modul; 262 nonblank, non-comment lines mit 13 Fehler)
Aktor: 20 Studenten (skills with C: min 1, median 3 and max 4 out of 4; programming experience: min 1, median 3 and max 4 out of 4)
Effektivität

Inspektion (Code Reading; Fagan Inspektionen), %: 38 (code inspection)
Testen (funktionales; Strukturbasierendes), %: 9 (structural testing using 100% coverage of all branches, multiply conditions, loops and relation operators; in sequence after inspection)

[So, 2002]

Typ: Experiment
Artefakt: Code
Aktor: 26+15 Studenten
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 17,9; 34,6
Testen (funktionales; Strukturbasierendes), %: 43,0

[Runeson, 2003]

Typ: Experiment
Artefakt: Code
Aktor: 30 Studenten
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 27,5
Testen (funktionales; Strukturbasierendes), %: 37,5

[Juristo, 2003]

Typ: Experiment
Artefakt: Code
Aktor: 196, 46 Studenten
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 20, –
Testen (funktionales; Strukturbasierendes), %: 37,7; 35,5, 75,8; 71,4

[Andersson, 2003]

Typ: Experiment
Artefakt: Design
Aktor: 51 Studenten
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 53,5
Testen (funktionales; Strukturbasierendes), %: 41,8

[Conradi, 1999]

Typ: Fallstudie
Artefakt: Design, Code
Aktor: Experte

Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: –
Testen (funktionales; Strukturbasierendes), %: –

[Berling, 2003]

Typ: Fallstudie
Artefakt: Anforderungen, Design, Code
Aktor: Experte
Effektivität
Inspektion (Code Reading; Fagan Inspektionen), %: 86,5
Testen (funktionales; Strukturbasierendes), %: 80,0

Dokumenten Information

Titel:	AP 3.1: Effektivität von QS-Maßnahmen Stand der Wissenschaft
Datum:	30. Juni, 2007
Report:	TestBalance-096.07/D
Status:	Final
Verteilung:	Öffentlich