Masterarbeit

Analyse und Evaluierung von Test-getriebener Softwareentwicklung am Beispiel der Entwicklung kommerzieller Software

Patrick Hofmann

29. Oktober 2013

Betreuer: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Technische Universität Graz
Institut für Softwaretechnologie

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Danksagung

Viele standen mir in den unterschiedlichen Phasen meines Studiums und beim Abschluss dieser Arbeit zur Seite. Von meinen Freunden, Bekannten und Familienmitgliedern möchte ich mich vor allem bei jenen bedanken, die mich über die gesamte Strecke dieses Weges begleitet haben, aber natürlich auch bei denjenigen, die mich auf subtile Weise mit kleinen Richtungskorrekturen durch Rat und Tat vom Weg nicht abkommen ließen.

Es liegt in der Natur von Namensaufzählungen zumindest eine Person zu vergessen. Ich werde daher auch nicht versuchen alle Namen, die wahrscheinlich diese ganze Seite füllen könnten, direkten zu nennen. Stattdessen hoffe ich, dass sich alle Personen, die mein Leben durch ihre Unterstützung so sehr bereichern, durch diese Danksagung angesprochen fühlen.

Kurzfassung

Die vorliegende Arbeit gibt dem Leser eine Einführung in die Methodik der Test-getriebenen Softwareentwicklung, evaluiert und analysiert diese auf wissenschaftlicher Basis, wobei gleichzeitig der Zusammenhang zur Wirtschaftlichkeit eine zentrale Rolle spielt. Um dies zu erreichen wird zuerst die Motivation für den Bedarf erläutert und die Technik selbst vorgestellt. Weiters werden dem Leser sowohl unterschiedliche Testarten, Methoden und Metriken vorgestellt als auch Software selbst, die im Zuge der Test-getriebenen Softwareentwicklung ihre Anwendung findet. Nachdem eine Reihe von wissenschaftlichen Studien einschließlich der Ergebnisse vorgestellt und unterschiedliche Projektmanagementmethoden erklärt werden, schließt die Arbeit durch die Analyse eines selbst durchgeführten Experiments (der Programmierung eines Softwareprodukts für den produktiven Einsatz in der Industrie) ab.

Abstract

This thesis starts with an introduction of test driven development, analyses and evaluates it from a scientific point of view yet still tries to keep the connection to business. First of all the motivation for using test driven development as well as the methodology itself is being presented. After discussing several test practices, methodologies and metrics the thesis switches to a more scientific view. Furthermore some project management methods are being introduced and discussed. Finally the thesis ends with the analysis of an experiment (programming a commercial software product) which has been developed by the author.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Ziel und Abgrenzung der Arbeit

Die vorliegende Arbeit soll dem geschätzten Leser eine Einführung in die Methodik der Test-getriebenen Softwareentwicklung geben, sowie diese auf wissenschaftlicher Basis zu evaluieren und zu analysieren. Gleichzeitig spielt der Zusammenhang zu einer wirtschaftlichen Vorgangsweise eine zentrale Rolle. Um dies zu erreichen wird zuerst die Motivation für den Bedarf erläutert und die Technik selbst vorgestellt. Weiters werden dem Leser sowohl unterschiedliche Testarten, Methoden und Metriken präsentiert als auch Software selbst, die im Zuge der Test-getriebenen Softwareentwicklung ihre Anwendung findet. Nachdem eine Reihe von wissenschaftlichen Studien mit den Ergebnissen vorgestellt werden, wird die Arbeit durch die Analyse eines selbst durchgeführten Experiments (u.z. der Programmierung eines Softwareprodukts für den produktiven Einsatz in der Industrie) abgerundet.

1.2 Motivation für Test-getriebene Softwareentwicklung

Test-getriebene Softwareentwicklung suggeriert Tests, Tests und Tests. Warum sollte ein Programmierer oder sogar ganze Teams sich eine Methodik anlernen, die sich hauptsächlich dem Verifizieren widmet anstatt dem Implementieren, was ja eigentlich deren Aufgabe ist?

Um dies zu erklären wird jedem Programmierer eine Tatsache unterstellt - nämlich Angst.

Gemeint ist dabei die Angst, schon von Anfang an nicht das Ende eines schwierigen Problems zu sehen. Angst, dass die Aufgabe nicht klar genug ist, Angst, dass ein Problem zu groß ist, oder Angst, die Aufgabe einfach nicht zufriedenstellend zu lösen.
Dabei ergibt sich aus dieser Angst eine Reihe von unangenehmen Folgen:

- sie lässt einen zögern
- sie bringt Leute dazu weniger zu kommunizieren
- sie zieht weitere Angst vor Feedback nach sich
- sie lässt einen schlecht fühlen

Diese Angst wird überwunden, indem man sich mit dem Problem beschäftigt. Es werden weitere Informationen eingeholt, Diagramme gezeichnet, Klassen oder eben Tests geschrieben.

Die Idee hinter Test-getriebener Softwareentwicklung besteht also darin, durch das Schreiben von Tests diese Angst zu reduzieren. Dabei könnte erneut Angst entstehen, dass Tests für alles und jeden Fall geschrieben werden müssen, und das auch noch im Vorhinein - was das Problem ja noch verschärfen würde.

Grundsätzlich stimmt das sogar, aber der Trick ist folgender: nicht alle Tests müssen geschrieben werden bevor die eigentliche Implementierung durchgeführt wird.

Jede Idee, jede Anforderung, im Prinzip alles was man bereits weiss und zur Umsetzung der Aufgabe hilfreich ist wird in Form eines Testfalls niedergeschrieben. Damit ist der Testfall gegeben. Die Aufgabe ist formuliert - nämlich in einer Weise, die für jeden Programmierer klar verständlich ist.

Funktioniert ein Testfall weiss man, dass man der Lösung des Problems näher gekommen ist - und genau das nimmt die vorher erwähnten Ängste:

- Das Ende einer Aufgabe ist nicht zu erkennen: Testfälle zeigen welche Aufgaben schon erledigt sind und welche noch implementiert werden müssen.
- Eine Aufgabe erscheint nicht klar genug: Testfälle definieren die Aufgabe und decken während der Testfallerstellung Unklarheiten auf.
- Ein Problem ist zu groß: Testfälle geben dem Entwickler die Möglichkeit ein Problem durch mehrere Testfälle zu beschreiben und in kleinen Schritten zu lösen.
- Die Aufgabe wird nicht zufriedenstellend gelöst: Testfälle beschreiben das gewünschte Ergebnis. Ein gelöster Testfall sollte daher auch die Aufgabe zufriedenstellend lösen.

Plötzlich ergibt es Sinn über Probleme zu sprechen, der Softwareentwickler muss nicht mehr zögern - er kann einfach loslegen und nähert sich mit jeder Quellcode-Zeile seinem Ziel.

Die Angst wird dem Programmierer also genommen - doch was passiert mit dem Projektmanager?
Bei ihm entsteht nun eine nicht ganz unbegründete Angst: Braucht das alles nicht länger? Und vor allem - kostet das nicht mehr?

Seit geraumer Zeit wird den Managern das berühmt-berüchtigte Magische Dreieck (siehe Abbildung 1.1[1]) eingebläut: Kosten, Zeit und Qualität arbeiten gegeneinander.

Um dieser Problematik zu begegnen wurde das sogenannte Projektmanagement Rechteck entwickelt, welches eine weitere Größe beschreibt: den Umfang eines Projekts oder eines Produkts. Es veranschaulicht, dass durch verringerten Um-

[1] Siehe http://it-wissenschaft.de/333/magisches-dreieck-kosten-zeit-oder-qualitat/ (zuletzt besucht 27.10.2013)

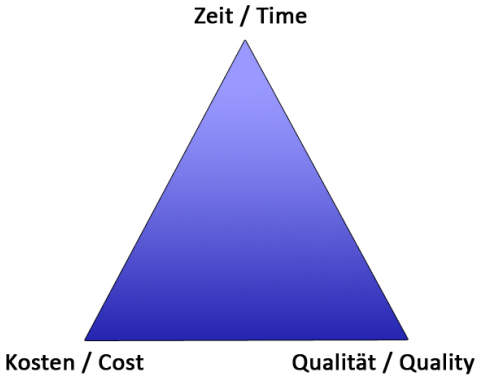

Abbildung 1.1: Magisches Dreieck

fang auch höhere Qualität gehalten werden kann (siehe Abbildung 1.2[2]). Aktuelle agile Softwareentwicklungsmethoden konzentrieren sich daher stärker darauf, den Umfang in kurzen Iterationen zu erweitern um dem Kunden hohe Qualität zu gewährleisten und dabei keine Fehlentwicklungen durchzuführen.

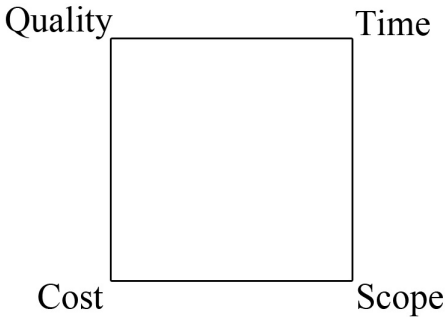

Abbildung 1.2: Projektmanagment Rechteck

Kent Beck geht dabei noch einen Schritt weiter und rät davon ab große Projekte durch lange Listen von Anforderungen (engl. requirements) in einem Vertrag oder Pflichtenheft zu definieren. Dadurch würde sich das Risiko für einen Projektfehlschlag auf beiden Seiten erhöhen - sowohl beim Kunden das falsche Produkt zu bekommen, als auch beim Entwickler zu hohe Kosten tragen zu müssen. Stattdessen sollten Zeit, Kosten und Qualität definiert und der Umfang als über die

[2]Siehe http://betweencode.com/2012/09/business-perspective/ (zuletzt besucht 27.10.2013)

Projektlaufzeit weiter verhandelbarer Punkt offen gelassen werden. Der Umfang kann so durch kleinere Verträge in kürzeren Iterationen ständig erweitert werden.[3]

Die Test-getriebene Softwareentwicklung unterstützt dabei den Auftraggeber zu jedem Zeitpunkt ein funktionales Produkt zu bekommen, dessen Umfang jederzeit erweitert werden kann ohne die Kontrolle über die Kosten zu verlieren.

[3]Siehe Beck (2004) Seite 69

2 Test-getriebene Softwareentwicklung

2.1 Ursprung und Entwicklung von Test-getriebener Softwareentwicklung

Test-getriebene Softwareentwicklung wurde durch den steilen Aufstieg agiler Software-Prozesse schnell in den Vordergrund gerückt - die eigentlichen Wurzeln kommen jedoch aus den evolutionären Prozess-Modellen aus den 1950ern. Heutzutage stehen natürlich bedeutend mehr Tools zur Umsetzung einer schnellen Testumgebung und fortlaufenden Regression-Tests zur Verfügung.

