Agile Softwareentwicklung im Forschungsumfeld

Bachelorarbeit im Fach Computational Engineering

vorgelegt von

Christoph Settgast

geb. *13.11.1986* in *Kassel*

angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 2 (Programmiersysteme)
Friedrich–Alexander–Universität Erlangen–Nürnberg
(Prof. Dr. M. Philippsen)

in Zusammenarbeit mit der

Abteilung Hochfrequenz- und Mikrowellentechnik
Fraunhofer–Institut für Integrierte Schaltungen

Betreuer

Prof. Dr. Dirk Riehle
apl. Prof. Dr. Gabriella Kókai

Beginn der Arbeit: *02.11.2009*
Abgabe der Arbeit: *25.01.2010*

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen–Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den *25.01.2010*

Christoph Settgast

Bachelorarbeit

Thema: Agile Entwicklungsmethoden im Forschungsumfeld

Hintergrund: Softwareentwicklung in Forschungsabteilungen läuft meist nach organisch gewachsenen Strukturen ab. Dies ist sowohl auf die reine Prototypenentwicklung ohne späteren Support als auch auf die hohe Fluktuation unter den studentischen Mitarbeitern zurückzuführen. Daher sind Softwareentwicklungsmethoden mit hohem Dokumentationsumfang wie das V-Modell selten anzutreffen. Die agilen Entwicklungsmethoden, auch eXtreme Programming genannt, verfolgen hier einen anderen Ansatz, der eventuell auch im Forschungsumfeld erfolgsversprechend ist.

Aufgabenstellung: Nach Dokumentation der aktuellen Entwicklungssituation soll anhand der Methode der testgetriebenen Entwicklung (Test-Driven-Development) die Eignung von agilen Softwaremethoden für das Forschungsumfeld beurteilt werden. Dazu eignet sich eine Fallstudie, deren Ziele und Methodik selbst festgelegt werden kann. Dabei ist die Auswahl geeigneter Metriken und Aufgabenstellungen zu begründen. Weiterhin müssen beeinflussende Faktoren beschrieben und soweit möglich minimiert werden. Schließlich soll die Studie selbst durchgeführt werden.

Betreuung: Prof. Dr. Dirk Riehle, apl. Prof. Dr. Gabriella Kókai

Bearbeiter: Christoph Settgast

Zusammenfassung

Softwareentwicklung in Forschungsabteilungen läuft meist nach organisch gewachsenen Strukturen ab. Dies ist sowohl auf die reine Prototypenentwicklung ohne späteren Support als auch auf die hohe Fluktuation unter den studentischen Mitarbeitern zurückzuführen. Methoden der klassischen Softwaretechnik sind sehr selten anzutreffen. Innerhalb dieser Arbeit wurde untersucht, inwiefern sich agile Softwareentwicklung mit ihrer radikal anderen Grundeinstellung für dieses Umfeld eignet.

Dabei wurde der aktuelle Entwicklungsprozess dokumentiert und dann die organisatorischen Rahmenbedingungen mit den grundlegenden Annahmen agiler Softwareentwicklung in Einklang gebracht. Die Einführung wurde beispielhaft mit testgetriebener Entwicklung als Experiment entworfen und durchgeführt. Dabei wurde deutlich, dass sowohl die Einarbeitungszeit als auch der kontinuierliche Zeitaufwand in vertretbarem Rahmen blieben. Lediglich die Selbstdisziplin der Entwickler stellte sich als möglicher Hinderungsgrund für die Anwendbarkeit heraus.

Abstract

Software development in research facilities is mostly following unstructured patterns. This can be attributed to the prototyping products, which do not necessarily need support after their development and a frequently changing participating students. Classical software engineering approaches are not very widespread, but agile methods may get accepted due to their completely different values. With flexibility over organization the need of research may be fitted a lot better.

The current development process was documented in detail and the values of agile development methods seemed to be consistent with the organizational structures of the studied research group. Additionally an experiment was developed and set up to mimic the introduction of agile methods. The practice of test-driven development was chosen for that and it turned out to need few training and little additional time. Only the discipline of the developers may hold up a broad adoption in the research group.

Inhaltsverzeichnis

1 Einleitung 1

2 Voraussetzungen und aktueller Entwicklungsprozess 3

2.1 Organisatorische Rahmenbedingungen . . . 3
2.1.1 Hierarchie . . . 3
2.1.2 Projektstruktur . . . 3
2.1.3 Produkte . . . 4
2.2 Mitarbeiter . . . 5
2.2.1 Ausbildung . . . 5
2.2.2 Kenntnisstand . . . 6
2.3 Aktueller Entwicklungsprozess . . . 7
2.3.1 Anforderungen und Entwurf . . . 8
2.3.2 Programmierung und Fehlerbehebung . . . 9
2.3.3 Test und Dokumentation . . . 10

3 Agile Softwareentwicklung 12

3.1 Grundlagen . . . 12
3.2 Anforderungen und Grenzen . . . 16
3.3 Unterstützende Werkzeuge . . . 18
3.3.1 Testwerkzeuge . . . 18
3.3.2 Testauswertung . . . 20
3.4 Grundsätzliche Überlegungen zur Umsetzung im Forschungsumfeld . . . 21
3.4.1 Projekte . . . 21
3.4.2 Mitarbeiter . . . 21

4 Definition und Ausarbeitung des Experiments 23

4.1 Ziel . . . 23
4.2 Entwurf . . . 24
4.3 Abstraktion . . . 25
4.4 Gültigkeitsbetrachtung . . . 27
4.5 Ausarbeitung . . . 28

5 Auswertung des Experiments 31
5.1 Einarbeitung 31
5.2 Zeitaufwand für Tests 31
5.3 Anforderungen an die Entwickler 32
5.4 Qualität des Quelltextes 32
6 Weiterführende Untersuchungen 35
6.1 Quantitative XP-Studie 35
6.2 IBM Fallstudie 35
6.3 Kontrollierte Experimente 36
7 Zusammenfassung und Ausblick 37
A Anhang: Fragebögen 39

Tabellenverzeichnis

2.1 Fachliche Ausbildung der Mitarbeiter 5
2.2 Auswertung der Fragen zu Entwicklungsmethoden 6
2.3 Selbsteinschätzung des Kenntnisstandes 7
2.4 Verteilung des Kenntnisstandes beim „eXtreme Programming“ 8
2.5 Anforderungsmanagement und Entwurf 9
2.6 Implementierung und Fehlersuche 10
2.7 Test und Dokumentation 11

3.1 Auszug der Testmethoden von Google Test 18
3.2 Grundsätzliche Überlegungen zur Umsetzung im Forschungsumfeld . . . 22

4.1 Ziel der Studie in GQM-Schreibweise 25
4.2 Schema einer Abstraktionskarte 25

5.1 Zeitmessung zur testgetriebenen Entwicklung 31
5.2 Qualitätsauswertung 33

1. Einleitung

Klassische Softwaretechnik ist aus verschiedenen Gründen im Forschungsumfeld bisher nur wenig verbreitet. Agile Entwicklungsmethoden dagegen brechen mit vielen bisherigen Grundsätzen, um anstatt strenger Vorgaben größtmögliche Flexibilität zu bieten. Daher wird in dieser Arbeit untersucht, ob aufgrund dieses neuen Denkansatzes die Softwaretechnik auch im Forschungsumfeld stärker eingebracht werden kann.

Das eXtreme Programming verspricht, „dass Entwickler jeden Tag an den Dingen arbeiten können, die wirklich wichtig sind“[3] und sowohl Kunden als auch Projektleiter „in Mitten eines Projekts die Richtung ändern können, ohne dass exorbitante Kosten auf sie zukommen“[3]. Kritiker hingegen bemängeln, dass es „alles über Bord werfe, was in den letzten Jahrzehnten an Vorgehensmethodiken so mühsam erarbeitet worden ist“[21].

Im Forschungsumfeld ist dagegen meist eine andere Art der „extremen“ Programmierung vorzufinden. Die hohe Fluktuation der Mitarbeiter erzwingt große organisatorische Flexibilität. Außerdem werden vorwiegend Prototypen entwickelt, die kurze Entwicklungszyklen erfordern und kaum Wartungsanforderungen besitzen. Diese Prototypen wurden bisher durch reine Hardwareentwicklung realisiert, jedoch werden zunehmend Funktionen in Software implementiert. Die Softwareentwicklung folgt bisher dem intuitiven Ansatz, der in der Literatur als „Build-and-Fix“-Prozess bezeichnet wird. Der Entwicklungsprozess enthält keine festen Meilensteine wie Analyse, Entwurf und Test. Dabei verhindert geringe Modularität häufig das Einbinden fertiger Bibliotheken, wodurch eigene Implementierungen für Standardfunktionalität nötig sind. Außerdem werden Dokumentations- und Kodierrichtlinien, falls sie überhaupt vorliegen, oft nicht eingehalten.

Zur Beantwortung der eingangs gestellten Frage werden zunächst der Kontext und der aktuell eingesetzte Entwicklungsprozess im Forschungsumfeld des Fraunhofer IIS dokumentiert. Anschließend wird die agile Softwareentwicklung mit Schwerpunkt auf der testgetriebenen Entwicklung hinsichtlich ihrer Besonderheiten, Anforderungen und Grenzen vorgestellt. Dabei wird auch auf unterstützende Werkzeuge wie Testframeworks und Hilfsmittel zur statischen und dynamischen Quelltextanalyse eingegangen. Für die Untersuchung der Anwendbarkeit dieses Entwicklungsprozesses wird ein Experiment entworfen und durchgeführt. Dieses analysiert den Mehraufwand, der sich durch die Einarbeitung und den neuen Entwicklungsprozess ergibt. Außerdem wird qualitativ ausgewertet, wie sich Test und Modularität gegenüber dem „Build-and-Fix“-Prozess verändern.

Diese Arbeit beschränkt sich auf die Anwendbarkeit testgetriebener Entwicklung im Forschungsumfeld, an die sich weitere Untersuchungen zur Effizienz anschließen können. Ähnliche Fallstudien und kontrollierte Experimente [7, 15, 17] mit teilweise widersprüchlichen Ergebnissen wurden von Universitäten und großen Softwareunternehmen durchgeführt. Diese werden abschließend in der Arbeit vorgestellt.

2. Voraussetzungen und aktueller Entwicklungsprozess

In einem Forschungsbetrieb wie der untersuchten Abteilung des Fraunhofer IIS gibt es besondere Voraussetzungen, die an Entwicklungsprozesse gestellt werden. Der Prozess ist dabei eine abstrakte Handlungsvorlage, die innerhalb eines konkreten Projekts verwendet wird. Dabei kann es sich um Softwareprojekte, Hardwareprojekte oder beliebige andere Projekte handeln. Auch die agilen Praktiken werden in dieser Arbeit als leichtgewichtige Prozesse verstanden. Der Begriff „Entwicklungsmethode“ ist synonym zu verstehen.

2.1. Organisatorische Rahmenbedingungen

2.1.1. Hierarchie

Die Hierarchie der betrachteten Gruppe des Fraunhofer IIS ist wie in jeder Forschungseinrichtung flach gehalten, hier kommen auf einen Gruppenleiter 22 Mitarbeiter und nochmals 28 studentische Hilfskräfte. Im Schnitt betreut jeder Mitarbeiter mindestens eine studentische Hilfskraft oder einen Studenten, der seine Abschlussarbeit schreibt. Diese sind für den Softwareentwicklungsprozess von Belang, da im Rahmen der Examensarbeiten viel Programmierarbeit geleistet wird. Andererseits sind die Studenten nach der Fertigstellung ihrer Arbeit häufig nicht mehr vor Ort, sodass auch die Ansprechpartner für die jeweiligen Programmteile dann nicht mehr verfügbar sind. Aus dieser Besonderheit der hohen Mitarbeiterfluktuation ergibt sich ein entsprechend hohe Anforderung an Dokumentation und Lesbarkeit von Programmtexten.

2.1.2. Projektstruktur

Die Projekte werden in Matrixstruktur organisiert, daher besitzt jeder Mitarbeiter eine Kernkompetenz, die in mehreren Projekten Anwendung findet. Unter diesen Projekten lassen sich zwei Hauptkategorien ausmachen, der Einfachheit halber wird in der Arbeit daher nur zwischen diesen beiden unterschieden, die ohne größere Überschneidung voneinander abgrenzbar sind. Während Kategorie A ein Nachfolgeprojekt eines

großen Softwareprojekts bezeichnet, besteht Kategorie *B* aus Projekten, die sich bisher hauptsächlich aus Demo- und Messekomponenten zusammensetzen. Die beschriebene Matrixstuktur muss bei der Anwendbarkeit agiler Methoden berücksichtigt werden.

2.1.3. Produkte

Der veränderte Produktgedanke ist eine weitere Besonderheit des Umfelds. Forschungsergebnisse der Gruppe können nicht mit klassischen Softwareprodukten verglichen werden, da die Wartung und Pflege bei Forschungsprodukten weitgehend entfällt. Gerade bei Projekten der Kategorie *B* werden hauptsächlich Testsysteme für Messkampagnen und Prototypen für Fachmessen erstellt, die nach der jeweiligen Messe oder Messkampagne nicht weiter gepflegt werden müssen. Aufgrund der geringen Modularität ist es zudem nicht üblich, den Quelltext wiederzuverwenden. Daher entfällt die Notwendigkeit, eine ausreichende Dokumentation für Wartung und Pflege sowie weiterführende Projekte zu erstellen.