Die Einführung von Tests in die Softwareentwicklung kam schon sehr früh - jedoch mit einem 'Test-last' Ansatz, also der nachträglichen Evaluierung über die Fehlerfreiheit des kompilierten Quellcodes. In den 1980er Jahren entwickelte sich dieses System weiter und die Testfallerstellung wurde auf den Zeitpunkt vor der eigentlich Implementation gerückt. Doch diese Techniken waren, bevor schließlich Kent Beck durch Extreme Programming[1] mehrere Techniken vorstellte um die Softwareentwicklung effizienter zu gestalten, verhältnismäßig unbekannt (auch wenn Beck angab Test-first Methoden bereits als Kind in einem Buch gelesen zu haben).

Schließlich führte Kent Beck sein Werk extremer Programmierung fort und inkludierte Test-first Methoden in dem Buch 'test driven development by example'[2].

[1] Vgl. Beck (2004)
[2] Vgl. Beck (2002)

2.2 Vorgangsweise der Test-getriebenen Softwareentwicklung

Ron Jeffries, einer der drei Mitbegründer von Extreme Programming (XP), sagte es gehe um

> *„Clean code that works“*[3]

Doch was ist 'sauberer Code der funktioniert' und vor allem - warum ist das so wichtig?

Bjarne Stroustrup, Erfinder der Programmiersprache C++ beschreibt 'Clean Code' folgendermaßen:

> *„I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.“*[4]

'Clean Code' weist somit folgende Kriterien auf:

- geradlinig bzw. unkompliziert
- hat wenig Abhängigkeiten
- fehlertolerant
- performant bzw. optimal

[3] Siehe (Beck, 2002) Seite ix
[4] Siehe (Martin, 2008) Seite 7

- erfüllt eine einzige Aufgabe gut

kurzum - elegant und effizient.

Quellcode nach dem 'Clean Code' Prinzip zu erstellen bringt eine Reihe von Vorteilen:

- es ist leichter - sowohl für einen selbst auch auch für andere - mit 'Clean Code' zu arbeiten
- er gibt einem die Möglichkeit während der Entwicklung zu lernen und sich und den Code zu verbessern
- da er funktioniert, erfreuen sich auch die Benutzer der resultierenden Software über 'Clean Code'
- 'Clean Code' ist etwas, worauf man stolz sein kann

Doch wie ist man in der Lage 'Clean Code' zu erzeugen - wenn er doch so hilfreich ist?

Genau hier setzt die Test-getriebene Softwareentwicklung an und setzt dem Softwareentwickler eine überschaubare Anzahl an Regeln:

1. Es wird erst der Programmcode erzeugt, wenn es davor einen fehlschlagenden Test dafür gibt
2. Entferne doppelten Code bzw. Redundanzen

Zwei einfache Regeln, die in der Praxis leider nicht ganz so einfach auszuführen sind, denn

- das Design wird organischer
- jeder muss seine Testfälle selbst schreiben um produktiv weiterarbeiten zu können

- die Entwicklungsumgebung muss schnell auf Änderungen reagieren können
- das Design wird durch die Testfälle automatisch Komponenten-zentrierter

Die zwei Regeln drängen den Programmierer daher in ein sich immer wieder wiederholendes Schema (siehe Abbildung 2.1[5])

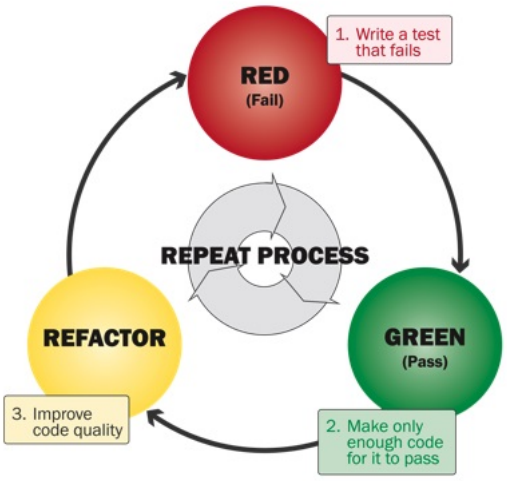

Abbildung 2.1: Red-Green-Refactor

1. ROT: schreibe einen Test der noch nicht funktioniert - gegebenenfalls sogar noch nicht einmal kompiliert
2. GRÜN: lasse den Test so schnell wie möglich funktionieren
3. REFACTOR: Entferne alle Doppelgleisigkeiten bzw. Redundanzen, die notwendig waren um den Test zum Laufen zu bringen

Unter der Prämisse, dass diese Art zu Programmieren durchführbar ist, existiert zu jederzeit nur Programmcode, dem ein Testfall gegenübersteht. Und gemäß dem 'Clean-Code' Prinzip sollte im Code auch nichts anderes passieren als das, was durch Testfälle abgedeckt ist.

[5]Siehe http://www.pathfindersolns.com/resources/industry-glossary/tdd (zuletzt besucht 27.10.2013)

Nicht mehr und nicht weniger.

Natürlich bleibt man auch weiterhin nicht gegen Fehler, Änderungswünsche oder neue Funktionalitäten gefeit. Doch das Vorgehen ist klar definiert:

- Fehler? Testfall!
- Änderungswunsch? Testfall!
- neues Feature? Testfall!

Auch wenn diese Vorgehensweise aufwändig klingt, die Versprechen von Test-getriebener Softwareentwicklung sind verlockend:[6]

- die kompilierte Software ist jederzeit funktional und soweit bekannt fehlerfrei
- die Qualitätssicherung kann sich auf neue Fehler konzentrieren anstatt immer wieder dieselben Fehler zu reporten
- die Software kann in kürzeren Abständen dem Kunden übergeben werden - was die Entwicklung in die richtige Richtung beschleunigt
- die Projektmanagement-Zyklen können in kürzeren Abständen erfolgen, daher können auch Aufgaben besser verteilt werden und die Kommunikation zwischen den Programmierern wird erhöht

[6] Vgl. Beck (2002)

2.3 Vorteile aus der Anwendung von Test-getriebener Softwareentwicklung

Der Test-getriebene Softwareentwicklungsansatz bietet neben den bereits beschriebenen Vorteilen (siehe Kapitel 1.2) auch eine Reihe von qualitätssichernden Elementen, die sich aus der Tatsache ergeben, dass Tests vor der Implementation geschrieben werden.[7]

- Tests werden überhaupt erst geschrieben

Für Programmierer ist es häufig eine unangenehme Last Testfälle zu schreiben - daher versuchen viele sie zu meiden. Durch die Tatsache, dass die Test-getriebene Softwareentwicklung den Test vor der eigentlichen Implementation zu schreiben verlangt, erhöht sich die Anzahl der implementierten Tests.

- Tests sind eine ergänzende Dokumentation des Quellcodes

Jeder Test zeigt die Verwendung einer Komponente, einer Funktion oder sogar eines ganzen Systems. Ein Entwickler, der bestehenden Quellcode bearbeitet, hat durch geschriebene Testfälle die Möglichkeit, die Intention des Autors eher zu begreifen.

Gleichermaßen kann sich ein Programmierer wahrscheinlich durch Testfälle eine Reihe von Kommentaren im Quelltext ersparen, da die Verwendung der implementierten Komponente leichter verständlich wird. Natürlich kann und soll ein Testfall nicht wichtige Hinweise in Kommentaren ersetzen, sondern kann sie als Anwendungsbeispiel ergänzen.

- Testfälle sind eher richtig

Da Testfälle ihrem Wesen nach eher einfacher aufgebaut sind als die Implementation des zu testenden Objekts, entstehen an dieser Stelle tendenziell weniger Fehler.

[7] Vgl. (Beck, 2002)

- Testfälle sind überprüfbar

In der Test-getriebenen Softwareentwicklung muss jeder erstellte Testfall zunächst einmal fehlschlagen - eventuell sogar nicht einmal kompilieren. Ein im Nachhinein erstellter Testfall, der sofort funktioniert, kann die Funktionalität nicht ausreichend sicherstellen, da erst die Veränderung der Implementation und daher der Fehlschlag des Tests seine wahre Prüfroutine validieren könnte. Diese Vorgehensweise wäre im Vergleich mit jener, den Testfall zuerst zu schreiben, sehr aufwändig. Eine Maßnahme diesen Aufwand zu reduzieren wäre das Mutation Testing (siehe Kapitel 2.4.12).

2.4 Unterschiedliche Test-Methoden erklärt

Während das vorhergehende Kapitel sich dem Prozess 'Test-getriebene Softwareentwicklung' widmete, gilt es nun einen Überblick über die unterschiedlichen Test-Methoden selbst zu bekommen. Abbildung 2.2 zeigt die klassische Testpyramide, die Aufschluss über die Abdeckungsraten der unterschiedlichen Testarten geben soll.[8]

2.4.1 Modultests

Bei Modultests (engl. Unit-Tests) wird ein Modul bzw. eine Komponente auf seine fehlerfreie Funktionalität anhand einer Spezifikation getestet. Dabei orientiert sich ein solcher Test üblicherweise an einem einheitlichen Aufbau:

1. Initialisierung der Komponente bzw. des Moduls
2. Anwenden von Operationen
3. Vergleich von IST- und SOLL-Zuständen bzw. Ergebnissen

[8] Vgl. Martin (2011) Seite 153

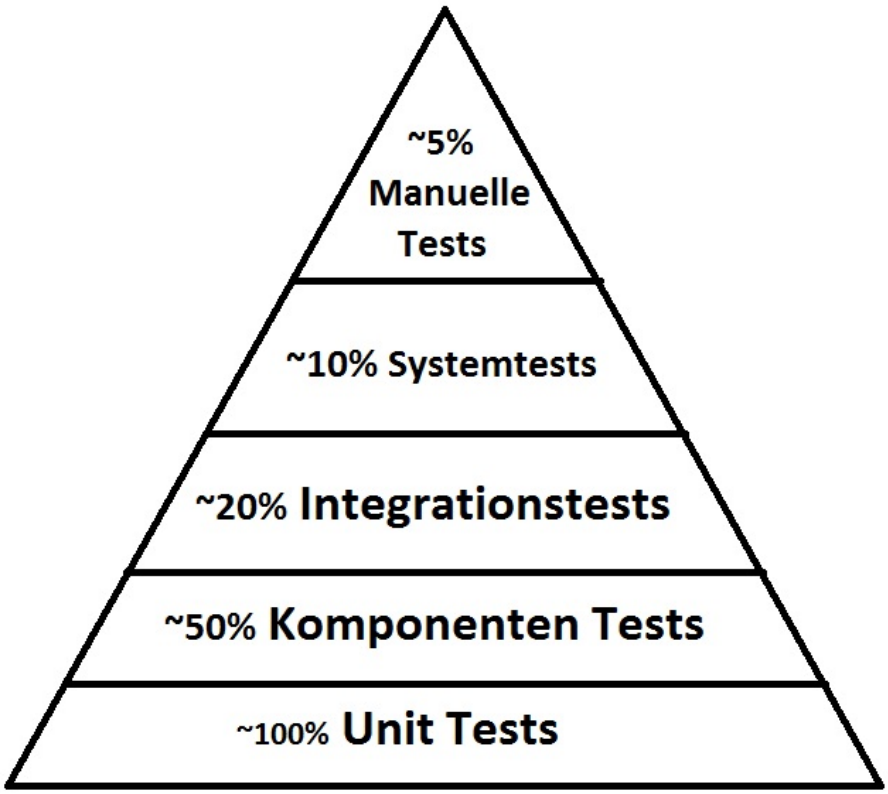

Abbildung 2.2: Test Pyramide

Typischerweise stellt Unit-Test Software Funktionen zur Verfügung um IST-Werte mit SOLL-Werten zu vergleichen bzw. Zustände zu verifizieren. Diese Funktionen sind als ASSERT-Methoden bekannt.