Einzig aufgrund der oben beschriebenen hohen Mitarbeiterfluktuation ist die Dokumentation schon während der Implementierung eines Projekts wichtig, da nur diese einen kontinuierlichen Wissensaustausch ermöglicht.

2.2. Mitarbeiter

Um die Umsetzung der Entwicklungsmethoden und -modelle zu verstehen, ist eine Bestandsaufnahme der aktuellen Vorgehensweise des Entwicklungsteams notwendig. Dazu wurden alle Mitarbeiter mit voller Stelle, die an der Planung oder Implementierung der Software beteiligt sind, in mündlichen Interviews befragt.

Der dabei verwendete Interviewleitfaden findet sich im Anhang A. Die gestellten Fragen sind bewusst allgemein formuliert, um eine Beeinflussung der Antwort zu vermeiden. So entwickelten sich im Interviewverlauf unterschiedliche Schwerpunkte, die im Protokoll qualitativ erfasst wurden.

Dabei werden alle Ergebnisse soweit wie möglich anonymisiert. Die verwendete Benennung der Entwickler von 1 bis 11 steht in keinem Zusammenhang mit der Befragungsreihenfolge.

2.2.1. Ausbildung

Das untersuchte Team besteht aus elf Entwicklern, zusammengesetzt aus sechs Elektrotechnikern, drei Informatikern, einem Wirtschaftsinformatiker und einem Informationstechniker, wie in Tabelle 2.1 ersichtlich.

Studiengang	Zahl
Elektrotechnik	6
Informatik	3
Wirtschaftsinformatik	1
Informationstechnik	1

Tabelle 2.1.: Fachliche Ausbildung der Mitarbeiter

Dabei wurden sieben Mitarbeiter an Universitäten, vier an Fachhochschulen ausgebildet. Der berufsqualifizierende Abschluss liegt bei sechs befragten Mitarbeiten erst eineinhalb bis zweieinhalb Jahre zurück, sodass das Studienfach noch als prägend für den aktuellen Kenntnisstand gesehen werden kann.

Sowohl der meist erst kurz zurückliegende Studienabschluss als auch die Tatsache, dass ausschließlich Akademiker in der Softwareentwicklung beschäftigt sind, ist typisch für das Forschungsumfeld. Mehrere Befragte betonten die vorgefundene Selbständigkeit und Freiheit des Arbeitsumfelds daher auch als besonders positiv.

Vor diesem Hintergrund soll nun der Kenntnisstand der Mitarbeiter bei Softwareentwicklungsmodellen und -prozessen beschrieben werden.

Entwicklungsmodell	Schon gehört	Selbst eingesetzt
Wasserfallmodell	7	1
Agile / XP	6	1
Spiralmodell	4	0
V-Modell	4	1
RUP	3	0
Prototyping	2	1
Blackbox/Whitebox-Testing	1	0
Model-Driven Development	1	1
Pair-Programming	1	0
SPICE	1	0
UML	1	1
keine Antwort	3	9

Tabelle 2.2.: „Von welchen Entwicklungsmethoden haben Sie schon einmal gehört?“ und „Welche Entwicklungsmethoden setzen Sie selbst sein?“

2.2.2. Kenntnisstand

Zur Abfrage des Kenntnisstandes wurde zuerst nach selbst eingesetzten Entwicklungsmethoden, dann nach wenigstens dem Namen nach bekannten Modellen gefragt. Schlussendlich sollte jedes Modell aus der folgenden Tabelle 2.3 auf einer Skala von 0 bis 10 bewertet werden. Dabei entspricht 0, dass das Modell dem Namen nach nicht bekannt ist, und 10, dass das Modell sofort eingesetzt werden könnte, wenn es nicht bereits verwendet wird.

Die Auswahl der Modelle erfolgte unter der Prämisse, dass diese sowohl die klassischen Entwicklungsmodelle der Softwaretechnik als auch die Praktiken der agilen Softwareentwicklung abdecken. Der Fokus lag der Arbeit entsprechend bei den agilen Entwicklungsmethoden, die auch einzeln abgefragt wurden.

Zuerst soll die offene Frage nach wenigstens dem Namen nach bekannten und eventuell selbst bereits eingesetzten Entwicklungsmodellen ausgewertet werden. Dabei wurde vom Fragesteller keinerlei Hilfestellung gegeben, um die Antwort auch nicht unbewusst in eine gewisse Richtung zu drängen. Dabei sind sowohl Mehrfachnennungen als auch keine Antwort möglich. Die Ergebnisse beider Fragen finden sich in Tabelle 2.2.

Wie dort ersichtlich, ist das meistgenannte bereits bekannte Modell das Wasserfallmodell, jedoch schon darauf folgen agile Modelle, die unter dem Schlagwort „Agile / XP“ zusammengefasst sind, da von den Befragten die Begriffe synonym verwendet wurden. Des Weiteren ist auffällig, dass drei Entwickler kein Modell nennen konnten, was sie nicht wenigstens dem Namen nach schon gehört hatten. Bei den bereits selbst eingesetzten

Entwicklungsmodell	Durchschnittswert	Standardabweichung
Wasserfallmodell	4,5	3,56
Spiralmodell	3,9	3,73
V-Modell	5,1	3,78
RUP	0,9	1,64
XP	3,7	3,80
Prototyping	5,0	3,52
Unit-Testing	5,0	3,29
Test-Driven Development	3,3	3,23
Feature-Driven Development	2,7	3,66
Modelbased Development	2,3	3,13
Pair-Programming	3,3	3,26
Continuous Integration	2,2	2,75

Tabelle 2.3.: Selbsteinschätzung des Kenntnisstandes

Modellen wird deutlich, dass neun der elf Befragten noch nie nach einem Vorgehensmodell arbeiteten, zwei dafür schon etliche Erfahrung im Umgang mit Prozessen mitbrachten.

Wie in Tabelle 2.3 ersichtlich, schätzen die Mitarbeiter ihre Kenntnisse bei vorgegebenen Prozessen selbst sehr unterschiedlich ein. Bei den klassischen Methoden besitzen die Entwickler auch im Forschungsumfeld noch einen Kenntnisvorsprung vor agilen Methoden, wie anhand des V-Modells erkennbar ist. Überraschend ist jedoch, dass auch Praktiken wie Prototyping und Unit-Testing schon gut bekannt sind.

Darüber hinaus ist eine große Streuung bei der Selbsteinschätzung zu bemerken, die an der im Verhältnis zum Durchschnittswert großen Standardabweichung erkennbar ist. Am Beispiel der Verteilung des Kenntnisstandes über XP in Tabelle 2.4 zeigt sich die Streuung darin, dass die Praktiken des eXtreme Programming entweder gut beherrscht werden, oder weitgehend unbekannt sind.

Der Kenntnisstand der Software-Entwickler lässt sich als sehr unterschiedlich zusammenfassen. Einige besitzen, entweder durch ihren Bildungs- oder Berufsweg oder aus Interesse viel Wissen über Entwicklungsprozesse, andere hingegen verlassen sich bei der Programmierung vollständig auf ihre zum Teil langjährige Erfahrung.

2.3. Aktueller Entwicklungsprozess

Der Entwicklungsprozess wird gegliedert in Anforderung, Entwurf, Implementierung und Test dargestellt. Während dieser Tätigkeiten sind auch Dokumentation und Fehlersuche

Entwickler	Selbsteinschätzung zum Modell XP
Entwickler 1	0
Entwickler 2	0
Entwickler 3	1
Entwickler 4	1
Entwickler 5	1
Entwickler 6	2
Entwickler 7	3
Entwickler 8	6
Entwickler 9	8
Entwickler 10	9
Entwickler 11	10

Tabelle 2.4.: Verteilung des Kenntnisstandes beim „eXtreme Programming“

bei der Softwareentwicklung wichtig, deswegen sollen auch diese beiden zur Beschreibung des aktuellen Prozesses dargestellt werden.

2.3.1. Anforderungen und Entwurf

Bei Anforderungen und Entwurf muss zwischen den zwei hauptsächlich durchgeführten Projektkategorien *A* und *B* unterschieden werden. Die Anforderungen für Programme entstehen vorwiegend in mündlichen Besprechungen und werden dann über Flurgespräche weiterverbreitet. Bei Projekt *A* werden die Anforderungen darauf hin in einem Dokument abgelegt, und dort wird auch der Grobentwurf durchgeführt.

Bei Projekt *B* hingegen sind sich die beteiligten Mitarbeiter einig, dass eine schriftliche Anforderungsdokumentation aufgrund des überschaubaren Softwareumfangs nicht notwendig und auch nicht sinnvoll ist. Ein Befragter gab beispielsweise an, dass der sich aus der Matrixstuktur ergebene alleinige Quelltextbesitz und der überschaubare Umfang von rund 500 Zeilen Programmtext lediglich manchmal eine Rückfrage mit Kollegen notwenig machen würde, aber keinerlei schriftliche Anforderungsanalyse. Auch für den Entwurf der Software gilt, dass dieser, wie in Tabelle 2.5 ersichtlich, nur in Ausnahmefällen weiter schriftlich durchgeführt wird. Hier wird ebenso mehrheitlich kein Softwareentwurf im klassischen Sinne durchgeführt, lediglich in Ausnahmefällen werden UML oder Blockdiagramme eingesetzt.

Häufig fiel auch die Aussage „das müsste man mal machen“, wenn Anforderungs- und Entwurfsprozesse mit Klassendiagramme mit UML von den Entwicklern angesprochen wurden. In der Praxis sei jedoch bisher kein so nennenswerter Vorteil erkennbar, daher würden die zum Teil bekannten Prozesse dann nicht eingesetzt.

Methode	Anzahl
Anforderungen	
Dokument	6
Meeting	5
persönlicher Kontakt	5
selbst gesetzte Anforderungen	4
Entwurf	
keine Entwurfsphase	6
UML	2
Blockdiagramm	2
Datenstrukturen	1

Tabelle 2.5.: Anforderungsmanagement und Entwurf

2.3.2. Programmierung und Fehlerbehebung

Die Implementierung der Software soll mit Hilfe der verwendeten Werkzeuge charakterisiert werden. Dazu wurden bei den Software-Entwicklern die verwendete Entwicklungsumgebungen abgefragt. Wie in Tabelle 2.6 ersichtlich, ist ein Texteditor bei neun Entwicklern das Mittel der Wahl. Lediglich sieben verwenden eine integrierte Entwicklungsumgebung wie Eclipse.

Daher ist es auch wenig verwunderlich, wenn auch bei der Fehlerbehebung selten komplexere Werkzeuge eingesetzt werden. Kostenlose Debugger wie „gdb“ werden in den meisten Fällen nur zur Fehlerbehebung im Falle von Speicherschutzverletzungen und daraus resultierenden Abstürzen eingesetzt. Die überwältigende Mehrheit verwendet Ausgabe-Debugging, da ihrer Meinung nach dies der einzig gangbare, oder wenigstens der einzig sinnvolle Weg ist. Dies wird von den Entwicklern mit guten Erfahrungen bei der Fehlersuche über Kontrollausgaben und schlechten Erfahrungen mit Debugger-Programmen begründet.

Beide Tatsachen sind vor allem vor dem Hintergrund interessant, da bei Anforderungen und Entwurf noch zwei klare Gruppen, zwischen Projekt A und Projekt B auszumachen waren, während bei Implementierung und Fehlerbehebung keine Unterschiede festgestellt werden können. Hierbei bleibt noch anzumerken, dass ein Mitarbeiter des Projekts A bei der Befragung explizit die Verwendung von Debuggern ausschloss, da die Anwendungen, die innerhalb des Projekts entwickelt werden, schon auf kleine Änderungen im Zeitverhalten sehr sensitiv seien. Daher sei die einzige Möglichkeit der Fehlersuche durch genaue Quelltext-Analyse gegeben, die selbstverständlich sehr viel Zeit in Anspruch nehme. Typischerweise werde hier zwischen zwei und drei Stunden pro Fehler verwendet.

Methode	Anzahl
Implementierung	
Texteditor	9
IDE	7
Fehlersuche	
Ausgabe-Debugging	9
Debugger (segfault)	5
Debugger (ausschließlich)	2
Logging-Framework	1

Tabelle 2.6.: Implementierung und Fehlersuche

Keiner der Entwickler antwortete, dass er einen festen Entwicklungsprozess oder Handlungsanweisung für die Implementierung verwende. Hier manifestiert sich die Vermutung, die eingangs formuliert wurde, dass klassische Softwaretechnik-Prozesse wie das V-Modell oder das Wasserfallmodell bisher keinen Anklang im Forschungsumfeld am Fraunhofer IIS fanden. Auch neuere agile Praktiken, wie Pair Programming oder die hier untersuchte testgetriebene Entwicklung werden bisher nicht eingesetzt.