Modultests stellen die Vorstufe zu den Integrationstests dar, die im folgenden Abschnitt erläutert werden.

2.4.2 Integrationstests

Integrationstest (im Englischen häufig mit I&T abgekürzt) beschreibt eine Testphase nach den Modul- bzw. Komponententests. Dabei werden die bereits getesteten Komponenten und Module in ihrem Zusammenspiel getestet. Da dies aus

ökonomischen Gründen häufig nicht für jede Kombination von Modulen möglich ist, beschränken sich solche Tests meist auf einzelne Testszenarien, die in Testplänen festgehalten werden.

Die Integrationstests können demnach als Black-Box-Tests für einzelne Subsysteme beschrieben werden.

Die Literatur unterscheidet drei Herangehensweisen für den Ablauf und die Implementierung von Integrationstests:

1. Big-Bang
2. Top-down
3. Bottom-up

Bei Big-Bang wird versucht alle bzw. zumindest die meisten der Komponenten in ihrem Zusammenspiel zu testen. Bei guter Planung kann dies der effizienteste und umfassendste Weg sein - bei schlechter Planung jedoch den ganzen Testprozess verlangsamen.

Beim Top-down Ansatz werden die obersten integrierten Module zuerst getestet und in jedem weiteren Schritt die dafür verwendeten Sub-Module bzw. Komponenten getestet. Dieser Ansatz hat den Vorteil fehlende Verzweigungen aufdecken zu können.

Beim Bottom-up Ansatz wird zuerst das Zusammenspiel von Submodulen getestet bevor das Zusammenspiel der integrierten Submodule getestet wird. Dieser Ansatz erleichtert das Finden von Fehlern.

Wie so häufig, gibt es auch hier einen Ansatz zwischen Botton-up und Top-down genannt Sandwich Testing, der hier aber nicht näher ausgeführt wird.

2.4.3 Systemtests

Beim Systemtest geht es darum das ganze System gegen alle funktionale und nicht funktionale Anforderungen zu testen. Hierfür wird typischerweise eine Testumgebung eingerichtet und mit Hilfe von Testdaten eine Reihe von Tests durchgeführt. Dabei soll die eingerichtete Testumgebung so gut als möglich der Umgebung des End-Users entsprechen. Im Gegensatz zum Akzeptanztest werden die Systemtests üblicherweise direkt durch den Entwickler durchgeführt.
Zu den während des Systemtests durchgeführten Testarten gehören unter anderem:

- GUI-Tests
- Load-Tests
- Stress-Tests
- Performance-Tests
- Regression-Tests

2.4.4 Regressionstests

Regressionstests sind Tests, die aufgrund von gefundenen Fehlern geschrieben werden. Sie sollen sicherstellen, dass ein bereits aufgetretener Fehler in einer zukünftigen Versionen des Programms behoben wird und anschließend auch behoben bleibt.
Die Wichtigkeit von Regressionstests zeigt sich unter anderem bei Patches zur Fehlerbehebung in bereits ausgelieferten Softwareprodukten. Hier kann es ohne solche Tests vorkommen, dass ein Fehler in der ausgelieferten Version behoben wurde, in folgenden Versionen jedoch auf die Integration der entsprechenden Fehlerbehebung vergessen wird. Der erneut durchgeführte Regressionstest zeigt diesen Fehler auf und soll einen Rückschritt der Software verhindern.

2.4.5 Manuelle Tests

Manuelle Tests bezeichnen den Prozess, Software nicht automatisiert auf Fehler zu testen. Dabei nimmt ein Tester die Rolle eines End-Users ein und versucht alle Funktionen einer Applikation auf ihr korrektes Verhalten zu überprüfen.

Um manuelle Tests nachvollziehbar zu halten, folgen sie häufig einem niedergeschriebenem Testplan, welcher selbst auch gewöhnlicherweise mit Prioritäten versehen ist.

2.4.6 Akzeptanztests

Beim Akzeptanztest wird das gesamte System durch den Auftraggeber bzw. den Kunden getestet. Er ist ebenfalls unter den Namen Abnahmetest oder Verfahrenstest bekannt.

Der Akzeptanztest ist häufig die rechtliche Grundlage für die erfolgreiche Abnahme von Software durch den Auftraggeber. Diese Form des Tests wird meist bereits auf einem System ausgeführt, das dem produktiven Einsatz beim Kunden entspricht sowie mit Test-Daten durchgeführt, die dem Entwickler im Voraus noch nicht bekannt waren um ein Overfitting (dem Abstimmen des Programms auf bekannte Test-Daten) zu vermeiden.

Der Ablauf solcher Tests entspricht typischerweise einem Black-Box-Verfahren. Getestet wird das kompilierte Programm selbst, anstatt des dahinter stehenden Quellcodes. Je nach Möglichkeit kommen hier sowohl manuelle Tests, als auch GUI-Tests, sowie skriptbasierte Testszenarien zum Einsatz.

Akzeptanztests werden jedoch nicht nur auf die finale Software durchgeführt, sondern zB. auch bei Milestone-basierten Projekten auf den vereinbarten Zustand der Software zu einem bestimmten Zeitpunkt. Dies ist häufig ein Mittel zur Evaluierung, ob ein Softwareprojekt in die nächste Phase der Entwicklung fortschreiten kann.

2.4.7 Usability Tests

Der Usability Test ist eine Technik zur empirischen Softwareevaluierung. Dabei wird der Umgang von potenziellen Benutzern mit der Software überprüft. Es wird also weniger die Funktionalität des Produkts evaluiert, sondern die Interaktivität des Benutzers mit der Software.

Hierfür gibt es unterschiedliche Möglichkeiten:

- Offline-Usability Tests
- Online-Usability Tests
- Experten-Analyse

Bei Offline Usability Tests erhalten potenzielle Benutzer Konzepte oder Designs der Software zusammen mit Aufgabenstellungen zur Lösung eines Szenarios. Die Reaktionen des Benutzers werden dabei aufgezeichnet und anschließend evaluiert.

Beim Online Usability Test arbeiten potenzielle Benutzer bereits mit der Software selbst. Diese muss nicht zwangsläufig vollständig funktional sein, sondern zumindest die vorgelegte Aufgabenstellung lösen können. Die Evaluierung kann sowohl durch Befragungen, als auch durch Messungen erfolgen - zB. wie lange braucht es durchschnittlich eine Aufgabe zu erfüllen oder in wievielen Schritten wird die Aufgabe durchschnittlich erfüllt.

Die Experten-Analyse ist hingegen ein wissenschaftlicher Ansatz, der die Software nach bekannten Gebrauchsmustern und Verhaltensstudien evaluiert.

Welche und ob Usability Tests durchgeführt werden, hängt stark vom Einsatzgebiet der Software ab. So wird zum Beispiel die Benutzung eines Betriebssystems für Mobiltelefone stärker einer solchen Analyse unterzogen als die Software für ein wissenschaftliches Expertensystem.

2.4.8 Smoke-Tests

Als Smoke-Tests werden schnelle, oberflächliche Tests bezeichnet, die meist von den Entwicklern selbst vor einem Build (oder gar Release) oder eben als Vortest durch Tester durchgeführt werden.

Dabei geht es um eine Sammlung von Tests, um die grundsätzliche Funktionalität der Software zu überprüfen. Erst bei Bestehen dieser Tests wird die Software für weiterführende Tests akzeptiert. Beispielhafte Fragen hierfür sind:

- Werden alle Modultests bestanden?
- Startet die Software?
- Stürzt die Software durch wahlloses Drücken von Buttons ab?

Aus diesem Grund wird der Smoke-Test auch häufig Build-Verification-Test genannt, da er die grundsätzliche Funktionalität des Builds überprüft. Solche Tests können auch von anderen Entwicklern durchgeführt werden, bevor neuer Quellcode in ein offizielles Repository gelangt.

2.4.9 Load- und Stress- Tests

Die Idee hinter den Load Tests besteht darin, das System oder eine Komponente auf den 'extremen' Einsatz hin zu testen. Dabei können Bottlenecks (verzögernde Komponenten oder Funktionen) bzw. die Stabilität des Systems bei hochfrequentierter Nutzung erkannt werden.

Diese Form ähnelt daher sehr dem Stress-Test bzw. geht grenzenlos in diesen über und steht der spezifizierten und gezielten Durchführung von DOS- (Denial of Service) Attacke gegenüber. In den meisten Fällen möchten Entwickler durch diese Form des Tests gewährleisten, dass ein System auch bei unerwartet hoher Benutzerzahl noch immer reaktionsfähig bleibt.

2.4.10 Rechnernetz-Test

Bei Rechnernetz-Tests handelt es sich um Testmethoden, die das korrekte Interagieren von Software in verteilten Systemen evaluiert.

In den meisten Fällen handelt es sich hier um durch ein Netzwerk verbundene Systeme, wobei Latenz, Integrität, Priorität und ähnliche Parameter der Datenkommunikation bzw. deren Datenpakete verifiziert werden müssen. Vor allem in den immer stärker verwendeten Cloud-Services spielt diese Form des Testens eine größer werdende Rolle.

Rechnernetz-Tests werden häufig in Kombination mit Stress-Tests durchgeführt um Grenzfälle in der Kommunikation von verteilten Systemen zu evaluieren und zu analysieren.

2.4.11 Monkey Tests

Unter Monkey Testing versteht man die Methode, Eingaben mit relativ einfachen Mitteln zu simulieren und die grundsätzliche Verarbeitbarkeit zu verifizieren.
Es werden 4 Arten von Monkey Tests unterschieden:

1. Monkey Button Push Testing
2. Smart Monkey Testing
3. Brilliant Monkey Testing
4. Dumb Monkey Testing

Monkey Button Push Testing soll die sich wiederholenden Eingaben von Benutzern erleichtern. Dabei werden Eingaben durchgeführt und das Ergebnis überprüft - ähnlich dem Affen, der beim Drücken des roten Knopfes seine Banane bekommt.

Das Smart Monkey Testing automatisiert Benutzereingaben anhand statistischer

Wahrscheinlichkeiten, die z.B. durch Benutzerprofile erhoben wurden. Je nach IQ-Level des Tests (dh. wie intelligent dieser Test gestaltet sein soll), werden die Tests als einzelne Eingaben unabhängig voneinander gewertet und analysiert oder bei höherem IQ-Level die Korrelationen der Eingaben in Betracht gezogen.

Das Brilliant Monkey Testing wird auf Basis einer State-Machine durchgeführt. Es soll relevante oder irrelevante Eingaben von Benutzern durchführen. In jedem Zustand der State-Machine hat das durchführende Test-Programm eine Reihe von Möglichkeiten Eingaben zu tätigen und das zu testende Programm auf sein korrektes Verhalten (bzw. seinen korrekten Zustand) zu analysieren.

Dumb Monkey Testing ist das wahllose Drücken von Knöpfen oder Tätigen anderer Eingaben. Es wird häufig als Stress-Test einer Anwendung benutzt.

2.4.12 Mutation Testing

Mutation Testing (oft auch Mutationsanalyse oder Programmmutation genannt) ist eine Methode um die Qualität der bestehenden Softwaretests zu evaluieren. Dabei werden kleine Veränderungen am Quellcode durchgeführt, die als typische Fehler angesehen werden. Darunter fällt das Verwechseln von logischen Operatoren oder das Verwenden von falschen Variablen in Programm-Schleifen.

Ziel dieser Testform sollte nun sein, dass die bestehenden Testfälle diese Veränderungen als Fehler erkennen - ansonsten ist die Software nicht ausreichend getestet und die Qualität des Produkts wurde nicht ausreichend verifiziert.

Wie bei allen Tests obliegt es natürlich auch hier dem gesunde Menschenverstand, die Sinnhaftigkeit eines weiteren bzw. erweiterten Tests zu erkennen und daher zu implementieren, obwohl der Fehler ja scheinbar nicht im System bestand.
Trotzdem kann es sinnvoll sein auch solche Tests einzuführen, da sie eine gewisse Sicherheit und Qualität der ausgelieferten Software versprechen und sich spätestens beim Refactoring die nächsten Fehler einschleichen könnten.

Zur Durchführung solcher Tests braucht es einen eigenen Typ von Test-Software, zu dem unter anderem das Open Source Programm PITest[9] von Alexandre Victoor gehört.

[9]Siehe http://pitest.org (zuletzt besucht 27.10.2013)

3 Metriken und Methoden

Bisher wurde auf den Prozess der Test-getriebenen Softwareentwicklung sowie auf die unterschiedlichen Testarten eingegangen.

In diesem Kapitel soll nun auf verschiedene Metriken und Methoden, die die Umsetzung von Test-getriebener Softwareentwicklung unterstützen, eingegangen werden. Die folgenden Abschnitte beschreiben neben direkten Umsetzungshilfen, wie Pair-Programming und Code-Review, auch Metriken, die zur Evaluierung des geschriebenen Quellcode bzw. dessen Testfällen hilfreich sein können.

3.1 Testabdeckung (Code-Coverage)

Die Metrik der Testabdeckung beschreibt einen prozentualen oder absoluten Wert, in welchem Ausmaß der implementierte Quellcode durch Testfälle ausgeführt und daher abgedeckt wird.

Idealerweise sollte bei der Test-getriebenen Softwareentwicklung stets eine 100%ige Abdeckung erfolgen. In der Praxis ist dies jedoch aus Kosten-Nutzen-Gründen teilweise nicht möglich.

Es existiert eine Reihe von Basiskriterien, die bei einer Analyse der Abdeckung evaluiert werden:

1. Funktionsabdeckung: jede Funktion wird durch einen Test einmal ausgeführt

2. Anweisungsabdeckung: jede Anweisung wird im Programm ausgeführt

3. Verzweigungsabdeckung: jede Verzweigung wird abgedeckt (zB. IF-ELSE-Konstrukte)

4. Entscheidungsabdeckung: auch wenn diese der Verzweigungsabdeckung sehr ähnlich sind, können unterschiedliche Entscheidungen auch anders als simple Verzweigungen erfolgen (zB. unterschiedliche Programmeinstiegspunkte)

5. Bedingungsabdeckung: jede Bedingung wird als wahr und falsch getestet

6. Zustandsabdeckung: jeder Zustand wird in einem Endlichen Automat (engl. Finit-State-Machine) getestet

7. Parameterabdeckung: die Parameterwerte wurden für eine Funktion hinreichend getestet

Für die Evaluierung der Funktionalität einer Komponente ist es häufig notwendig verschiedene Testfälle für Grenzbereiche und den unterschiedlichen Abdeckungskriterien zu erstellen, sodass eine Überdeckung von Testfällen erfolgt und Funktionen teilweise mehrfach ausgeführt werden müssen. Diese Überdeckung wird durch eine weitere Kennzahl beschrieben, die sich Testintensität nennt und im folgenden Abschnitt beschrieben wird.

Je nach dem Qualitätsanspruch der Software können auch noch weitere Abdeckungskriterien überprüft werden, wie zum Beispiel Schleifenabdeckung. Diese Tests sind häufig bedeutend aufwändiger und werden daher oft nur in Bereichen mit absoluter NULL-Fehler-Toleranz durchgeführt, wie zum Beispiel im Flugwesen.

3.2 Test-Intensität

Um Funktionen und Komponenten mit allen Verzweigungen bzw. Pfaden und Ausnahmen sowie Grenzfällen zu testen, müssen häufig sogenannte Überdeckungstests

erstellt werden.
Vor allem für Grenzfälle wird häufig ein Error Guessing (also dem Erraten von möglichen Fehlern durch Tester oder Entwickler) durchgeführt. Die Methode ähnelt daher sehr dem Smoke-Test, der das Testobjekt schnell auf unterschiedliche Funktionen testet ohne 'abzurauchen'.

Je nach Parametern einer Funktion kann aus Zeit-, sowie Kosten-Nutzen-Gründen häufig eine Funktion nicht vollständig getestet werden, sodass die Testfälle hier auf bedachte Problemfälle reduziert werden und bei Bedarf (z.B. dem Entdecken eines Fehlers) erweitert werden.

3.3 Code-Duplizierung

Unter Code-Duplizierung (auch Code-Wiederholung genannt) wird die wörtliche oder semantische Wiederholung von Codepassagen verstanden, die häufig aus Bequemlichkeit des Entwicklers entsteht, für gleiche semantische Codepassagen nicht eine abstrahierte Funktionalität zu schaffen, sondern diese einfach zu kopieren.

Die Code-Duplizierung ist daher natürlich nicht als Technik oder Methode, sondern als Metrik zu verstehen. Sie widerspricht dem Prinzip des 'Clean Code' bzw. ist auch als Anti-Pattern 'DRY (don't repeat yourself)' bekannt.

Softwaretools zur statischen Analyse von Quellcode sind teilweise in der Lage sowohl wörtliche als auch semantische Wiederholungen zu erkennen, und geben hierfür Hilfestellungen zur Auflösung.

Code-Duplizierung zieht eine Reihe möglicher Problemfälle nach sich:

- ein entdeckter Fehler im Quellcode muss an mehreren Stellen behoben werden
- das gleiche gilt auch für die Wartung und Optimierung von Quellcode

- der gesamte Quellcode wird durch das Kopieren von Codepassagen unübersichtlicher und länger
- das Übersetzen von Quellcode in Programmcode wird langsamer
- gleiche Aufgaben werden durch unterschiedliche Codepassagen durchgeführt und verschleiern die Verantwortlichkeit bzw. die Zuständigkeit

In der Test-getriebenen Softwareentwicklung wird die Phase des Refactorings (beschrieben im folgenden Abschnitt) genau dazu benutzt um doppelte Codepassagen zu erkennen und aufzulösen.

3.4 Extreme Programming

Extreme Programming (XP) ist eine Reihe von Methoden und Praktiken, die sehr ausführlich im Buch 'Extreme Programming Explained' von Kent Beck beschrieben werden.[1]

Kent Beck beschreibt 12 Praktiken, deren gemeinsame Umsetzung sowohl die Produktivität als auch die Qualität eines Softwareprojekts steigern sollen. Das Zusammenspiel dieser Praktiken veranschaulicht er in einer Zeichnung, die in Abbildung 3.1 dargestellt ist.

3.4.1 The Planning Game

Ziel des Planungsspiels ist es, mit möglichst wenig Aufwand einen großen Nutzen aus der programmierten Software für den Kunden zu ziehen.

Um dies zu Erreichen wird eine Planungsphase mit allen beteiligten Personen (natürlich sofern zumutbar) eingelegt, die mit Hilfe von Story Cards Ideen und

[1] Vgl. Beck (2004)

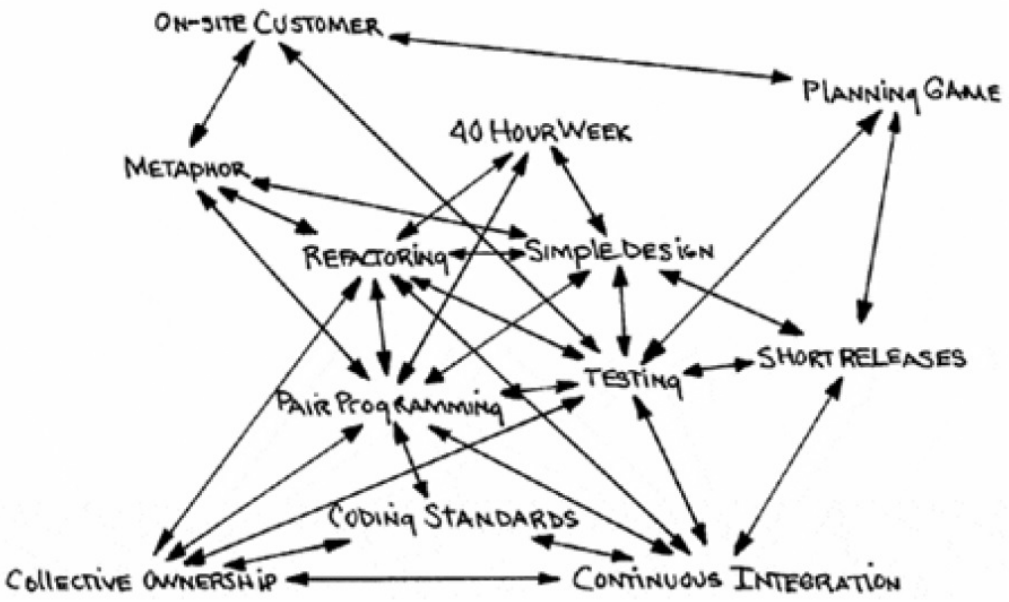

Abbildung 3.1: Zusammenspiel der XP Praktiken

Aufgaben der Software niederschreiben.

Als Resultat des Planungsspiels sollte eine Reihe von User-Stories vorliegen. Diese beschreiben im Großen und Ganzen die Version der Software, die im nächsten Iterationszyklus entwickelt werden soll.

3.4.2 Short Releases

Nachdem sich aus dem Planungsspiel die wertvollsten User-Stories ergeben haben, kann konsequenterweise davon ausgegangen werden, dass jede dieser Produktfunktionen einen wichtigen Wert für den Kunden haben. Dem Kunden sollte also diese Funktionalität so schnell wie möglich zugänglich gemacht werden - und genau das passiert mit Releases in kurzen Abständen.

Um dies erreichen zu können, muss bereits von Anfang an ein Workflow erarbeitet werden, der den Aufwand für das Builden, Packagen und Releasen der Software so klein hält, dass dies in besonders kurzen Abständen erfolgen kann.