2.3.3. Test und Dokumentation

Die Ergebnisse zur Frage, wie Test und Dokumentation durchgeführt werden, sind in Tabelle 2.7 zusammengefasst. Vier von elf Entwicklern antworteten, dass Unit-Tests eigentlich das richtige Werkzeug seien, jedoch hatten nur zwei der vier Entwickler bisher die Zeit gefunden, auch selbst Unit-Tests zu erstellen. Die anderen zwei Entwickler gaben an, dass der Nutzen von Unit-Tests klar sei, ihnen jedoch bisher die Zeit gefehlt habe, sich in das Thema genauer einzuarbeiten, um selbst Unit-Tests erstellen zu können. Von den verbliebenen sieben Entwicklern hatten sechs bereits von Unit-Tests wenigstens schon einmal gehört, nur ein Entwickler konnte sich unter dem Stichwort nichts vorstellen.

Des Weiteren gaben die Entwickler in den Interviews zu Protokoll, dass Software-Tests nicht einzeln, sondern falls überhaupt nur innerhalb eines Systemtests durchgeführt würden. Zusätzlich wurde angemerkt, dass meist keine Zeit für einen umfassenden Systemtest bleibe, da der Zeitrahmen für die Entwicklung sehr eng gesteckt sei.

Die Dokumentation der Softwareentwicklung geschieht laut den Befragten entweder implizit über ein Ticketsystem oder über mehrere Wikis und verstreute Dokumente. Hauptsächlich wird sie jedoch innerhalb des Quelltextes als Kommentar eingepflegt und kann dann mit Hilfe von „doxygen“ in eine lesbare Form gebracht werden.

Methode	Anzahl
Test	
manuell	6
Unit-Tests	4
Testprogramme	3
Dokumentation	
Doxygen	4
Dokumente	3
Wiki	2
keine	1

Tabelle 2.7.: Test und Dokumentation

Das Hauptproblem dieser verstreuten Dokumentation ist deren schlechte Auffindbarkeit, die auch von den Mitarbeitern bemängelt wurde. Ein Grund ist hier in der den Mitarbeitern gegebenen Freiheit zu suchen, da keine verbindlichen Vorgaben existieren, wie und wo Dokumentationen zu verfassen sind.

Ein Entwickler gab im Interview an, dass für Software-Dokumentation innerhalb des Entwicklungszyklus kaum Zeit bleibe, und diese daher meist nicht stattfinde.

Insgesamt lässt sich hier also zusammenfassen, dass für den eigenständigen Softwaretest meist wenig Zeit verwendet wird, und falls dieser durchgeführt wird, kaum Hilfsmittel wie Unit-Tests eingesetzt werden. Die Dokumentation der Software ist, soweit sie überhaupt vorhanden ist, an mehreren Stellen verstreut und daher schwer auffindbar.

Anforderungen und Entwurf der Software werden hauptsächlich über Meetings und den persönlichen Kontakt der Entwickler untereinander koordiniert und dann selbstständig durchgeführt. Dabei werden nur im Projekt *A* Anforderungen an die Software in Dokumenten abgelegt, für Projekt *B* ist laut Aussagen der Entwickler kein schriftliches Anforderungsmanagement und kein schriftlicher Entwurf nötig. Weitergehende unterstützende Software kommt in der Anforderungs- und Entwurfphase nicht zum Einsatz.

Bei der Programmierung kommen sowohl Texteditoren als auch integrierte Entwicklungsumgebungen wie Eclipse oder Visual Studio zum Einsatz. Bei der Fehlersuche hingegen verlassen sich neun der elf befragten Entwickler auf Ausgaben, lediglich zwei verwenden zur Fehlersuche einen Debugger, der in die Entwicklungsumgebung integriert ist.

Dedizierte Software-Tests werden laut den Entwicklern nur selten durchgeführt, lediglich zwei der elf Befragten setzen selbst Unit-Tests ein.

3. Agile Softwareentwicklung

3.1. Grundlagen

Agile Softwaretechnik und die dazugehörigen leichtgewichtigen Entwicklungsprozesse erfahren seit ihrer Vorstellung durch das „Agile Manifesto“[4] im Jahr 2001 große Aufmerksamkeit unter Entwicklern und vermehrt auch Projektleitern. Diese Popularität in der Praxis ist anhand zahlreicher erschienener Bücher, Artikel und zuletzt auch vermehrt wissenschaftlicher Arbeiten festzustellen. Dabei konzentrieren sich agile Praktiken auf den zu erstellenden Quelltext und verzichten so weit wie möglich auf organisatorische oder dokumentarische Tätigkeiten und sind damit als Gegenentwurf zu klassischen Prozessen wie dem Wasserfallmodell oder dem V-Modell zu sehen.

Das agile Manifest stellt „Individuen und deren Interaktionen über starre Prozesse und vorgeschriebene Werkzeuge, funktionierende Software über vollständige Dokumentation, Zusammenarbeit mit dem Kunden über Vertragsverhandlungen und das Annehmen von Änderungen über das starre Folgen eines vorher ausgearbeiteten Plans“[4].

Im Rahmen dieser agilen Prinzipen werden heute mehrere Praktiken und Prozesse verstanden. Diese lassen sich in programmiernähere Prozesse, dem sog. „eXtreme Programming", kurz XP und projektmanagementorientierte Prozesse untergliedern. Ich werde mich im Folgenden auf die programmiernäheren Methoden beschränken, daher sind auch unter „agilen Entwicklungsprozessen“ nur diese Praktiken des eXtreme Programming zu verstehen.

Das eXtreme Programming [3] basiert auf vier grundlegenden Werten, auf denen fünf Grundprinzipien und zehn weitere Prinzipien basieren, und ist in zwölf Praktiken für Entwickler konkretisiert. Diese werden jeweils mit ihrer englischen Originalbezeichnung aus [3] geklammert genannt.

Die grundlegenden Werte sind nach Beck [3] „Kommunikation, Einfachheit, Feedback und Eigenverantwortung“.

Kommunikation (Communication) Intensive Kommunikation der Entwickler untereinander und mit dem Kunden sollen im Vordergrund stehen, da dies ermögliche, schnelle Rückmeldungen zu erhalten und auftretende Probleme so bald wie möglich zu bemerken.

Einfachheit (Simplicity) Je einfacher eine gewählte Lösung ist, um so geringer ist, der Annahme von XP nach, die Wahrscheinlichkeit, dass das Projekt an der Komplexität scheitert. Daher sollen ausdrücklich keine Vorbereitungen für zukünftige Erweiterungen getroffen, sondern erst bei tatsächlichem Bedarf ergänzt werden. Es ist besser, „heute etwas Einfaches zu erstellen und morgen etwas mehr Aufwand zu investieren, um Änderungen einzubauen, als heute Komplexes zu entwickeln, das morgen nicht [. . .] genutzt wird“ [21].

Feedback (Feedback) Je schneller die Rückmeldung auf Entscheidungen vorliegt, desto besser kann darauf reagiert werden. Kleinschrittige Entwicklungen und ein Kunde dauerhaft vor-Ort sollen neben Unit-Tests für diese Rückmeldung sorgen.

Eigenverantwortung (Courage) Je mutiger Entwickler, im Bewusstsein ihrer Verantwortung für das Projekt, Änderungen und Planungen gestalten, desto eher werden Missstände beseitigt, Quelltext von Kollegen verbessert und offener Austausch gefördert, so die zugrunde liegende Annahme von XP.

Auf diesen Werten basieren nach Beck et. al. [3] folgende Prinzipien.

Schnelles Feedback (Rapid feedback) Beispiele aus der Verhaltensforschung zeigen, wie Beck [3] und Slembeck [22] beschreiben, dass nur sofortige Rückmeldungen den Lernprozess unterstützen. Schnelles Feedback zu Entwurfsentscheidungen, Test- und Implementierungswegen helfen auch Entwicklern besser, die Auswirkungen dieser zu verstehen, als wenn erst Wochen oder Monate später eine Rückmeldung beispielsweise durch die Testabteilung erfolgt.

Einfachheit (Assume simplicity) Dieser Grundsatz, ein Problem zuerst auf die möglichst einfache Art und Weise zu lösen, steht im Gegensatz zu den Ansätzen der klassischen Softwaretechnik. So wird angenommen, dass Wiederverwendbarkeit meist unnötigerweise eingebaut wird, sodass es aus Zeit- und Kostengründen besser ist, zuerst die möglichst einfache Lösung zu implementieren und dann erst bei tatsächlichem Bedarf komplexere Lösungen zu wählen.

Inkrementelle Änderungen (Incremental change) Große Änderungen funktionieren nur sehr selten in der Softwareentwicklung, da die Seiteneffekte häufig nicht überschaubar sind. Daher sollen möglichst kleinschrittige Überarbeitungen gewählt werden. Dies gilt nach Beck [3] auch für die Einführung des eXtreme Programming selbst.

Änderbarkeit unterstützen (Embracing change) „Nichts ist so beständig wie der Wandel“ gilt wohl unwidersprochen in jedem Softwareprojekt, daher sollte dieser dauerhafte Wandel mit in den Modellen und Methoden der Entwicklung eingearbeitet sein.

Qualitativ hochwertige Ergebnisse (Quality work) Die persönliche Motivation von Mitarbeitern sei um so höher, je besser die Ergebnisse seien, die sie vorweisen

könnten. Außerdem dürfen Entwicklungsmethoden nicht unbewusst zu Lasten der Qualität gehen, daher muss die Qualität der resultierenden Software immer auch bei der Bewertung der Entwicklungsmethode bedacht werden.

Daneben gibt es noch weitere Prinzipien, die die oben genannten ergänzen. Diese sollen hier nur der Vollständigkeit halber genannt werden und sind in [3] ausführlich beschrieben.

- Lehre das Lernen (Teach learning)
- Kleiner Anfangsaufwand (Small initial investment)
- Versuche zu gewinnen (Play to win)
- Konkrete Beispiele (Concrete experiments)
- Offene, ehrliche Kommunikation (Open, honest communication)
- Instinkte nutzen, nicht dagegen arbeiten (Work with people's instincts, not against them)
- Verantwortung annehmen (Accepted responsibility)
- An Gegebenheiten anpassen (Local adaptation)
- Leichtes Gepäck (Travel light)
- Ehrliche Messungen (Honest measurement)

Abschließend sollen die Praktiken des eXtreme Programming vorgestellt werden.

Das Planungsspiel (The Planning Game) Es wird schnell ausgehandelt, was für die nächste Veröffentlichung getan werden soll, unter Beachtung technischer und wirtschaftlicher Gründe. Vorhandene Pläne werden, falls sie von der Realität überholt wurden, entsprechend angepasst.

Kleine Veröffentlichungen (Small releases) Neue Versionen werden in kurzen Zyklen veröffentlicht.

Metapher (Metaphor) Die Entwicklung wird von einem gemeinsamen Anwendungsfall geleitet, den sowohl der Kunde als auch der Entwickler verstehen.

Einfacher Entwurf (Simple design) Der Entwurf des Systems wird so klar und einfach wie möglich gehalten und Komplexität so häufig wie möglich aus dem System entfernt.

Test (Testing) Jeder Entwickler schreibt kontinuierlich Unit-Tests, nach Möglichkeit vor der eigentlichen Implementierung. Falls Tests fehlschlagen, muss der Programmtext erst so geändert werden, dass wieder alle Tests erfolgreich ablaufen.

Refaktorisierung (Refactoring) Das Programm wird in regelmäßigen Abständen so restrukturiert, dass Redundanz verringert und der Entwurf vereinfacht, aber keinerlei neue Funktionalität hinzugefügt wird.

Paar-Programmierung (Pair programming) Aller Quelltext wird von zwei Programmieren, die gemeinsam vor einem Entwicklungssystem sitzen, entworfen.

Gemeinsamer Quelltextbesitz (Collective ownership) Jeder darf jederzeit an jeder Stelle im Quelltext Änderungen durchführen.

Kontinuierliche Integration (Continuous integration) Das Gesamtsystem wird nach jeder abgeschlossenen Entwicklungsaufgabe in ein lauffähiges Produkt übersetzt.

40-Stunden-Woche (40 hour week) Überstunden zahlen sich hinsichtlich der Produktivität nicht aus.

Kunde vor Ort (On-site customer) Bei jedem Projekt ist mindestens ein Kundenvertreter im Team, der jederzeit für Fragen zur Verfügung steht.

Kodierrichtlinien (Coding standards) Entwickler schreiben Quelltext nach gemeinsam beschlossenen Richtlinien, um auf einer gemeinsamen Basis auch über den Quelltext kommunizieren zu können.

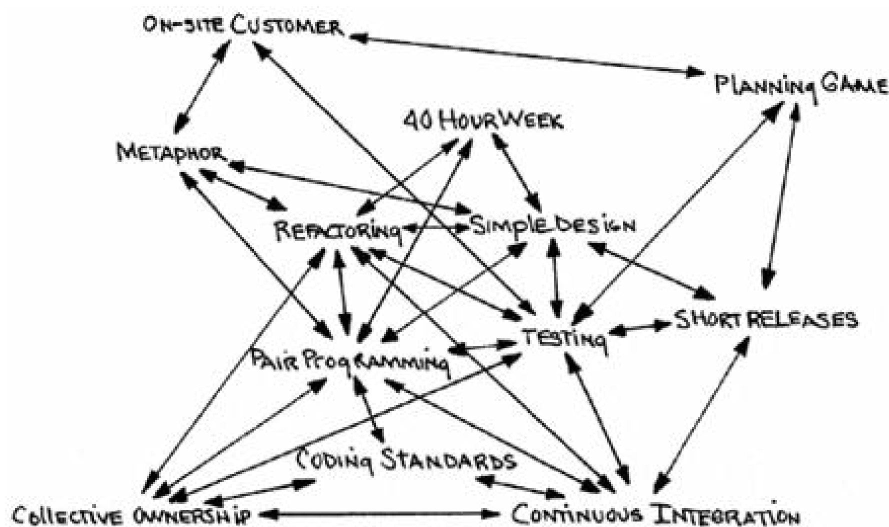

Abbildung 3.1.: Zusammenhänge zwischen den Praktiken des XP [3]

Alle Praktiken, auch die testgetriebene Entwicklung, müssen immer mit den vier Werten Kommunikation, Einfachheit, Feedback und Eigenverantwortung im Hinterkopf betrachtet werden.

Die Praktiken sind, zusammen mit ihrer gegenseitigen Beeinflussung, in Abbildung 3.1 verzeichnet. Dort ist erkennbar, dass „Test" und „Paar-Programmierung" mit je acht ein- und ausgehenden Kanten die zentralen Rollen innerhalb des eXtreme Programming einnehmen. Außerdem wichtig sind die „Refaktorisierung" mit sieben ein- und ausgehenden Kanten und „kontinuierliche Integration" und „einfacher Entwurf" mit je sechs ein- und ausgehenden Kanten. Anhand der Grafik ist aber auch sichtbar, wie eng die einzelnen Praktiken zusammenhängen, und welche Probleme es daher für das eXtreme Programming bereiten würde, falls zentrale Praktiken missachtet würden. Daher ist es sinnvoll, mit einer Untermenge der Praktiken zu beginnen, sodass ein Entwicklungsteam bei der Einführung nicht überfordert wird.

Die Praktik der testgetriebenen Entwicklung kann auch als alleinstehendes Entwicklungsmodell verwendet werden, wie Beck in [2] beschreibt. Auch Rumpe verweist in [21] darauf, dass „die stärkere Betonung der automatisierten Tests und des Refactoring" einen guten Einstieg in die Praktiken des eXtreme Programming böten. So lassen sich die oben beschriebenen Prinzipien und Werte in eine kleines, sehr kurz beschreibbares Entwicklungsmodell fassen. Daher ist der Einstieg über testgetriebene Entwicklung ein Beginn mit den Kernpraktiken, wie aus Abbildung 3.1 ersichtlich ist.

Die kürzeste Beschreibung für testgetriebene Entwicklung lautet „Anforderung, Test, Implementierung" und wird in der Literatur auch als Farbkodierung mit „Red, Green, Refactor"[2] paraphrasiert. Dies bedeutet in der Praxis, dass pro Anforderung zuerst ein Testfall geschrieben wird. Daraufhin wird die Programmbasis so verändert, dass dieser Test als bestanden gewertet werden kann. Abschließend wird alle Redundanz im Quelltext eliminiert, während darauf geachtet wird, dass sowohl der gerade erstellte Test als auch alle anderen Testfälle immer noch bestanden werden müssen. Danach wird mit der nächsten Anforderung fortgefahren, wie in Abbildung 3.2 zu sehen ist.

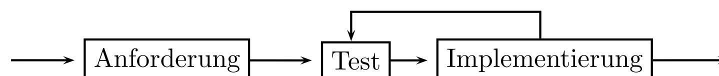

Abbildung 3.2.: testgetriebene Entwicklung als Prozess (nach [19])

3.2. Anforderungen und Grenzen

Nach Beck [2] ergeben sich durch diese testgetriebene Vorgehensweise vier Implikationen für die konkrete Entwicklung.

Es muss möglich sein, den Softwareentwurf in kleinen Schritten und parallel zur Implementierung durchzuführen, da für jeden Entwurfsschritt Informationen und Rückmeldungen aus den vorangegangenen Test- und Implementierungsphasen mit einbezogen

werden sollen. Daher werden für spätere Entwurfsschritte schon ausführbare Programmteile benötigt.

Außerdem muss der Entwurf so verlaufen, dass Module mit hoher Kohäsion und geringer Kopplung entstehen, damit ein Test der Module überhaupt in angemessener Zeit möglich ist. Diese Anforderung wird noch klarer, wenn man sie von einer anderen Seite betrachtet: Alles, was nicht testbar ist, darf auch nicht entwickelt werden. Daher kann es sein, dass mehr Zeit für den Entwurf einzelner Module verwendet werden muss, um die Testbarkeit zu gewährleisten. Die Rechtfertigung für diesen Mehraufwand sieht das eXtreme Programming in der auf lange Sicht erhofften verbesserten Wiederverwendbarkeit der Module und verbesserten Reaktionsfähigkeit der Entwicklung auf Änderungen.

Des Weiteren muss die Trennung zwischen Tester und Entwickler aufgehoben werden, da sonst die kleinschrittige Entwicklung mit häufigem Wechsel zwischen Test und Implementierung nicht möglich ist.

Auch an die Entwicklungsumgebung werden Anforderungen gestellt. Hier ist es wichtig, dass der Übersetzungsvorgang des Quelltextes schnell vonstatten geht, wodurch der einzelne Entwickler schnell Rückmeldung erhält, ob die Tests erfolgreich waren oder nicht. Dies stellt gerade bei größeren Projekten eine Schwierigkeit dar, da hier meist nur einmal täglich oder manchmal sogar nur wöchentlich übersetzt wird, sodass die kleinschrittige, testgetriebene Entwicklung besonderer Behandlung bedarf. Mit Hilfe von klar voneinander getrennten Softwaremodulen kann dann beispielsweise erreicht werden, dass nur ein Modul für den dazugehörigen Modultest übersetzt werden muss und so die Übersetzungszeit während der Entwicklung im Rahmen gehalten wird.

Damit werden auch die Grenzen der testgetriebenen Entwicklung aufgezeigt. Sowohl bei stark nebenläufigen Problemen als auch bei Sicherheitssoftware stößt testgetriebene Entwicklung an ihre Grenzen. Nebenläufigkeiten lassen sich meist nicht durch einfachen Ablauf des Testfalls erzwingen, aus diesem Grund erlaubt der Testausgang keine Aussage über die Nebenläufigkeitseigenschaften des Programms. Bei Sicherheitssoftware verhält es sich ähnlich, da dort zwar auf Fehlerfreiheit hingearbeitet wird, diese jedoch mit den ausführlichsten Tests nicht sichergestellt werden kann und daher auch eine menschliche Plausibilitätsprüfung des Programmentwurfs notwenig ist.

Eine weitere in der Praxis notwendige Anforderung stellt die automatisierte Auswertung der Tests durch das Entwicklungssystem dar. Nur mit schneller Rückmeldung, ob Tests erfolgreich waren, und falls nicht, welche Tests an welcher Stelle fehlschlugen, lässt sich dieses schnelle Feedback realisieren.

Zuletzt sei noch eine Anforderung genannt, die im Forschungsbetrieb hinderlich sein kann. Es ist notwendig, dass Entwickler so diszipliniert arbeiten, dass tatsächlich das Muster „Anforderung, Test, Implementierung“ eingehalten wird, und nicht doch kleine Teile ohne dazugehörigen Test implementiert werden, und damit die Vorteile der testgetriebenen Entwicklung zum Teil entfallen. Im Forschungsbetrieb besteht diese Gefahr

Vergleich	Methode
bool actual	EXPECT_TRUE(actual);
expected == actual	EXPECT_EQ(expected, actual);
val1 != val2	EXPECT_NE(val1, val2);
val1 <= val2	EXPECT_LE(val1, val2);
val1 < val2	EXPECT_LT(val1, val2);
val1 >= val2	EXPECT_GE(val1, val2);
val1 > val2	EXPECT_GT(val1, val2);
char* cstring1 == char* cstring2	EXPECT_STREQ(cstring1, cstring2);
cstring1 != cstring2	EXPECT_STRNE(cstring1, cstring2);
float expected == float actual	EXPECT_FLOAT_EQ(expected, actual);
double expected == double actual	EXPECT_DOUBLE_EQ(expected, actual);
val1 == val2 ± error	EXPECT_NEAR(val1, val2, error);

Tabelle 3.1.: Auszug der Testmethoden von Google Test [13]

besonders, wenn die konkreten Anforderungen noch nicht klar ersichtlich sind. Laut den Entwicklern wird teilweise erst über die Implementierung der konkrete Algorithmus schrittweise erarbeitet. Hier muss dann trotzdem die Disziplin aufgebracht werden, für jeden Teilschritt zuerst einen dazugehörigen Testfall zu erstellen, und erst daraufhin die dazugehörige Implementierung.

3.3. Unterstützende Werkzeuge

Die in Kapitel 3.2 genannten Anforderungen an das Entwicklungssystem lassen sich mit Hilfe mehrerer Werkzeuge erfüllen.

3.3.1. Testwerkzeuge

Um nach einem Testlauf schnelle Rückmeldung zu erhalten, ob dieser erfolgreich war, bieten sich gleich mehrere Testframeworks an. Diese bieten alle ähnliche Funktionalität, das bekannteste Framework nach dem xUnit-Schema ist das von Gamma und Beck entwickelte „JUnit“ für die Sprache Java [11]. Hier werden die Testframeworks beispielhaft anhand des „Google C++ Testing Frameworks“ [13] erläutert, da dessen Syntax vergleichsweise knapp ist.

Die erste Funktion eines Testframeworks ist die vereinfachte Überprüfung von Rückgabewerten mit Hilfe von „Matchern“ und Makros. Dabei kann auf Gleichheit, Ungleichheit, Größer und Kleiner zurückgegriffen werden. Außerdem werden meist C-String-

Vergleiche, Gleitkommavergleiche mit Toleranz und viele weitere Spezialfunktionen geboten. Eine Übersicht der „Matcher“ findet sich in Tabelle 3.1.

Solch eine Überprüfung für die Funktion **int** sum(**int**, **int**); ist für den Testfall von $1 + 1 = 2$ beispielhaft in Listing 3.1 vorzufinden.

```
int main(void) {
        if (sum(1,1) == 2) {
                return EXIT_SUCCESS;
        } else {
                return EXIT_FAILURE;
        }
}
```

Listing 3.1: Test ohne Testframework

Die gleiche Überprüfung reduziert sich mit Framework auf folgende Zeile:

void EXPECT_EQ(2, (sum(1,1));