3.4.3 Metaphore

Die Metapher ist ein Begriff für ein Objekt, eine Funktionalität oder eine Eigenschaft, für die es bisher kein eigenes Wort gibt und stellt die Versinnbildlichung einer abstrakten Vorstellung dar, die durch das bekannte Wort leichter verständlich werden soll.
In der Softwareentwicklung soll die Metapher das Ziel bzw. den Zweck eines Produkts oder einer Funktionalität zum Ausdruck bringen, ohne genau auf die einzelnen Funktionalitäten oder Eigenschaften einzugehen. Hierfür werden häufig Analogien aus der realen Welt herangezogen, welche der gewünschten Software-Funktionalität ähnlich sind. Als Beispiel sei hier der Papierkorb eines Betriebssystems erwähnt: Die Eigenschaften und Funktionen eines Papierkorbs aus der realen Welt lassen sich als Metapher leicht auf eine Software-Funktionalität übertragen.

3.4.4 Simple Design

Einfaches Design entsteht unter anderem durch die Einhaltung der zwei Prinzipien YAGNI (you ain't gonna need it) und KISS (keep it small and simple). Ein Ziel des einfachen Designs sollte es sein 'Clean Code' (siehe Kapitel 2.2) zu erzeugen.

Kent Beck hält fest, wie wichtig es ist ein einfaches Design beizubehalten und beschreibt es durch eine Reihe von Kriterien:

- es können alle Tests ausgeführt werden
- beschreibt die Logik des Programms eindeutig (weist also keine Doppelgleisigkeiten auf)
- es beschreibt jede zur Entwicklung notwendige Intention
- ist so klein wie möglich und so groß wie nötig

Problematisch dabei ist die Umsetzung eines solchen Designs. Es kann häufig mit größeren und teilweise zeitaufwändigen Refactorings verbunden sein. Gerade un-

ter Zeitdruck kann dem Entwickler diese Arbeit im Vergleich zu einem schnell funktionierenden Workaround als unproduktiv oder gar unsinnig erscheinen. Weiters muss sich das einfache Design auch gegen die Argumentation der Effizienz komplexer Algorithmen behaupten, deren Verständnis selbst nicht einfach ist.

Diesen Problemen begegnet das einfache Design mit der erhöhten Übersichtlichkeit in der Logik des Programmflusses, deren Mangel häufig Ursache von Fehlern und Missverständnissen ist. Die Vorteile des einfachen Designs werden vor allem bei steigender Komplexität der Software oder bei Projekten mit großen Teams offensichtlich. Es erlaubt anderen Entwicklern, aber auch dem Entwickler der Implementation selbst, sich schneller im Quellcode zurecht zu finden, was dahingehend auch die Produktivität der Entwickler steigert.

Ein einfaches Design ist daher auf lange Sicht immer zu bevorzugen um die Wartbarkeit und Qualität der Software zu erhöhen.

3.4.5 Testing

Testen ist eine grundlegende Praktik, die es ermöglicht die Funktionalität eines Moduls oder ganzen Programms zu verifizieren. Ohne einen Test könnte nie jemand mit Sicherheit sagen, ob die beschriebene Funktionalität wirklich existiert oder nicht. In XP existiert eine Funktionalität erst, wenn es einen bestehenden Test dafür gibt, der erfolgreich absolviert wird. Im Idealfall kann dieser Test automatisiert durchgeführt werden - in jedem Fall, jedoch, muss bereits vor der Implementation klar sein, was das Ergebnis des geschriebenen Quellcodes bewirken soll und dieser Test kann und soll festgehalten werden.

3.4.6 Refactoring

Refactoring bedeutet bestehende Funktionalität einfacher zu gestalten. Sie sollte immer dann angewendet werden, wenn der Code dazu auffordert.

Das klassische Beispiel dafür ist das Vermeiden von Duplikationen. Gemäß der Praktik des einfachen Designs sollte der Quellcode immer so strukturiert werden, dass er immer klar verständlich bleibt. Hierfür ist die vorhergehende Praktik ideal, denn die bestehenden Testfälle garantieren dem Entwickler, dass durch die Vereinfachung des Designs die Funktionalität nicht in Mitleidenschaft gezogen wird.

3.4.7 Pair-Programming

Diese Praktik basiert auf dem Prinzip: 4 Augen sehen besser als zwei. Das geht natürlich nur, wenn alle 4 Augen auf die gleiche Sache gerichtet sind. Für die Programmierer bedeutet dies: zwei Personen, ein Bildschirm, eine Tastatur, eine Maus.

Das angestrebte Ziel dieser Praktik besteht darin, Fehler so früh wie möglich zu erkennen (also den Reviewing Prozess auf den frühesten Zeitpunkt zu legen), aber auch strategische Design-Implementationen zu geteilter Hand durchzuführen. Der resultierende Code wird dadurch einfacher und verständlicher, da er bereits von Anfang an nicht nur für einen Entwickler geschrieben wird und das Design durchdachter ist. Diese Praktik trägt daher maßgeblich dazu bei 'Clean Code' zu erzeugen (siehe Kapitel 2.2).

Kent Beck bezeichnet diese Praktik als besonders dynamisch und merkt an, dass auch der Wechsel von Pair-Programming Partnern in regelmäßigen Abständen sinnvoll ist.

3.4.8 Collective Ownership

XP verlangt von jedem Entwickler für das gesamte System verantwortlich zu sein. Natürlich kennt sich nicht jeder in jedem Quellcodeabschnitt gleich gut aus, was aber nicht zwangsläufig heisst, dass dieser nicht verändert werden darf oder gar sollte.

Wenn sich eine Situation ergibt, in der ein Codeabschnitt entdeckt wird, der verbessert werden könnte, so ist das im Zuge der Praktik eines einfachen Designs zu erledigen - von jedem, egal an welcher Stelle.

3.4.9 Continuous Integration

Diese Praktik beschreibt die Vorgehensweise, dass jede Funktionalität, die dem gesamten System hinzugefügt wird, dieses auch verbessert. Die Testfälle des bestehenden Systems laufen immer zu 100%.

Somit gibt es keinen erkennbaren Grund, weshalb das System durch die neue Funktionalität zerstört würde - also etwas hinzugefügt wird, ohne dass die bestehenden und neuen Tests wieder zu 100% erfolgreich durchgeführt werden.

3.4.10 40-Hour Week

Bei dieser Praktik geht es im Speziellen nicht darum eine genaue 40-Stunden Woche einzuhalten. Jeder Entwickler arbeitet mit begrenzten körperlichen und geistigen Ressourcen. Wird über diese Ressourcen hinaus gearbeitet, leidet darunter die spätere Arbeit.

Es sollte daher darauf geachtet werden, dass jeder Entwickler innerhalb dessen arbeitet, was er dauerhaft leisten kann um kreativ und produktiv zu bleiben.

Ausnahmen sind natürlich möglich, manchmal sogar notwendig. Lässt sich jedoch ein Problem durch eine Woche Überstunden nicht lösen, ist eine weitere Woche Überstunden wahrscheinlich nicht die richtige Herangehensweise.

3.4.11 On-Site Customer

Für jedes Produkt gibt es einen Kunden, der das Produkt schließlich benutzen soll. Schließlich weiss er am besten, was das Produkt können soll. Je länger der Kommunikationsweg mit dem Kunden dauert, desto später werden Rückfragen der Entwickler beantwortet und das Projekt zieht sich in die Länge.

Natürlich ist es häufig nicht möglich den Kunden direkt am Entwicklungsort zu haben. Dennoch beschreibt diese Praktik die Notwendigkeit, die Iterationen für Rückfragen mit dem Kunden so kurz als möglich zu halten um Fehlentwicklungen zu vermeiden.

3.4.12 Coding Standards

Die Einhaltung von Coding Standards ergibt sich konsequenterweise aus den zwei Praktiken 'einfaches Design' und 'Collective Ownership'.

Angesichts der Tatsache, dass jeder Entwickler in der Lage sein sollte den bestehenden Quellcode zu ändern und zu verbessern, wird vorausgesetzt, dass jeder Entwickler ebenfalls in der Lage ist, den Code so schnell wie möglich zu verstehen.

Auf diese Wiese kommunizieren die Entwickler in der gleichen Sprache, was dem schnellen Verständnis förderlich ist.

3.5 Build-Automatisierung

Die Build-Automatisierung ist ein wesentlicher Bestandteil bei der Test-getriebenen Softwareentwicklung. Sie ist dazu gedacht, die täglichen und automatisierbaren Aufgaben des Softwareentwicklers zu erleichtern und zu beschleunigen. Zu diesen Aufgaben gehören beispielsweise:

- das Übersetzen von Quellcode in Programmcode
- das Linken von Objektcode zum vollständigen Programm
- das Durchführen von Tests
- das Zusammenpacken des Programms und notwendiger Daten
- das Verteilen des Programms auf Produktionssysteme
- das generische Erstellen von Dokumentation und Release Notes

Die Test-getriebene Softwareentwicklung gibt diesem System noch zusätzlich den Rahmen, auf ihrem Gebiet performant zu laufen um die Iterationszeit bei der Implementation zu beschleunigen.

3.6 Framework for Integrated Tests

Das Framework für Integrated Tests (abgekürzt FIT) ist sowohl eine Methode als auch ein Softwareprodukt selbst. Es steht für die Automatisierung von Akzeptanztests und muss im Gegensatz zu Modultests nicht direkt als Quellcode formuliert werden, sondern wird in externen Dokumenten beschrieben. Das Open Source basierte Framework wurde von Ward Cunningham entwickelt und erlaubt die Formulierung der Anforderungen in externen Dokumenten, wie beispielsweise in Microsoft Office Datenformaten oder auch in einem Wiki.

Das FIT wurde speziell für die Test-getriebene Softwareentwicklung programmiert und dient vor allem dazu, Testern ohne Kenntnissen von Programmiersprachen die automatisierte Erstellung von Tests zu ermöglichen.

Hierfür werden tabellarisch unterschiedliche Tests mit IST-Eingabewerten und SOLL-Ausgabewerten geschrieben, welche von der FIT-Software eingelesen und durchgeführt werden. Anschließend wird ein Dokument als Resultat erstellt, in dem die Testergebnisse eingetragen werden.

3.7 Code Review

Unter Code Review wird die systematische Evaluierung von Quellcode durch eine weitere Person (meist selbst auch Entwickler) durchgeführt. Sie ist dazu gedacht die Modifikationen am Quellcode nach dem 4-Augen-Prinzip zu besprechen und dabei Fehler und Problemfelder zu entdecken - gegebenenfalls auch gleich zu beheben.

Das Review wird ebenfalls durch die IEEE-Norm 729 beschrieben und bezeichnet dort einen Prozess, in dem Projektergebnisse einem oder mehreren Gutachtern präsentiert, und von diesem/n genehmigt werden.

Das Code Review zählt daher zu den Qualitätssicherungsmaßnahmen und spielt im Zuge der Test-getriebenen Softwareentwicklung selbst eine eher untergeordnete Rolle.

Dennoch kann auch dort dieser Prozess als hilfreich angesehen werden, da seine Aufgabe darin besteht einen Fehler zu erkennen, bevor dieser im System integriert wird und dort Schäden verursacht.

Der Prozess selbst ist weniger formal definiert und erlaubt unterschiedliche Anwendungsmöglichkeiten:

- Over-the-shoulder: der Gutachter schaut dem Entwickler über die Schulter während dieser dem Gutachter den Quellcode erklärt.