Ein komplettes Beispiel ist in Listing 3.2 zu finden. Weitere Beispiele für die Anwendung von Testframeworks finden sich unter [13].

```
#include <gtest/gtest.h>

TEST(Sum, OneAndOne) {
        EXPECT_EQ(2, sum(1, 1));
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Listing 3.2: Kompletter Testfall mit Testrunner im Testframework

Daneben besitzt ein Testframework die Aufgabe, Testfälle in mehrere hierarchische Kategorien aufzuteilen. Dabei wird von Testsuiten gesprochen, die sich in mehrere Testfälle untergliedern, die sich wiederum aus mehreren einzelnen Tests zusammensetzen. Bei der Ausführung wird dann diese Hierarchie durchlaufen, aber jedem einzelnen Test eine eigene Umgebung zur Verfügung gestellt, sodass gegenseitige Beeinflussung durch die Reihenfolge der Tests ausgeschlossen wird. Dazu werden so genannte „Testrunner“ zur Verfügung gestellt, die auch für die Aufbereitung der Testergebnisse per Ausgabe oder XML zuständig sind. Durch das XML-Format ist eine automatisierte Weiterverarbeitung der Ergebnisse möglich.

Zusammenfassend lässt sich feststellen, dass Testframeworks Entwicklern die Möglichkeit bieten, kurze und möglichst lesbare Testfälle zu erstellen. Dabei können die Tests hierarchisch untergliedert und dann mit Hilfe des Frameworks aufgerufen werden. Die Resultate erhält der Entwickler dann auf der Ausgabe und als weiterzuverarbeitende XML-Datei. Testgetriebene Entwicklung ist auch ohne diese Unterstützung durch ein Framework möglich, dann aber deutlich aufwändiger.

3.3.2. Testauswertung

Für das Forschungsumfeld am Fraunhofer IIS wurde zur Testauswertung die einheitlich verwendete Entwicklungsumgebung um Testframeworkunterstützung erweitert. Diese kann bei der Entwicklung mit „make test" jederzeit aufgerufen werden, dabei werden alle Testfälle übersetzt und gestartet. Abschließend findet eine Auswertung hinsichtlich der Zeilenabdeckung statt. Diese ermöglicht es, zu erkennen, wie viele der vorhandenen Programmzeilen tatsächlich während des Testdurchlaufs ausgeführt wurden. So kann schnell erkannt werden, ob Testfälle für Programmfunktionen fehlen.

Abschließend sei noch die automatisierte Auswertung von Testfällen und Testabdeckung mit Hilfe von Werkzeugen wie Sonar [23] und MetriC++ [5] genannt. Diese ermöglichen eine kontinuierliche Überwachung der Entwicklung und zeichnen wichtige Metriken, wie die Zahl der Anweisungen oder den Kommentaranteil auf. Zusätzlich wird die zyklomatische Komplexität nach McCabe [16] für Funktionen und Klassen berechnet. Daneben sind noch viele weitere Metriken definierbar.

Daneben unterstützt Sonar den Entwickler bei der Einhaltung von Kodierrichtlinien, die mit Werkzeugen wie „Checkstyle" und „Findbugs" überprüft werden. Dadurch wird versucht, typische Implementierungsfehler sowie Speicherlecks mit statischer Quelltextanalyse zu erkennen. Als Hilfe bei der Behebung der Defekte können diese im Kontext des Quelltextes angezeigt werden.

Alle erfassten Metriken werden darüber hinaus mit der dazugehörigen Version und dem Datum des Programmes zur Trendanalyse abgespeichert. So kann die Entwicklung eines Programms auch im Nachhinein anhand der Metriken nachverfolgt werden.

Die erweiterbare Plugin-Architektur von Sonar erlaubt es darüber hinaus, neue Sprachen neben Java und weitere Metriken und Auswertungen in das Werkzeug zu integrieren. Das oben genannte MetriC++ ist ein Beispiel dafür, wie Analysewerkzeuge für C- und C++-Quelltexte in die Oberfläche von Sonar aufgenommen werden können.

3.4. Grundsätzliche Überlegungen zur Umsetzung im Forschungsumfeld

Nach der Darstellung der Situation im Forschungsumfeld in Kapitel 2 und den Anforderungen und Prinzipien der agilen Softwareentwicklung im Allgemeinen und des eXtreme Programming im Besonderen in Kapitel 3 wird versucht, beides zuerst in grobe Deckung zu bringen. Dazu wird zwischen Anforderungen an Projekte und Anforderungen an die Mitarbeiter unterschieden.

3.4.1. Projekte

Wie in Kapitel 2.1.2 beschrieben wird im vorgefundenen Umfeld die Matrixstruktur mit kleinen Projekten gelebt. Daher arbeiten im Schnitt fünf bis zehn Mitarbeiter für ein Projekt, wobei einer davon den Großteil der Arbeit beiträgt. Auch für agile Softwareentwicklung werden kleine Teams vorgeschlagen, Beck spricht hier in [3] von Größen bis zehn Entwicklern, bei 20 hält er es schon für ineffektiv. Auch Müller et. al. bestätigen dies in ihrer Fallstudie [17] mit der Angabe von sechs bis acht Entwicklern pro Team.

Außerdem sind die vielen Änderungen im Forschungsumfeld ein bezeichnendes Charakteristikum, dass vom Prozess unterstützt werden muss. Aufgrund unklarer Anforderungen oder Entwicklungen im technischen Neuland sind Schwierigkeiten oder der genaue Weg zu Beginn eines Projekts nur schwer abzuschätzen. Daher geschieht die Entwicklungsarbeit meist in kleinen Schritten, in denen in Prototypen eine Vorgehensweise ausprobiert wird, von der zu Beginn noch nicht klar ist, ob sie tatsächlich zum Ziel führt. Die agile Softwareentwicklung hat genau dieses Charakteristikum zu einem zentralen Prinzip erhoben, sodass hier keinerlei Probleme zu erwarten sind.

Nach den organisatorischen Rahmenbedingungen spricht daher nichts gegen eine Anwendbarkeit von agiler Softwareentwicklung.

3.4.2. Mitarbeiter

Die zweite zu beantwortende Frage bezieht sich auch die grundlegenden Voraussetzungen, die die Entwickler im Forschungsumfeld mitbringen. Agile Softwareentwicklung geht von örtlich nicht verteilten Teams aus, wodurch Kommunikation ohne weitere Hürden möglich ist. Dies ist im Umfeld meist gegeben, die Entwickler beschrieben den vorhandenen persönlichen und fachlichen Austausch sogar als sehr gut und produktiv.

Vor allem unter den studentischen Mitarbeitern besteht hohe Fluktuation, da diese häufig nur ihre Abschlussarbeit verfassen oder ein Praktikum ableisten, sodass die Beschäftigung meist auf sechs, teilweise auf drei Monate beschränkt ist. Hier eröffnet sich ein

Forschungsumfeld	Agile Methoden
• Kleine Teams • Matrixstruktur • Meist örtliche Nähe • Guter persönlicher Kontakt • viele Änderungen, Anforderungen werden erst im Laufe des Projekts klar • Hohe Fluktuation	• Kleine Teams • Örtliche Nähe • Kommunikation • Änderungen sind selbstverständlich • Lehre das Lernen

Tabelle 3.2.: Grundsätzliche Überlegungen zur Umsetzung im Forschungsumfeld

Problemfeld, da die vollständige Menge der Praktiken eine gewisse Einarbeitungszeit voraussetzt. Studentische Mitarbeiter sollten daher eventuell nicht alle, sondern nur eine reduzierte Untermenge der Praktiken verwenden. Es bietet sich hier der Einstieg mit testgetriebener Entwicklung an.

Es sollte andererseits aber auch nicht unerwähnt bleiben, dass die agile Softwareentwicklung bezüglich der Einarbeitung Vorteile anderen Prozessen gegenüber bietet, da nicht dogmatisch an Details des Prozesses festgehalten werden muss, sondern Anpassungen und Änderungen auch den Prozess selbst betreffen sollen. Daneben ist es im eXtreme Programming wichtig, dass die Werkzeuge und Praktiken nicht von oben herab beschlossen werden sollen, sondern dass Entwicklern Möglichkeiten und Wege der eigenen Verbesserung an die Hand gegeben werden sollen. Dies ist mit dem Prinzip „Lehre das Lernen“ oben aufgeführt.

Wie in Tabelle 3.2 ersichtlich, bleibt festzuhalten, dass kein großer Hinderungsgrund, sondern mögliche Erfolgsfaktoren für agile Softwareentwicklung im Forschungsumfeld gefunden wurden. Die kleinen Projektteams, viele Änderungen und der gute persönliche Kontakt sprechen für eine Eignung agiler Softwareentwicklung. Eine Untersuchung kann hier jedoch nicht enden, im Folgenden wird die Einführung testgetriebener Entwicklung als Experiment begleitet und untersucht.

4. Definition und Ausarbeitung des Experiments

Wie in Kapitel 3 aufgezeigt, wird testgetriebene Entwicklung als guter Einstieg in agile Entwicklungsmethoden gesehen. Außerdem spricht nach einer ersten Analyse keine der grundlegenden Annahmen gegen eine Eignung. Es ist jedoch noch offen, ob dieser Prozess tatsächlich auf das Forschungsumfeld anwendbar ist.

Um der Antwort darauf etwas näher zu kommen, bietet sich ein Experiment an, wie es Prechelt in [18] beschreibt. Dabei wird zuerst das zu untersuchende Ziel bestimmt, darauf aufbauend mit Hilfe der „Goal/Question/Metric"-Methode[1] passende Metriken definiert und nach Vorbereitung aller nötigen Materialien das Experiment durchgeführt.

4.1. Ziel

Die zentrale Fragestellung, die im Laufe des Experiments beantwortet werden soll, lautet:

Analyse der Anwendbarkeit testgetriebener Entwicklung im Forschungsumfeld

Es wird konsequent eine Analyse der Anwendbarkeit vorgenommen. In der Literatur gibt es bisher keine vergleichbare Aufarbeitung. Die Anwendbarkeit ist als erste und schwächste mögliche Form der Akzeptanz zu verstehen, die Effektivität wird hier explizit nicht betrachtet. Daher ist dies als erster Schritt aufzufassen, eine folgende Arbeit kann dann die Effektivität testgetriebener Entwicklung beinhalten.

Die Anwendbarkeit für den Entwicklungsbetrieb im Forschungsumfeld soll anhand der im Fraunhofer IIS vorherrschenden Rahmenbedingungen entschieden werden. Ein geeigneter Prozess ist mit der flachen Hierarchie, der Matrixstruktur für Projekte, der hohen Fluktuation unter studentischen Mitarbeitern und unterschiedlichen Kenntnisständen der Mitarbeiter vereinbar. Bestenfalls sollte der Prozess aus den genannten Faktoren Nutzen ziehen, diesen in jedem Fall aber nicht entgegenstehen.

Des Weiteren ist die Anwendbarkeit an den Resultaten, also der Qualität des entstehenden Quelltextes zu messen. Dieser Faktor muss beachtet werden, denn der bequemste Prozess eignet sich nicht, wenn das Ergebnis, in diesem Fall der resultierende Programmtext, nicht den gewünschten Qualitätsansprüchen genügt. Gleichzeitig ist dies eine sehr

schwer messbare Größe, da weder ein Modul dem anderen gleicht, noch eine Iteration eines Moduls der vorangegangenen. Daher wird für die Anwendbarkeit lediglich die Tendenz der Softwarequalität betrachtet.

4.2. Entwurf

Für den Entwurf der Studie wird die „Goal/Question/Metric"-Methode (GQM) [1] herangezogen, da diese es neben einer Verfeinerung des gesuchten Ziels erlaubt, die passenden Metriken systematisch herauszufinden.

Dazu wird zuerst das *Messobjekt* festgelegt. Dieses kann das Produkt, ein Prozess, Metriken, Modelle oder Theorien sein, beispielsweise sei hier der Entwicklungs- oder Testprozess, die zyklomatische Komplexität oder das fertige Produkt genannt. In diesem konkreten Fall ist das der Prozess der testgetriebenen Entwicklung.

Des Weiteren muss der *Zweck* der Studie festgelegt werden. Dies kann den Einfluss oder die Charakterisierung eines oder mehrerer Faktoren sein, hier wird es, wie oben bereits beschrieben, die Analyse des Entwicklungsprozesses sein.

Der wichtigste Aspekt der GQM-Methode ist es, den *Qualitätsfokus*, also den Fokus des Experiments genau zu bestimmen. Dies sind typischerweise die Effektivität, die Kosten, die Zuverlässigkeit, die Sicherheit, die Wartbarkeit, die Performance oder wie hier die Anwendbarkeit des Messobjekts.

Daneben gilt es, den *Blickwinkel* zu definieren und zu beschreiben, da auch dieser sicherlich Einfluss auf das Ergebnis des Experiments haben wird. Dabei stehen der Entwickler, der Projektleiter, der Kunde oder eventuell auch der Forscher zur Auswahl. In diesem Fall wird die Anwendbarkeit aus Sicht des Programmierers betrachtet, da sich konkrete Entwicklungsprozesse auf ihn am stärksten auswirken.

Schließlich ist es wichtig, den *Kontext* der Studie genau festzuhalten. Dabei sind sowohl die handelnden Subjekte als auch die beobachteten und erstellten Objekte, also beispielsweise der Quelltext, zu definieren. Subjekte können durch den Kenntnisstand, die Gruppengröße oder den Ausbildungsweg beschrieben werden, Objekte durch Größe, Komplexität, Qualität oder Priorität. Die genaue Beschreibung der Personen und Rahmenbedingungen entscheidet sowohl über die Reproduzierbarkeit eines Experiments als auch über deren Ergebnis. In dieser Arbeit soll das in Kapitel 2 dargestellte Forschungsumfeld des Fraunhofer IIS vorausgesetzt werden.

Zusammengefasst findet sich dies in Tabelle 4.1.

Messobjekt	Zweck	Qualitätsfokus	Blickwinkel	Kontext
testgetriebene Entwicklung	Analyse	Anwendbarkeit	Entwickler	Forschungsumfeld Fraunhofer IIS

Tabelle 4.1.: Ziel der Studie in GQM-Schreibweise

4.3. Abstraktion

Bei der GQM-Methode wird ein Abstraktionsverfahren vorgeschlagen [8], welches ermöglicht, Einflussfaktoren und Qualitätsfaktoren voneinander zu trennen und mögliche Gefahren für die Gültigkeit zu beschreiben. Danach sieht das generelle Vorgehen folgendermaßen aus: Zuerst werden Qualitätsfaktoren aufgestellt, die Einfluss auf den zu untersuchenden Qualitätsfokus besitzen. Anschließend wird der aktuelle Zustand der Qualitätsfaktoren in den Basishypothesen beschrieben. Es folgt die Darstellung, welche Einflüsse auf diese Basishypothesen möglich sind, so genannte Einflusshypothesen. Abschließend werden die daraus gewonnenen Einflussfaktoren dargestellt. Diese vier Schritte, von Qualitätsfaktoren, über Basishypothesen zu Einflusshypothesen und Einflussfaktoren wird dann auf den „Abstraktionskarten“, schematisch in Tabelle 4.2 zu sehen, zusammengefasst. In der Literatur werden der Qualitätsfokus auch als unabhängige Variable, die Qualitätsfaktoren als abhängige Variablen und die Einflussfaktoren als Störvariablen bezeichnet.

Qualitätsfaktoren Was beeinflusst den Qualitätsfokus?	**Einflussfaktoren** Welche Faktoren beeinflussen die Qualitätsfaktoren und damit den Qualitätsfokus?
Basishypothesen Wie sieht der aktuelle Stand der Qualitätsfaktoren aus?	**Einflusshypothesen** Was hat Einfluss auf die Basishypothesen?

Tabelle 4.2.: Schema einer Abstraktionskarte

Nachfolgend wird eine Abstraktionskarte für den oben genannten Qualitätsfokus „Anwendbarkeit testgetriebener Entwicklung“ ausgearbeitet.

Da in diesem Fall der zu untersuchende Fokus die Anwendbarkeit ist, werden drei Faktoren festgelegt, die die Anwendbarkeit beeinflussen. Diese sind Einarbeitungszeit, kontinuierlicher Zeitaufwand und Anforderungen an den Entwickler.

Einarbeitungszeit Je höher die Einarbeitungszeit, desto schlechter ist die Anwendbarkeit auf eine Entwicklergruppe mit hoher Fluktuation. Die Einarbeitungszeit kann

daher auch als einfaches Maß für die Lernkurve einer Methode gesehen werden. Gleichzeitig ist die benötigte Einarbeitungszeit ein Maß für das benötigte Vorwissen der Entwickler. Je mehr Vorwissen verlangt wird, desto höher ist die Einarbeitungszeit, wenn man von normal qualifizierten Mitarbeitern ausgeht.

kontinuierlicher Zeitaufwand Ein Prozess, der große Zeitaufwände neben der Implementierung erfordert, ist nicht anwendbar, da die Akzeptanz eines solchen Prozesses nicht gegeben wäre. Daher muss sich der zusätzliche Zeitaufwand in einem erträglichen Rahmen bewegen, der in dieser Arbeit mit 25% festgelegt ist. Hier wird in Betracht gezogen, dass ein Prozess, der hohen Dokumentationsaufwand oder administrativen Aufwand aufweist, nicht gut für das Forschungsumfeld geeignet ist, da lediglich Prototypen entwickelt werden.

Anforderungen an Entwickler Je mehr Disziplin und Konzentration die Methode selbst benötigt, desto weniger Aufmerksamkeit bleibt für die eigentliche Arbeit. Daraus ergibt sich dann, dass Methoden und Prozesse, die viel Disziplin benötigen, keine gute Anwendbarkeit im Forschungsumfeld besitzen. Dies ist auch ein Grund, warum klassische schwergewichtige Prozesse des Software-Engineering wie das V-Modell oder das Wasserfallmodell bisher keine Anwendung innerhalb des hier beschriebenen Umfelds gefunden haben.

Die Qualität des Quelltextes stellt keinen Freiheitsgrad dar. Wie oben erwähnt, ist ein Prozess nicht anwendbar, wenn der resultierende Quelltext nicht von guter Qualität ist, sondern die Anwendbarkeit, also geringe Einarbeitungszeit und geringen Anforderungen an Entwickler nur durch geringe Qualität ermöglicht wurden. Daher muss die Qualität der erstellten Software gemessen werden, diese sollte sich aber nicht in großem Umfang verändern. Falls sie verbessert wird, ist dies für weitere Studien, die die Effektivität testgetriebener Entwicklung zum Inhalt haben, interessant, jedoch nicht für die reine Anwendbarkeit, die hier untersucht werden soll.

Nach dieser Festlegung der Qualitätsfaktoren wird deren aktueller Stand beschrieben.

Einarbeitungszeit Aktuell entwickelt jeder nach seinem Erfahrungsschatz und Vorwissen, daher ist es für neue Mitarbeiter schwer, herauszufinden, welcher Entwicklungsprozess für die Projektanforderungen erforderlich ist. Das Vorwissen ist innerhalb der Mitarbeiter sehr unterschiedlich verteilt, wie in Kapitel 2.2 dargelegt.

kontinuierlicher Zeitaufwand Da nur die Anwendbarkeit der testgetriebenen Entwicklung im Fokus steht, wird hier lediglich dessen Zeitverteilung betrachtet.

Anforderungen an Entwickler Da kein einheitlicher Prozess existiert, und daher auch Dokumentation und Administration der Softwareentwicklung jedem persönlich überlassen wird, gibt es neben funktionierenden Programmen keine festen Anforderungen an die Disziplin der Entwickler.

Aus diesen Hypothesen lassen sich Einflusshypothesen ableiten, die folgendermaßen lauten:

Je mehr Vorwissen, desto geringer die Einarbeitungszeit Je mehr der einzelne Entwickler über Tests, Testframeworks, Prozesse und Methoden weiß, desto eher kann er testgetriebene Entwicklung sofort einsetzen. Im Idealfall ist ihm „eXtreme Programming" mit den dazugehörigen Werten, Prinzipien und Praktiken bereits bekannt, sodass er ohne Einarbeitungszeit testgetriebene Entwicklung anwenden kann. Gleichzeitig ist mit Vorwissen auch das Wissen aus der Anwendungsdomäne zu bezeichnen. Je besser ein Entwickler bereits in ein Szenario eingearbeitet ist, desto höher sein Vorwissen und desto weniger Einarbeitungszeit fällt an. Außerdem wird auch das Wissen um eine Softwarestruktur als Vorwissen bezeichnet. Je besser ein Entwickler bereits mit einem Teil der Software vertraut ist, desto besser kann er auch neue Funktionalität hinzufügen.

Je mehr Disziplin, desto geringer die Anforderungen Mit je mehr Disziplin ein Mitarbeiter an die Entwicklung herangeht, desto geringer erscheinen ihm die Anforderungen, die die Entwicklungsmethode an ihn stellt. Er wird aufgrund der Disziplin administrative und dokumentarische Pflichten als selbstverständlich erachten und nicht als große Anforderung an ihn selbst sehen.

Aus den Einflusshypothesen lassen sich jetzt leicht die Einflussfaktoren, Vorwissen und Disziplin der Entwickler, schlussfolgern.

- Vorwissen
- Disziplin

Die Kontrolle dieser Einflussfaktoren gilt es zu erreichen, um Änderungen am Qualitätsfokus vor allem auf Änderungen an den Qualitätsfaktoren zurückführen zu können.

4.4. Gültigkeitsbetrachtung

Zur Feststellung der Gültigkeit des Ergebnisses beschreibt Prechelt in „Kontrollierte Experimente in der Softwaretechnik" [18] vier Kategorien, die es zu überprüfen gilt. Diese sind innere und äußere Gültigkeit sowie Konstruktgültigkeit und Schlussfolgerungsgültigkeit, wie in Abbildung 4.1 gezeigt.

Innere Gültigkeit Die innere Gültigkeit beschreibt den Grad, inwieweit Änderungen am Qualitätsfokus tatsächlich auf Änderungen an den Qualitätsfaktoren und nicht auf Änderungen an den Einflussfaktoren zurückzuführen sind. Sie sind in Abbildung 4.1 mit Kreis 2 markiert.

Äußere Gültigkeit Die äußere Gültigkeit fasst Auswirkungen zusammen, welche die Übertragbarkeit des Experiments auf andere Anwendungsfälle beeinflussen. Dabei

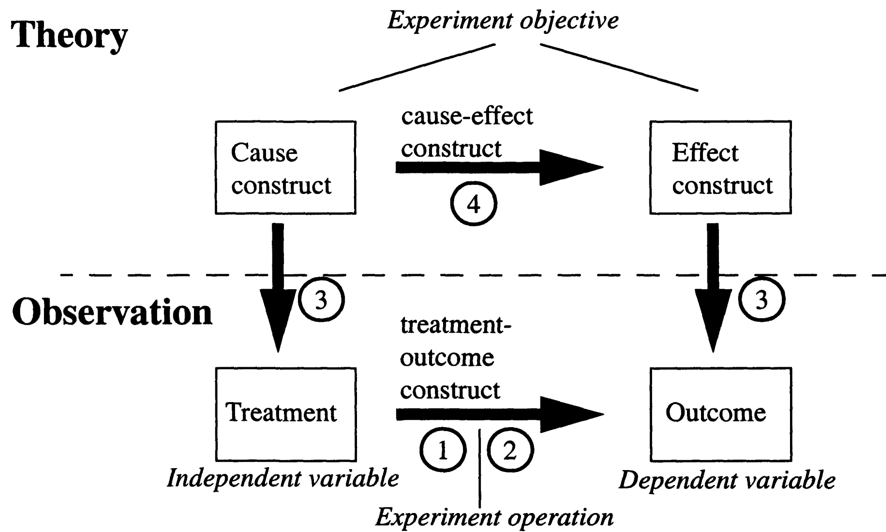

Abbildung 4.1.: Vier Gültigkeitskategorien [10]

handelt es sich beispielsweise um Motivation, Kenntnisstand, Art und Größe der Software bis hin zum Nervenzustand, Termindruck und Ähnliches. Diese sind in Abbildung 4.1 unter Kreis 4 zu finden

Konstruktgültigkeit Die Konstruktgültigkeit beantwortet die Frage, inwiefern der Experimentaufbau tatsächlich die Fragestellung widerspiegelt. Diese ist bedroht, falls Faktoren in das Experiment mit einbezogen werden, die keinerlei ursächlichen Effekt auf den Qualitätsfokus besitzen. Diese in Abbildung 4.1 unter Kreis 3 zu finden.

Schlussfolgerungsgültigkeit Mit der Schlussfolgerungsgültigkeit wird die statistische Verarbeitung und Interpretation der gewonnenen Daten beschrieben. Falls statistische Verfahren nicht beherrscht werden, so sind Fehler dabei in dieser Kategorie einzuordnen. Diese ist in Abbildung 4.1 mit Kreis 1 markiert.

Während des Experiments gilt es, besonders die Faktoren so gering wie möglich zu halten, welche die innere Gültigkeit gefährden können.

4.5. Ausarbeitung

Zur Minimierung der Einflussfaktoren auf die Qualitätsfaktoren wird folgender Ansatz während des Experiments verfolgt. Es werden zwei Entwickler I und II zufällig ausgewählt, die jeweils nach einer kurzen Einführung in testgetriebene Entwicklung ihre

Entwicklung an dem Programmpaket, in das sie schon zuvor eingearbeitet waren, mit testgetriebener Entwicklung fortführen. Dieser Ansatz wird als „natürliches Experiment“ [18] bezeichnet, da durch möglichst geringes Eingreifen ein hoher Realitätsgrad hergestellt wird. Da in dem Umfeld nicht dauerhaft Implementierungsarbeiten geschehen, ist das Experiment auf eine Dauer von einer Woche ausgelegt.

Dieser Ansatz minimiert Auswirkungen, die durch die Einarbeitung in ein anderes Programmpaket entstanden wären. Daher ist dieser Ansatz, Entwickler bei ihrer bisherigen Aufgabe zu belassen insofern realistisch, da bei Anwendung von testgetriebener Entwicklung in der gesamten Abteilung ebenfalls jeder bei seiner Aufgabe belassen würde. Dies ist daher ein Weg, den Einflussfaktor Vorwissen während des Experiments gering zu halten.

So ist außerdem sichergestellt, dass die Disziplin, der weitere Einflussfaktor, während des Experiments möglichst konstant gehalten wird. Ansonsten könnten Änderungen an den Qualitätsfaktoren auch auf die unterschiedliche Disziplin der Entwickler, und nicht die veränderte Arbeitsweise zurückgeführt werden.

Mit der Durchführung als „natürliches Experiment“ werden also die beiden Einflussfaktoren, Vorwissen und Disziplin, möglichst konstant gehalten und so die innere Gültigkeit erhöht.

Damit ist allerdings auch die Einschränkung verbunden, dass die Aussagen nur jeweils für einen Entwickler getroffen werden können, da ansonsten die Einflussfaktoren nicht mehr als konstant angesehen werden können. Für kontrollierte Experimente, die länger durchgeführt werden, lässt sich diese Beschränkung durch eine zufällige Auswahl und genügend große Stichprobe umgehen, da sich dann Unterschiede zwischen den Entwicklern statistisch gesehen gegenseitig aufheben.

Zur Auswertung wird während des Experiments eine Zeiterfassung durchgeführt. Dabei werden Einarbeitungszeit in den Prozess und die kontinuierliche Zeitverteilung auf Test und Implementierung aufgezeichnet. Eine mögliche Unsicherheit, die an dieser Stelle in Kauf genommen wird, sind Fehler, die den Entwicklern bei der Zeitprotokollierung unterlaufen. Diese können sich beispielsweise aus kurzen Unterbrechnungen der Tätigkeit durch Rückfragen von Kollegen oder kurzfristig anstehenden anderen Aufgaben ergeben. Zusätzlich sind vergleichende Aussagen zur kontinuierlichen Zeitverteilung aufgrund fehlender Datenerfassung beim bisherigen Entwicklungsprozess nicht möglich. Es werden also beispielsweise keine Aussagen angestrebt, ob die testgetriebene Entwicklung schneller oder langsamer ist als die bisherige.