- Email-pass-around: vor allem in Entwicklungsumgebungen, in denen das System dies automatisiert unterstützt, erhält der Gutachter ein EMail mit dem veränderten Quellcode, den er selbst begutachtet und nach positiver Beurteilung auch frei gibt.

- Pair-Programming: Der Gutachter ist in diesem Fall ständig an der Entwicklung beteiligt und macht während der Eingabe auf Fehler und Verbesserungen im Design aufmerksam. Diese Methode ist ausführlicher in Kapitel 3.4.7

beschrieben.

- Tool-assisted code review: Es wird versucht nach formalen Kriterien so weit als möglich ein automatisiertes Review vorzunehmen.

In jedem Fall soll dadurch die Qualität der resultierenden Software erhöht werden.

Daher laufen beim Code Review folgende Qualitätsprozesse ab:

- Dem Entwickler fallen selbst Verbesserungsmöglichkeiten bei der Vorstellung seiner Modifikationen auf.
- Der Gutachter stellt im Zuge seines Reviews Verständnisfragen, die daraufhin als Kommentare im Quellcode oder in eine Dokumentation einfließen können.
- Der Gutachter entdeckt selbst Probleme oder Verbesserungsmöglichkeiten und empfiehlt diese dem Entwickler.

Die Stärken eines Review werden vor allem bei folgenden Problemfeldern ausgespielt:

- dem Nicht-Einhalten von Standards
- Fehler im Verständnis der Anforderungen
- Fehler in den Anforderungen selbst
- Strukturelle Probleme
- Hinweise auf Probleme in der Wartbarkeit von Quellcode
- Fehler in Spezifikationen von Schnittstellen

Das Resultat eines Code Reviews sollte in jedem Fall die Qualitätsverbesserung von Quellcode sein und diesen daher fehlerärmer, effizienter und sauberer machen.

4 Evaluierung und Analyse von Test-getriebener Softwareentwicklung

Im kommerziellen Umfeld kommen schnell einige Fragen zu Tage, die sich kritisch mit der Durchführung von Test-getriebener Softwareentwicklung beschäftigen.

1. Wie schnell ist die Software fertiggestellt?
2. Wie gut passt die Software auf die Anforderungen?
3. Wie schnell können Fehler behoben werden?
4. Wie viele Fehler entstehen während der Entwicklung bzw. sind gleichzeitig vorhanden?
5. Wie sehr sträuben sich Entwickler, diese Methoden der Test-getriebenen Softwareentwicklung anzunehmen?

Für jede dieser Fragen gibt es Metriken, die den Unternehmen Gewissheit darüber verschaffen können. Natürlich setzt jede Metrik auch einen Vergleich voraus, d.h., die endgültige Sicherheit über die Effizienz von Test-getriebener Softwareentwicklung im Vergleich zur 'normalen Softwareentwicklung' könnte nur bei paralleler Umsetzung des gleichen Produkts und im Nachhinein entstehen.

Des Weiteren würde ebenfalls nicht zwangsläufig eine allgemeine Aussage über

die Wirtschaftlichkeit erfolgen, sondern nur für diesen speziellen Anwendungsfall.

Trotzdem lässt sich bei direktem Vergleich auch eine Tendenz über die mögliche Wirtschaftlichkeit dieses Prozesses sagen, so dass die Metriken nun (korrespondierend zu den oben genannten Fragestellungen) wie folgt aufgelistet werden:

1. Time to Release - Entwicklungsdauer in Personen-Stunden, bzw. absolut in Produktionsmonaten.

2. Externe Akzeptanztests - mit dem Ergebnis, wie viele dieser Tests erfolgreich absolviert wurden.

3. Burn-Down Chart - Anzahl der zu einem Zeitpunkt (typischerweise täglich) bekannten und nicht behobenen Issues.

4. Ebenfalls über Burn-Down Chart kann die Summe aller aufgetretenen Issues gemessen werden.

5. Die soziale Akzeptanz kann durch psychologische und soziale Studien, sowie im persönlichen Gespräch mit den umsetzenden Entwicklern in Erfahrung gebracht werden. Weiters kann die soziale Akzeptanz durch anonyme Fragenkataloge analysiert werden.

Diese Metriken bergen jedoch ein großes Manko, welches es Unternehmen häufig nicht ermöglicht eine Evaluierung selbst vorzunehmen: Zumindest für eine erste Evaluierung müsste eine Entwicklung zweigleisig produziert werden, was zu vermehrten Kosten für diese Entwicklung führen kann.

Aus wirtschaftlichen Gründen sind deswegen Unternehmen häufig weder gewillt noch in der Lage diese Form der Evaluierung durchzuführen, sodass hier auf wissenschaftliche Studien zurückgegriffen werden muss.

4.1 Wissenschaftliche Studien und Artikel über Test-getriebene Softwareentwicklung

Aufgrund der im vorhergehenden Kapitel erwähnten wirtschaftlichen Mankos, wurden viele Studien in Zusammenarbeit mit Universitäten und anderen wissenschaftlichen Einrichtungen durchgeführt - direkte Umsetzungen anhand kommerzieller Projekte sind in der Literatur dünn gesät.

4.1.1 Sun Microsystems und Bethel College

Die erste erwähnte Studie wurde von Reid Kaufmann und David Janzen im Auftrag einer Zusammenarbeit von Sun Microsystems und dem Bethel College durchgeführt.
Es wurden zwei Gruppen von jeweils 4 Studenten als Testobjekte herangezogen und jeweils dieselbe Aufgabenstellung bei unterschiedlichen Herangehensweisen evaluiert.

Als Ergebnis dieser Studie wurden beobachtet, dass die Test-first Gruppe zumindest eine größere Sicherheit über die Richtigkeit ihres Codes verspürte, was auch darin resultierte, abgesehen von den Testfällen, ca. 50% mehr Code zu erzeugen (was auch darauf schließen lässt, dass Refactoring nur zu einem geringeren Teil durchgeführt wurde). [1]

4.1.2 IBM Corporation und North Carolina State University

Die nächste Studie wurde von Laurie Williams, E. Michael Maximillen und Mladen Vouk durchgeführt mit dem spannenden Ergebnis, dass der Test-getriebene Ansatz selbst bei einem Team mit weniger Erfahrung eine deutliche Senkung der Programm-Defekte bewirkte. Obwohl diesem Effekt zwar leicht gehobene Entwicklungskosten gegenüber stehen, scheint IBM inzwischen grundsätzlich den Ansatz

[1] Vgl. (Kaufmann & Janzen, 2003)

der Test-getriebenen Entwicklung zu bevorzugen. Wichtig an dieser Studie ist vor allem zu bemerken, dass sie an einem laufenden kommerziellen Projekt von IBM durchgeführt wurde und daher starke Aussagekraft für das Unternehmen hatte.

Auch in dieser Studie zeigten die Probanden, dass vor allem bei der Implementierung von Code zum Ende des Projekts (quasi in letzter Sekunde) vom Entwicklungsteam mit dem Test-getrieben Ansatz sowohl mit mehr Selbstsicherheit der Entwickler, als auch mit bedeutend weniger Fehlern durchgeführt wurden.[2]

4.1.3 Virginia Polytechnic Institute und North Carolina State University

Als erste Studie mit 24 professionellen Programmierern wurde 2003 von George und Williams eine Studie zur Produktivitäts- und Qualitätsanalyse im Zuge von Test-getriebener Softwareentwicklung veröffentlicht.

Dabei stellten die beiden Wissenschaftler fest, dass bedeutend mehr Tests bei dem Test-first Ansatz geschrieben wurde, als bei einem Test-last Ansatz. Dies resultierte in zwei Konsequenzen: Es zeigte sich, bei einem anschließenden Black-Box-Test als Akzeptanztest und zur Überprüfung der resultierenden Qualität, dass die Test-getriebene Gruppe um 18% mehr der durchgeführten Analyse-Tests im Vergleich zur Kontrollgruppe bestand. Gleichermaßen war jedoch auch die Entwicklungszeit der Kontrollgruppe um 16% kürzer als die der Test-getriebenen Gruppe.[3]

4.1.4 IBM Corporation und North Carolina State University (2)

Ebenfalls 2003 von Laurie Williams und E. Michael Maximillen wurde eine weitere Fallstudie durchgeführt. Das Paper zeigt, dass Test-getriebene Softwareentwick-

[2] Vgl. (Williams et al., 2003)
[3] Vgl. (George & Williams, 2003)

lung eine um 50% reduzierte Fehlerrate produzierte, bei nur minimalen Produktivitätseinbußen. Das Ergebnis entstand durch den Vergleich mit einem ähnlichen Projekt, welches ohne Test-getriebene Entwicklung implementiert wurde. Dieser Vergleich erscheint sehr vage, weshalb das Ergebnis dieser Studie dahingehend in Frage gestellt werden muss.

Interessanter ist die Schlussfolgerung der beiden Autoren, dass der resultierende Code auch ein 'durchdachteres' Design aufwies, was natürlich nicht durch Zahlen belegbar war sondern nur die Meinung der Autoren widerspiegelt.[4]

4.1.5 Simex LLC und University of Kansas

2005 veröffentlichte die IEEE Computer Society einen Artikel von David Janzen und Hossein Saiedian. Sie beschäftigen sich umfassend mit Test-getriebener Softwareentwicklung in der Industrie und beziehen sich auf unterschiedliche Studien, die alle den Qualitäts-Effekt dieser Methode hervorheben (jedoch auch den Einfluss auf die Produktivität im Sinne der Entwicklungsdauer aufzeigen). Diese Studien zeigen eine Fehlerverbesserung zwischen 18% und 50% bei zeitlichen Produktivitätseinbußen bis zu 16%.

Der Artikel zeigt ebenfalls Studien aus dem akademischen Umfeld auf, die sich etwas ernüchternder darstellen - nämlich bei Qualitätsverbesserungen bis zu 54% bei Produktivitätseinbußen bis zu 50%.

Auch wenn der Artikel der beiden Autoren auf den möglichen starken positiven Einfluss von Test-getriebener Softwareentwicklung hinweisen, machen sie dennoch darauf aufmerksam, dass es deutlich zu wenig Fallstudien zu diesem Thema gibt um eindeutige Aussagen treffen zu können.[5]

[4] Vgl. (Maximilien & Williams, 2003)
[5] Vgl. (Janzen & Saiedian, 2005)

4.1.6 Research Center on Software und Alarcos Research Group

Im Jahr 2006 wurde ein Experiment durchgeführt, das einerseits große Hoffnung über die Aussagekraft versprach - schließlich jedoch ein recht ernüchterndes Ergebnis lieferte: 28 erfahrene Programmierer wurden in 4 gleich große Gruppen aufgeteilt, in denen jeweils zwei Gruppen einen Test-getriebenen Softwareentwicklungsansatz und die zwei anderen Gruppen das Testen nach der Implementation (TAC - Test after Coding) durchführten.