Vergleiche bzgl. der Softwarequalität werden so angelegt, dass diese vor Einführung in die testgetriebene Entwicklung und nach Durchführung des Experiments abgeschätzt wird.

Dazu lassen sich die Metriken Komplexitätsmittel $COMP = \frac{Komplexität}{Klassen}$, Kommentaranteil $COMM = \frac{Kommentarzeilen}{Anweisungen}$, Dublikatsanteil $DUP = \frac{Dublikate}{Anweisungen}$ und Zeilenabdeckung $COV = \frac{ausgeführteZeilen}{Anweisungen}$ verwenden. Da nur eine grobe Abschätzung der Kontinuität in der Softwarequalität Q notwendig ist, wird das Komplexitätsmittel $Q1$ und die drei anderen Metriken in gleich gewichteter kombinierter Form $Q2$ betrachtet.

$$Q1 = COMP$$

$$Q2 = \frac{1}{3} \cdot COMM + \frac{1}{3} \cdot (1 - DUP) + \frac{1}{3} \cdot COV$$

Für weitergehende Software-Qualitätsanalysen sei auf [6] und [20] verwiesen.

Abschließend wird der in Anhang A aufgeführte Fragebogen verwendet. Mit diesem wird der Qualitätsfaktor „Anforderungen“ der testgetriebenen Entwicklung an den Entwickler abgefragt. Dies ist notwendig, da für diesen Qualitätsfaktor keine direkte Metrik existiert. Deswegen muss er über einen Fragebogen in Erfahrung gebracht werden.

Zusammenfassend werden die Entwickler durch das „natürliche Experiment“ in einem Umfeld belassen, das so realistisch wie möglich ist, um die innere Gültigkeit zu erhöhen. Gleichzeitig beschränkt diese Methode die externe Gültigkeit, da das hier vorherrschende Umfeld Einfluss auf die Anwendbarkeit hat. Die Aussagen des Experiments müssen immer im Kontext des Forschungsumfelds am Fraunhofer IIS betrachtet werden.

5. Auswertung des Experiments

5.1. Einarbeitung

Die aufgewendete Zeit für die Einarbeitung in die testgetriebene Entwicklung wird aus der Zeiterfassung der Entwickler entnommen. Die resultierenden Werte finden sich in Tabelle 5.1.

Entwickler	Einarbeitung (%)	Test (%)	Implementierung (%)	Summe
I	0,3h (3,1%)	1,9h (17,7%)	8,6h (79,2%)	10,8h
II	0,2h (3,4%)	0,8h (17,1%)	3,9h (79,5%)	4,9h

Tabelle 5.1.: Zeitmessung zur testgetriebenen Entwicklung

Wie dort zu sehen, ist der Zeitaufwand, der für die Einarbeitung benötigt wird, sehr gering. Damit gewährleistet werden kann, dass die Entwickler die Praktik tatsächlich verstanden hatten, wurde in der Nachbereitung des Experiments auch die Prinzipien und die Unterschiede zwischen ihrer bisherigen Entwicklung und der testgetriebenen Entwicklung abgefragt. Beide Entwickler verstanden die Idee und die Herangehensweise nach dem Experiment. Die geringe Einarbeitungszeit entstand also nicht aus Unkenntnis über den Prozess.

Unter Berücksichtigung dieses Qualitätsfaktors ist die Anwendbarkeit der testgetriebenen Entwicklung also sehr hoch.

Damit erfüllt sich das erste Versprechen, das in den Prinzipien des eXtreme Programming genannt wird: Die Praktiken sind, wie hier am Beispiel der testgetriebenen Entwicklung, tatsächlich für die Entwickler *I* und *II* leicht zu erlernen gewesen, obwohl die bisherige Entwicklung nicht als agile Vorgehensweise beschrieben werden kann.

5.2. Zeitaufwand für Tests

Der Zeitaufwand, der für die Erstellung der Testfälle notwendig ist, befindet sich, wie in Tabelle 5.1 ersichtlich, ebenfalls unter der in Kapitel 4.3 gewählten Schranke. Außerdem

ist im Rahmen der Messungenauigkeit keinerlei Unterschied zwischen den Entwicklern auszumachen.

Eine Unsicherheit, mit der diese Aussage jedoch behaftet ist, stellt die Vollständigkeit der entwickelten Tests dar. Die Zeilenabdeckung bei Entwickler I und II für den während des Experiments entwickelten Quelltext beträgt 69% und 78%. Daher ist es möglich, dass der geringe Zeitaufwand zum Teil aufgrund unvollständiger Tests zu Stande kommt.

Trotzdem kann auch unter diesem Gesichtspunkt von einer Anwendbarkeit der testgetriebenen Entwicklung gesprochen werden, da schon mit dem geringen Mehraufwand eine wesentlich bessere Testabdeckung erreicht wird.

5.3. Anforderungen an die Entwickler

Beide Entwickler antworteten auf die Frage, welcher Aufwand für testgetriebene Entwicklung benötigt werde, dass subjektiv allenfalls ein geringer Unterschied zwischen ihrer bisher angewandten Methode und der testgetriebenen Entwicklung bestehe.

Bei der Frage nach der benötigten Selbstdisziplin sieht das Antwortverhalten hingegen unterschiedlich aus. So schilderte Entwickler I, dass viel Disziplin notwendig sei, um tatsächlich immer den Test vor der Implementierung zu entwickeln und nicht in das übliche Entwicklungsmuster zurückzufallen. Entwickler II dagegen erklärte, dass bei seinen Modulen keine Disziplin vonnöten gewesen sei, da diese leicht testbar und damit Testfälle leicht zu entwickeln gewesen seien. Der Unterschied hier ist sowohl auf die unterschiedlichen Module als auch auf die persönlichen Vorlieben der Entwickler zurückzuführen, wodurch hinsichtlich der Anforderungen, die testgetriebene Entwicklung an Entwickler stellt, keine klare Aussage zur Anwendbarkeit abgeleitet werden kann. Dies ist also ein möglicher Schwachpunkt des Prozesses, da die Anforderungen an den Entwickler von jedem persönlich unterschiedlich belastend wahrgenommen werden, sodass im schlechtesten Fall die Anwendbarkeit durch diesen Qualitätsfaktor verhindert wird.

5.4. Qualität des Quelltextes

Zur Auswertung der in Kaptitel 4.5 entworfenen Qualitätsindikatoren wurde das Werkzeug Sonar [23] und Lcov [14] verwendet. Dabei wurden für jeden Entwickler die Qualitätsindikatoren vor und nach der Durchführung des Experiments erhoben, um grobe Ausschläge in der Softwarequalität feststellen zu können.

Daneben gaben die Entwickler auch selbst Rückmeldung zu den subjektiv merkbaren Unterschieden im Quelltext. Entwickler I gab an, dass sich die Qualität des Quelltextes nur in einem Punkt unterscheide: Durch die Konzentration auf den Test zu Beginn ei-

nes Moduls werde die Aufmerksamkeit auf die Schnittstellen gelenkt, wodurch diese sauberer, also besser durchdacht, entwickelt würden, da man von Anfang an die Testbarkeit von Methoden im Kopf habe. Dies würde dann zu erhöhter Kohäsion der Module und geringerer Kopplung zwischen den Modulen führen. Außerdem machte er auf einen Unterschied bei der Testimplementierung aufmerksam: Bei nach der Implementierung geschriebenen Tests würde er sich meist von der Implementierung leiten lassen, so dass die Tests in die selbe gedankliche Richtung geschrieben würden und zwar viele, aber einfache Tests entworfen werden. Im Gegensatz dazu würde bei testgetriebener Entwicklung eher weniger Tests geschrieben, dafür würden diese bei ihm besser die Randwerte der Eingabeparameter abdecken als bei Tests, die nach der Implementierung geschrieben seien.

Entwickler *II* hingegen gab an, dass für ihn keinerlei Unterschiede in der Qualität des Quelltextes auszumachen seien, die Schwankungen der Indikatoren müssen auf deren Auswahl zurückzuführen sein, bleiben aber im Rahmen der benötigten Genauigkeit.

Die Ergebnisse dieser Erhebung sind in Tabelle 5.2 zusammengefasst.

Entwickler	Qualitätsindikatoren vorher	Qualitätsindikatoren nachher	Einschätzung des Entwicklers
I	$Q1 = 16{,}7$ $Q2 = 48\%$	$Q1 = 16{,}1$ $Q2 = 55\%$	bessere Wiederverwendbarkeit
II	$Q1 = 155{,}8$ $Q2 = 28{,}6\%$	$Q1 = 169{,}4$ $Q2 = 36{,}1\%$	kein Unterschied

Tabelle 5.2.: Qualitätsauswertung

Diese Abschätzung unterstützt also die Vermutung, dass sich die Qualität des Quelltextes während des Experiments nicht wesentlich veränderte. Diese Einschätzung ist natürlich mit einer Unsicherheit behaftet, die aus der Auswahl der Metriken resultiert, aber da nicht die Effektivität testgetriebener Entwicklung, sondern nur die Anwendbarkeit untersucht werden sollte, ist diese Abschätzung zur Bestätigung der Aussagen der Entwickler ausreichend.

Unter Berücksichtigung der drei gewählten Qualitätsfaktoren „Einarbeitungszeit“, „kontinuierlicher Zeitaufwand“ und „Anforderungen an die Entwickler“ kommt das Experiment hinsichtlich der Anwendbarkeit testgetriebener Entwicklung zu den im Folgenden beschriebenen Ergebnissen. Die Einarbeitungszeit ist gering, auch die Testframeworks nach dem xUnit-Muster sind den Anwendern nach leicht zu erlernen. Der kontinuierliche Zeitaufwand ist ebenfalls geringer als vielleicht befürchtet, schwankt aber mit der Konsequenz der Entwickler und der daraus resultierenden Testabdeckung. Daher sind die Anforderungen an die Entwickler hinsichtlich Disziplin der einzige Qualitätsfaktor, der die Anwendbarkeit behindert. Die Softwarequalität konnte während des Experiments

ungefähr konstant gehalten werden, hier ist jedoch eine weitergehende Studie notwendig. Mit der Einschränkung hinsichtlich der Disziplin der Entwickler lässt sich daher folgern, dass testgetriebene Entwicklung auch für das Forschungsumfeld geeignet ist.

6. Weiterführende Untersuchungen

Bei der Entscheidung, einen Entwicklungsprozess wie die testgetriebene Entwicklung einzuführen, kann es helfen, die Erfahrungen anderer Gruppen in Betracht zu ziehen. Daher sollen Ergebnisse anderer Studien hier abschließend vorgestellt werden.

6.1. Quantitative XP-Studie

Rumpe und Schröder stellen in einer quantitativen Untersuchung [21] die Grundlagen des eXtreme Programming dar und führten eine groß angelegte, weltweite Umfrage zu Projekten mit XP-Praktiken durch. Dabei wurden generelle Fragen zur Akzeptanz von XP gestellt. Diese ergaben beispielsweise eine 93%-Ja-Anwort auf die Frage, ob XP wieder verwendet werden sollte, außerdem endeten über 90% der Projekte erfolgreich.

Daneben wurde der Einsatz der einzelnen Praktiken untersucht und herausgefunden, dass „Test“ und „Paar-Programmierung“ bei 18% der Befragten als positivste Faktoren für den Projekterfolg genannt wurden.

Aber auch die Probleme mit den Praktiken wurden untersucht. So gaben 40% der Befragten an, dass sie das Element der Metapher nicht verwendeten und 30% stuften das Fehlen des Kunden vor Ort als Risiko für das Projekt ein. Dies wurde durch eine gewisse Skepsis beim Management bestätigt, ob die Werte von XP mit der Unternehmensphilosophie des Kunden vereinbar wäre. Die meisten Kunden würden es, dem Management nach zu urteilen, nicht gestatten, einen Mitarbeiter ständig innerhalb des Projektteams einzusetzen.

6.2. IBM Fallstudie

Direkt mit der testgetriebenen Entwicklung befasste sich eine aufwändige Studie von Williams et. al. [15, 24], die den Effekt der Defektreduzierung im Zusammenhang mit testgetriebener Entwicklung untersuchte. Dabei wurde die siebte Revision eines Produkts mit einer Neuentwicklung der gleichen Funktionalität verglichen. Das Umfeld bestand aus erfahrenen Entwicklern, die bereits über zehn Jahre Gerätetreiber entwickelten. Während der Entwicklung wurden rund 2400 Testfälle erstellt. Diese halfen, die Zahl der

Fehler während des Systemtests um rund 40% zu reduzieren, vergleichen mit der erfahrenen Gruppe, die bereits seit fünf Jahren Ad-hoc-Testen praktizierte. Das Resultat darf natürlich nur in dem Kontext bei IBM gesehen werden und ist aufgrund statistischer Unsicherheiten in der Herangehensweise als Fallstudie nicht ohne Weiteres übertragbar.

6.3. Kontrollierte Experimente