Es konnte statistisch festgestellt werden, dass Test-getriebene Softwareentwicklung länger gebraucht hat als der TAC Ansatz. Gleichzeitig konnte nicht eindeutig eine erhöhte Qualität der resultierten Software erkannt werden - obwohl die Autoren dennoch die erhöhte Qualität unterstellten. [6]

4.1.7 Microsoft Corporation

Ebenfalls im Jahr 2006 wurde die Effizienz von Test-getriebener Softwareentwicklung von Microsoft durchgeführt. Die Studie ist von besonderem Interesse, da ein Software-Großkonzern offensichtlich eine gestärktes Eigeninteresse an der Wirtschaftlichkeit dieser Entwicklungsmethode hegt, als ein universitäres Fallstudienprojekt.

Die Autoren Thirumalesh Bhat und Nachiappan Nagappan schlussfolgerten aus ihren 2 Fallstudien (mit einer Dauer von 6 und 12 Monaten) eine Qualitätsverbesserung durch Test-getriebene Softwareentwicklung um den Faktor 2.6 bis 4.2 - also um mehr als die Hälfte verminderte Fehlerrate. Gleichsam stellten sie eine Erhöhung der Entwicklungszeit zwischen 15% und 35% fest. [7]

[6] Vgl. (Canfora et al., 2006)
[7] Vgl. (Bhat & Nagappan, 2006)

4.1.8 IBM Corporation

2007 wurde erneut eine Studie von IBM zum Thema der Softwarequalität und Produktivität veröffentlicht. Erneut zeigte sich in dieser Studie, dass eine deutliche Reduktion von Defekten im Vergleich zu einer geringer Erhöhung der Entwicklungszeit durch Test-getriebene Softwareentwicklung entstand.

Interessante weitere Informationen konnte die Studie über die Codequalität geben: hier zeigte sich, dass trotz wachsender Quellcode-Basis (Lines of Code, kurz LOC) die zyklische Komplexität unterproportional anstieg. Diese Metrik weist darauf hin, dass Test-getriebene Softwareentwicklung vor allem in größeren Projekten für die Entwicklung eines strukturierten und weniger komplexen Designs führen kann.

Ebenfalls wurde in dieser Studie hervorgehoben, dass der Aufwand Tests zu erstellen über die Projektdauer hinweg immer geringer wurde - was darauf schließen lässt, dass für längere Projekte der Produktivitätsverlust durch Test-getriebene Softwareentwicklung immer weniger ausschlaggebend wird.[8]

4.1.9 Microsoft, Research Center San Jose und Carolina State University

Im Jahr 2008 wurde eine qualitative Langzeit-Studie herausgegeben, die auf Basis großer Projekte von Microsoft und IBM die Effizienz von Test-getriebener Softwareentwicklung evaluieren konnte.

Es wurden die Teams von wichtigen Projekten (wie z.B. Microsoft Windows oder Visual Studio) auf eine Test-getriebene Entwicklung umgestellt und die Produktivität und Qualität beobachtet.

Das interessante Ergebnis zeigte eine reduzierte Fehlerrate von 9% - 61% im

[8] Vgl. (Sanchez et al., 2007)

Vergleich zur nicht Test-getriebenen Softwareentwicklung. Andererseits erhöhte sich die Entwicklungszeit um 15% bis 35%, wobei die Projektleiter angaben, eine deutlich verringerte Zeit in die Wartung der Software investieren zu müssen - was den offensichtlichen Produktivitätsverlust relativiert.[9]

4.2 Fazit

Die vorgelegten Studien können mangels gut geeigneter Metriken und Studienbedingungen nur bedingt eindeutige Vor- und Nachteile der Test-getriebenen Softwareentwicklung aufzeigen. Dennoch wird versucht in den folgenden Abschnitten zumindest eine tendenzielle Bewertung durchzuführen.

4.2.1 Vorteile

- Bei praktisch allen Studien zeigt sich, dass die Test-getriebene Softwareentwicklung für mehr geschriebene Testfälle sorgt. Dahingehend ist diese Methode zumindest für die Produktivität in Bezug auf die Testfall-Erstellung günstig.

- Weiters beschreiben die Studien einen verringerten Bedarf einen Debugger zur Behebung von Problemen zu benutzen - stattdessen greifen vor allem Nicht-Experten hier eher auf die Möglichkeit zurück, im Versionierungssystem das aktuelle Problem mit dem erfolgreichen Absolvieren von Tests einer früheren Version zu vergleichen.

- Zumindest subjektiv geben Autoren der Studien an, dass das strukturelle Design des entstandenen Quellcodes 'durchdachter' zu sein scheint und führen dies auf die Tatsache zurück, dass Designs von Programm-Strukturen und deren Schnittstellen durch die Testfälle schon vor bzw. bei der Implementation benutzt werden.

[9] Vgl. (Nachiappan Nagappan & Williams, 2008)

- Da die Test-getriebene Softwareentwicklung die Arbeit in kleineren Schritten bevorzugt, zeigten sich vor allem bei Projekten mit großen Teams nur kleine Einbußen der Produktivität.

- Die Ergebnisse zeigten in praktisch allen Studien, dass die Test-getriebene Kontrollgruppe subjektiv eine höhere Sicherheit beim Programmieren verspürte.

- Die Test-getriebene Softwareentwicklung führte in fast allen Studien zu Strukturen, die mehr Modularität aufwiesen. Dabei konnte beim Quellcode, auch wenn dies nicht Gegenstand der Studien war, eine erhöhte Wartbarkeit festgestellt werden.

- Der Vorteil, dass Systeme durch Refactorings überarbeitet werden können ohne weitere Fehler zu produzieren wurde leider ungenügend in den Studien hervorgehoben. Die Test-getriebene Softwareentwicklung versucht hier das Prinzip 'never change a running system' außer Kraft zu setzen.

4.2.2 Nachteile

- Aus der Verwendung von Test-getriebener Softwareentwicklung scheint sich, zumindest nicht zwangsläufig, eine Qualitätsverbesserung abzuleiten - dafür divergieren die Studien in ihren Ergebnissen zu sehr - es fehlen daher Studien für spezielle Anwendungsfälle von Test-getriebener Softwareentwicklung um hier eindeutige Vor- und Nachteile hervorzuheben.

- Alle Studienfälle zeigen zumindest eine marginale Verschlechterung in der Entwicklungsdauer. Jedoch wies auch keine Folge-Studie einen Vergleich mit der Wartung von Software bzw. nachträglichen Fehlerverbesserungen der Software auf. Gerade diese Metrik stellt sich bei der Evaluierung von Test-getriebener Softwareentwicklung als problematisch heraus, da das erklärte Ziel agiler Softwaremethoden die fokussierte Entwicklung in Richtung eines gewünschten, jedoch unklaren, Umfangs ist und nicht die optimierte Entwicklungszeit für die Lösung einer fest definierten Problemstellung.

- Auch wenn viele Studien die subjektive Sicherheit für den eigenen Code aufzeigten, kann dies zum Trugschluss führen, dass der erstellte Code fehlerfrei sei. Die Entwickler können somit durch die Anwendung von Test-getriebener Softwareentwicklung in falsche Sicherheit gewiegt werden.

- Die größere Anzahl an Tests, die im Zuge der Test-getriebenen Softwareentwicklung erstellt werden spiegelt nicht zwangsläufig deren Qualität wider.

- Die erhöhte Anzahl an Tests kann zu Duplizierungen in den Testfällen führen, was vor allem bei Refactoring-Arbeiten des Quellcodes auch in ein Refactoring der Tests und damit in einen noch größeren Zeitverlust resultiert.

- Vor allem für die Multi-Thread Programmierung bzw. für nicht deterministische Systeme bietet Test-getriebene Softwareentwicklung keine einfache Methode zum Testen an und kann in einem verwirrenden Festhalten am Schema resultieren.

4.3 Kritik der Studien

Die wissenschaftlichen Studien konzentrierten sich sehr stark auf die zwei Metriken Produktivität (im Sinne der Entwicklungsdauer) und Vergleiche von Fehlerraten durch Akzeptanztests. Diese Metriken sind jedoch für eine wissenschaftliche Evaluierung von Test-getriebener Softwareentwicklung nicht ausreichend, da es nicht das erklärte Ziel dieser Softwaremethode darstellt diese im Vergleich zu anderen Methoden zu verbessern.

Weiters sind die vorgelegten Studien aufgrund ihrer Durchführung häufig nicht repräsentativ. Die Anzahl der analysierten Studienobjekte ist meistens zu gering und Fehler wurden nicht ihrer Wertigkeit nach klassifiziert.

Es wäre wünschenswert ein Design für eine Studie zu erheben, die in der Lage ist Test-getriebene Softwareentwicklung in Bezug auf die gewünschten Ziele zu analysieren und evaluieren.

5 Software im Entwicklungsprozess

Dieses Kapitel beschäftigt sich mit der praktischen Umsetzung von Test-getriebener Softwareentwicklung. Hierzu gibt es eine Reihe an Werkzeugen und integrierten Tools, die den Entwicklungsprozess vereinfachen bzw. erst praktikabel machen. Dies bezieht sich nicht nur auf die Test-getriebene Softwareentwicklung selbst sondern auch andere Software, die im Zuge eines erfolgreichen Softwareprojekts eingesetzt werden kann.

Es wurde hierfür ein Punktesystem (mit 5 möglichen Punkten) entworfen, das den Einsatz der unterschiedlichen Software in kommerziellen Entwicklungsumgebungen bewerten soll.

5.1 Software zur Unterstützung Test-getriebener Softwareentwicklung

Die folgenden Werkzeuge sind auf die Test-getriebene Softwareentwicklung spezialisiert. In den meisten Fällen beziehen sie sich auf Modul- und Regression-Testing, sowie auf die Build-Automatisierung.

5.1.1 Unit-Testing

5.1.1.1 xUnit

Unter xUnit werden verschiedene Frameworks verstanden, die für automatisierte Modultests entworfen wurden. Dabei werden Funktionalitäten bereitgestellt um Funktionen und Klassen durch ASSERT-Methoden auf korrektes Verhalten zu prüfen.

Das erste xUnit-Framework wurde von Kent Beck für die Programmiersprache Smalltalk entwickelt. Inzwischen gibt es für praktisch alle Programmiersprachen entsprechende Frameworks - die meisten davon sind unter die Open Source Lizenz GNU gestellt und frei verfügbar.

Usability: 4/5
Performance: 5/5
Einarbeitungszeit: 3/5

5.1.1.2 MSTest Manager

Der MSTest Manager des Konzerns Microsoft ist ein Werkzeug, das sowohl manuelle als auch automatisierte Tests durchführt. Dabei hilft das Tool auch beim Management von Fehlern und Aufgaben. MSTest Manager ist an unterschiedlichen Schnittstellen sowohl mit der Projektmanagementsoftware Team Foundation Server als auch mit der Entwicklungsumgebung Visual Studio verbunden und bietet so vielfältige Funktionalitäten zur Unterstützung von Regressionstests.