George und Williams führten einige kontrollierte Experimente [12] mit 24 professionellen Programmierpaaren durch. Die Entwicklung in Java wurde in der einen Gruppe testgetrieben durchgeführt, während die Kontrollgruppe nach dem Wasserfallmodell vorging. Die Resultate der Black-Box-Tests zeigten, dass die erste Gruppe 18% mehr Tests erfolgreich absolvierte, dafür aber auch 16% mehr Zeit benötigte. Eine statistische Analyse legt hier eine gewisse Korrelation zwischen benötigter Zeit und Softwarequalität dar. Andererseits wurde auch beobachtet, das die Kontrollgruppe häufig nicht die vom Wasserfallmodell geforderten Tests durchführte, die testgetriebene Entwicklung also ein Ansatz ist, die Testabdeckung durch Unit-Tests zu erhöhen.

Dieser Gedanke, die erhöhte Testabdeckung zu untersuchen, wurde auch von Erdogamus und Morisio in ihrem kontrollierten Experiment zur Effektivität testgetriebener Entwicklung [9] aufgegriffen. Die am Experiment teilnehmenden Studenten wurden dabei in zwei Gruppen aufgeteilt, die erste programmierte testgetrieben, die zweite erstellte die Testfälle erst nach der Implementierung. Dabei wurde die Gruppen hinsichtlich der resultierenden Softwarequalität und der Produktivität der Entwickler untersucht. Die statistischen Untersuchungen ergaben dabei, dass Programmierer, die testgetrieben entwickelten, im Schnitt mehr Tests entwarfen. Außerdem ließ sich feststellen, dass die Studenten, die mehr Tests entwarfen, auch bessere Softwarequalität erreichten, unabhängig von der Gruppe, in der sie sich befanden. Die erhöhte Softwarequalität resultierte also nur indirekt aus der Entwicklungsstrategie, ließ sich aber statistisch feststellen.

Abschließend führten noch Canfora et. al. ein kontrolliertes Experiment [7] mit erfahrenen Entwicklern durch. Dabei beteiligten sich 28 Entwickler, die mindestens fünf Jahre Erfahrung mit der eingesetzten Sprache Java vorweisen konnten und mindestens ein Jahr im untersuchten Unternehmen arbeiteten. Insgesamt fünf Stunden programmierten die Beteiligten an zwei unterschiedliche Aufgaben, wechselweise mit testgetriebener Entwicklung und mit Tests nach der Implementierung. Die Auswertung der gewonnenen Daten zeigte, dass testgetriebene Entwicklung mehr Zeit benötigt als die Kontrollgruppe, es konnte aber nicht bewiesen werden, dass bei testgetriebener Entwicklung auch eine verbesserte Softwarequalität zu erreichen ist. Dies steht im Widerspruch zu den Ergebnissen von Erdogamus et. al., die Autoren führen dies auf den engen Zeitrahmen zurück, der für das Experiment gesetzt wurde.

7. Zusammenfassung und Ausblick

Zur Antwort auf die zu Beginn gestellte Frage, ob testgetriebene Entwicklung sich für das Forschungsumfeld eigne, soll zuerst ein Zitat stehen, das während des Experiments von einem Entwickler geäußert wurde: Mit testgetriebener Entwicklung „nerven Tests nicht mehr so, da sie direkten Nutzen für die Entwicklung haben und nicht so gefühlt überflüssig sind“. Dies ist natürlich überspitzt formuliert, bringt aber einen Vorteil der testgetriebenen Entwicklung auf den Punkt. Tests werden als integraler Bestandteil der Entwicklung aufgefasst, die einen konkreten Beitrag zum Entwurf und zur Implementierung leisten.

Mit der Umfrage unter den Entwicklern wurde der Bildungsweg und die Kenntnisse der Entwickler hinsichtlich Entwicklungsprozessen dokumentiert. Dabei wurde bestätigt, dass Prozesse bisher nicht eingesetzt werden, das Wissen darum aber teilweise durchaus vorhanden ist. Außerdem wurde die aktuelle Vorgehensweise in der Entwicklung dokumentiert. So werden bei Anforderungen und Entwurf kaum Werkzeuge eingesetzt, auch die schriftliche Anforderungsdokumentation war nur teilweise anzutreffen. Bei der Implementierung lobten die Entwickler den Freiraum, der ihnen gelassen wurde, was sich auch in den eingesetzten Werkzeugen in der ganzen Bandbreite von einfachen Texteditoren bis zu integrierten Entwicklungsumgebungen äußerte. Es bestätigte sich außerdem, dass der Test der Software bisher eher wenig beachtet wird. Entweder wird nur manuell mit Ausgaben die korrekte Funktion der Software überprüft, allenfalls werden kleine Testprogramme geschrieben. Bei der Thematisierung der Fehlersuche stellte sich ebenfalls heraus, dass fast ausschließlich mit Meldungen auf der Standardausgabe gearbeitet wird. Die Dokumentation ist vorhanden, aber aufgrund geringer Wartung und Pflege keinesfalls vollständig und noch dazu schlecht auffindbar.

Bei genauerer Betrachtung ließen sich die Prinzipien der agilen Entwicklung hinsichtlich der organisatorischen Rahmenbedingungen gut mit dem Forschungsumfeld in Einklang bringen. Gerade die Selbstverständlichkeit mit der Änderungen, die während eines Projekts im Forschungsumfeld zwangsweise auftreten, in den Praktiken der agilen Entwicklung eingebettet sind, kann hier überzeugen. Aber auch die anderen Voraussetzungen, die agile Entwicklung erleichtern, wie örtliche Nähe und viel Kommunikation im Gegensatz zu exzessiver Dokumentation, sind im Forschungsumfeld der Normalzustand. Lediglich aufgrund der hohen Fluktuation der studentischen Mitarbeiter wird empfohlen, mit einer eingeschränkten Menge an Praktiken zu beginnen. Dies kann beispielsweise die testgetriebene Entwicklung sein, eine der Kernpraktiken des eXtreme Programming und

gleichzeitig eine Möglichkeit, den Test von Software besser in den Entwicklungsablauf zu integrieren.

Das dazu entwickelte Experiment zur Anwendbarkeit testgetriebener Entwicklung zeigte, dass dieser hinsichtlich Einarbeitung und kontinuierlichem Zeitaufwand nichts entgegensteht. Die oben genannte Möglichkeit der besseren Integration des Tests konnte beobachtet werden. Einzig die Disziplin der Mitarbeiter erschwert die Anwendbarkeit der Praktik, was in einem kontrollierten Experiment untersucht werden könnte. Daneben ist die Frage nach der Effektivität testgetriebener Entwicklung im Besonderen und agiler Softwareentwicklung im Allgemeinen auch nach diesem Experiment für das Forschungsumfeld noch unbeantwortet.

Aufgrund der geringen Zahl an Versuchspersonen sind die Resultate des Experiments nicht zu verallgemeinern, auch hier könnte eine größer angelegte Feldstudie im Rahmen eines kompletten Projekts Klarheit schaffen. Ein weiterer Ansatz für zukünftige Untersuchungen stellt die Qualität des Tests dar. Bisher wurde meist nur die Implementierung für Effektivitätsstudien herangezogen, doch wenn der Softwaretest als integraler Bestandteil angesehen wird, dann sollte auch dessen Qualität betrachtet werden.

A. Anhang: Fragebögen

Fragebogen an Mitarbeiter

Persönliche Daten

- Studiengang und Abschluss
- Projekt
- Zeit bei Fraunhofer Gesellschaft

Aktuelle Softwareentwicklung

- Klassifikation der Entwicklung (Firmware oder Software)?
- Wie gehen Sie bei Ihrer Softwareentwicklung vor? (Stichworte: Anforderungen, Programmierung, Test, Dokumentation)
- Welche Sprache wird eingesetzt (C/C++/Java)?
- Wie werden Fehler vermieden/gefunden/behoben?
- Welche Tools (Bugtracker, CI, . . .) setzen Sie ein?

Entwicklungsmodelle und -methoden

- Welche Entwicklungsmethoden setzen Sie selbst ein?
- Von welchen Entwicklungsmethoden haben Sie schon einmal gehört?
- Welche der folgenden Modelle und Methoden kennen Sie? Wie gut? (0: noch nie gehört - 10: kenne ich sehr gut)
  - Wasserfallmodell
  - Spiralmodell
  - V-Modell
  - RUP

- XP
- Prototyping
- Unit-Testing
- Test-Driven Development
- Feature-Driven Development
- Modelbased Development
- Pair-Programming
- Continuous Integration

Fragebogen zum Experiment

Entwicklungsprozess

- Welche Unterschiede bestehen zwischen Ihrem bisherigen Entwicklungsprozess und testgetriebener Entwicklung?
- Wie hoch ist der Aufwand für testgetriebene Entwicklung? (keiner, etwas, viel, sehr viel)
- Wie viel Selbstdisziplin war nötig? (keine, etwas, viel, sehr viel)
- Erwiesen sich die Testfälle als hilfreich für die Dokumentation des Quelltexts? (nicht hilfreich, etwas hilfreich, hilfreich, sehr hilfreich)

Quelltext

- Bestehen Unterschiede im Quelltext mit und ohne testgetriebener Entwicklung? Wenn ja, welche?
- Bestehen subjektive Unterschiede in der Qualität des Quelltexts? (Wiederverwendbarkeit, Kommentare, Test, Design, . . .)
- Wurden Anforderungen mit testgetriebener Entwicklung klarer? (nein, etwas, ja, sehr)
- Wurden Schnittstellen durch testgetriebene Entwicklung anders entworfen? (nein, etwas, ja, sehr)

Literaturverzeichnis

[1] Basili, V. R.: *Software modeling and measurement: the Goal/Question/Metric paradigm.* CS-TR-2956 UMIACS-TR-92-96, University of Maryland, 1992.

[2] Beck, K.: *Test Driven Development. By Example.* Addison-Wesley, Amsterdam, 2002.

[3] Beck, K. und C. Andres: *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[4] Beck, K., M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries et al.: *Manifesto for Agile Software Development.* The Agile Alliance, 2001.

[5] Bertet, N., E. E. Reyero, A. P. Castro et al.: *MetriC++.* Online: http://forge.isotrol.org/projects/show/org00005-metricpp.

[6] Boehm, B. W., J. R. Brown und M. Lipow: *Quantitative evaluation of software quality.* In: *Proceedings of the 2nd international conference on Software engineering*, Seiten 592–605. IEEE Computer Society Press Los Alamitos, CA, USA, 1976.

[7] Canfora, G., A. Cimitile, F. Garcia, M. Piattini und C. A. Visaggio: *Evaluating advantages of test driven development: a controlled experiment with professionals.* In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, Seite 371. ACM, 2006.

[8] Differding, C., B. Hoisl und C. M. Lott: *Technology Package for the Goal Question Metric Paradigm.* Technischer Bericht Internal Report 281/96, AG Software Engineering, Universität Kaiserslautern, 1996.

[9] Erdogmus, H., M. Morisio und M. Torchiano: *On the effectiveness of the test-first approach to programming.* IEEE Transactions on Software Engineering, Seiten 226–237, 2005.

[10] Freimut, B., T. Punter, S. Biffl und M. Ciolkowski: *State-of-the-Art in Empirical Studies.* Technischer Bericht ViSEK/007/E, ViSEK, 2002.

[11] Gamma, E. und K. Beck: *JUnit.* Online: http://www.junit.org.

[12] George, B. und L. Williams: *A structured experiment of test-driven development.* Information and Software Technology, 46(5):337–342, 2004.

[13] GOOGLE: *Google Test*. Online: http://code.google.com/p/googletest/.

[14] LTP: *LCOV*. Online: http://ltp.sourceforge.net/coverage/lcov.php.

[15] MAXIMILIEN, E. M. und L. WILLIAMS: *Assessing test-driven development at IBM*. In: *Proceedings of the 25th International Conference on Software Engineering*, Seiten 564–569. IEEE Computer Society Washington, DC, USA, 2003.

[16] MCCABE, T. J.: *A complexity measure*. TSE, 2(4):308–320, 1976.

[17] MÜLLER, M. M. und W. F. TICHY: *Case study: extreme programming in a university environment*. In: *Proceedings of the 23rd international conference on Software engineering*, Seiten 537–544. IEEE Computer Society Washington, DC, USA, 2001.

[18] PRECHELT, L.: *Kontrollierte Experimente in der Softwaretechnik*. Springer, 2001.

[19] RIEHLE, D.: *Agile and Open Source Software Development*. Online: http://osr.informatik.uni-erlangen.de/lehre/ws09/auos/, 2009.

[20] ROSENBERG, L. und L. HYATT: *Software Quality Metrics for Object Oriented System Environments*. Technischer Bericht SATC-TR-95-1001, SATC, 1995.

[21] RUMPE, B. und A. SCHRÖDER: *Quantitative Untersuchung des Exteme Programming Prozesses*. Technischer Bericht ViSEK/006/D, ViSEK, 2001.

[22] SLEMBECK, T.: *Evolution und bedingtes Lernen*. Handbuch der evolutorischen Ökonomik, 2, 2001.

[23] SONARSOURCE: *Sonar*. Online: http://sonar.codehaus.org.

[24] WILLIAMS, L., E. M. MAXIMILIEN und M. VOUK: *Test-driven development as a defect-reduction practice*. In: *IEEE International Symposium on Software Reliability Engineering, Denver, CO*, Seiten 34–45. IEEE Computer Society Washington, DC, USA, 2003.