Usability: 2/5
Performance: 4/5
Einarbeitungszeit: 3/5

5.1.1.3 Time Partition Testing

Mit Hilfe von Time Partition Testing (TPT) können Testfälle grafisch modelliert werden. Das Softwaretool der Firma PikeTec umfasst folgende Aufgabenbereiche:

- grafische Modellierung von Testfällen
- automatische Testdurchführung
- Ergebnisdarstellung von durchgeführten Tests
- Erstellung von Testdokumentation
- Management von Testfällen
- Test-Tracing - also der Nachverfolgbarkeit von Testfällen

Usability: 5/5
Performance: 3/5
Einarbeitungszeit: 4/5

5.1.2 Build Automatisierung

5.1.2.1 Make und Make-basierte Tools

Make bzw. Make-basierte Tools arbeiten mit einer einheitlichen Skriptsprache zur Ausführung von Kommandos in Abhängigkeit vordefinierter Bedingungen.

In der Test-getriebenen Softwareentwicklung kann damit vom Übersetzen des Quellcodes in den Programmcode bis zur Verteilung des fertigen Build-Pakets auf unterschiedlichen Arbeitsumgebungen durchgeführt werden. Auch die automatische Ausführung von Testfällen und die Generierung von Dokumentation sind beliebte Anwendungsmöglichkeiten dieser Tools. Als Teil des POSIX-Standards

ist es von Haus aus auf praktisch allen Linux- und Mac-Umgebungen zu finden, ist aber für alle bekannten Betriebssysteme verfügbar und in diverse Entwicklungsumgebungen integriert.

Usability: 2/5
Performance: 5/5
Einarbeitungszeit: 3/5

5.1.2.2 Apache ANT

Das Werkzeug ANT übernimmt weitestgehend dieselben Aufgaben wie Make. Es wurde auch genau aus diesem Grund von James Duncan Davidson im Jahr 1999 entworfen, da er dieselbe Funktionalität für Java benötigte. ANT kann jedoch durch die Laufzeitumgebung Java zusätzlich auf das bereitstehende Framework zurückgreifen und auf selbstprogrammierte Befehle zurückgreifen. Es ist somit ein wenig flexibler als Make - jedoch auch bedeutend größer im Umfang.

Usability: 3/5
Performance: 3/5
Einarbeitungszeit: 4/5

5.1.2.3 MSBuild

MSBuild von Microsoft ist das direkt in Visual Studio integrierte Kommandozeilen Werkzeug zum Kompilieren, Packagen, Testen und Verteilen von Software.

Das Werkzeug verwendet ähnlich zu ANT die Beschreibungssprache XML zur Definition von Build Prozessen und versteht sich aus verständlichen Gründen sehr gut mit Visual Studio Projektdefinitionen.

Tiefergreifendes Verständnis für dieses Tool ist vor allem dann von Interesse, wenn es um komplexere Aufgaben geht, als dies durch die vorgefertigte Formulare und Masken innerhalb von Visual Studio selbst möglich ist.

Usability: 3/5
Performance: 4/5
Einarbeitungszeit: 3/5

5.2 Weitere Software während der Entwicklung

Die nachstehenden Werkzeuge beziehen sich allgemein auf Software-Entwicklung. Im folgenden werden diese (für den Leser erwartungsgemäß bekannten Tools) in Hinblick auf Test-getriebene Softwareentwicklung evaluiert. Weiters wurde die Recherche auf die Entwicklung mit der Programmiersprache C# beschränkt, da die korrespondierende praktische Studie zu dieser vorliegenden Arbeit in dieser Programmiersprache durchgeführt wurde.

5.2.1 Entwicklungsumgebungen (IDEs)

Die nun folgenden Entwicklungsumgebungen wurden in Hinblick auf Test-getriebene Softwareentwicklung evaluiert. Nachdem die Benutzbarkeit der Entwicklungsumgebung auch eine Frage der Usability ist, wurden für ein besseres Verständnis auch Screenshots der entsprechenden IDEs eingefügt.

5.2.1.1 Microsoft Visual Studio 2012

Visual Studio zählt seit jeher zu den beliebtesten Umgebungen zur Programmierung von C# und bietet in der Ultimate Version eine Vielzahl an Werkzeugen für automatische Builds, Unit-Testing, UI-Testing, Code-Analysis und Code-Coverage.

Vor allem in Verbindung mit Microsoft Team Foundation Server sind auch Projektmanagement und Team-Workflows integriert. Gleichzeitig erlaubt eine API die Integration von Tools durch Drittanbieter, wie unter anderem auch JetBrains dotCover.

Abbildung 5.1 zeigt den Aufbau der Entwicklungsumgebung.

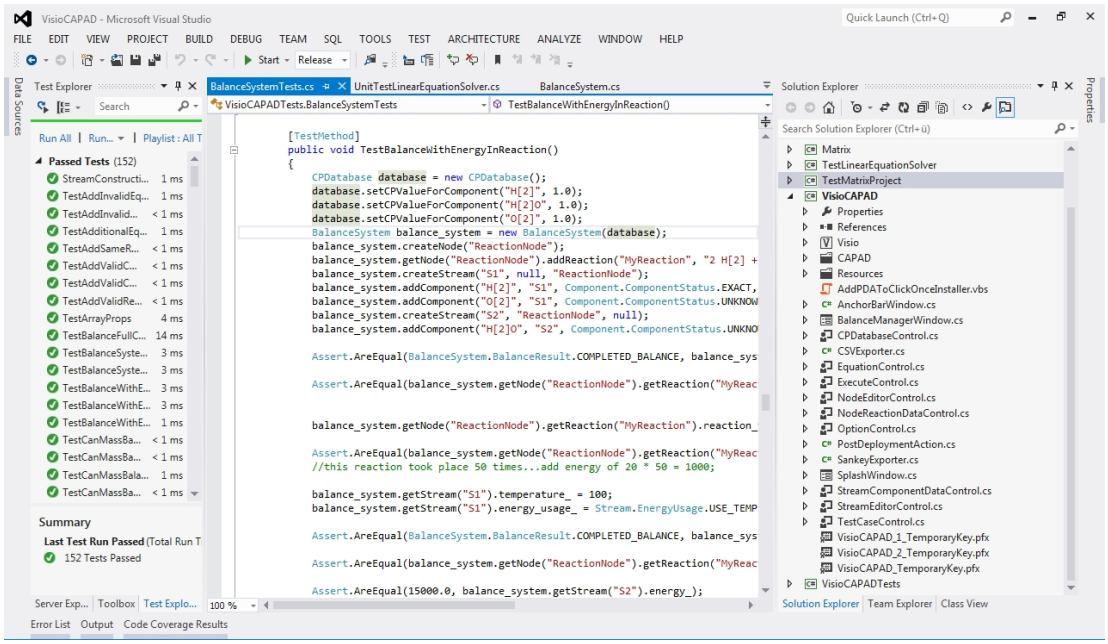

Abbildung 5.1: Visual Studio 2013 (Screenshot)

Usability: 5/5
Performance: 4/5
Einarbeitungszeit: 4/5

5.2.1.2 Eclipse

Eclipse ist einer der größten Open Source Vertreter der Entwicklungsumgebungen. Ursprünglich für die Entwicklung von Java-Programmen gedacht, gibt es nun Erweiterungen für C# und viele andere Programmiersprachen sowie dazugehörige Plugins.
Eclipse versteht sich gut auf Unit-Tests, automatisierte Builds (make oder ANT), UML-Darstellungen zur Code-Analyse und mit Plugins auch mit Code-Coverage oder Modified Conditions, z.B. durch CTC++ von VerifySoft Technology.
Abbildung 5.2[1] zeigt den Aufbau der Entwicklungsumgebung.

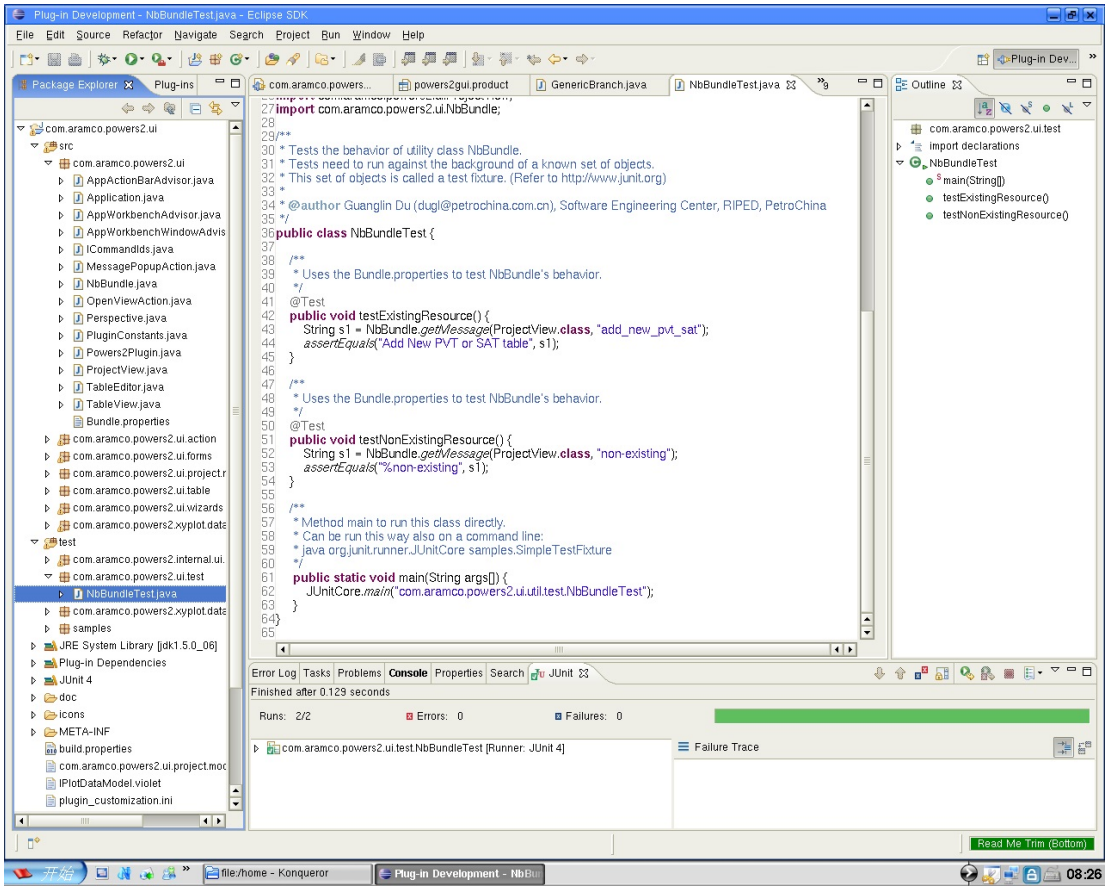

Abbildung 5.2: Eclipse (Screenshot)

[1] Siehe http://www.eclipse.org/screenshots/ (zuletzt besucht 27.10.2013)

Usability: 4/5
Performance: 3/5
Einarbeitungszeit: 3/5

5.2.1.3 Emacs

Emacs zählt zu den ältesten Vertretern der Open Source IDEs. Durch seine integrierte Python-API gibt es eine unüberschaubare Anzahl von Erweiterungen. Vor allem unter Linux gibt es eine große Community, die beständig an der Weiterentwicklung dieser IDE arbeitet und auch große Hilfestellungen in Foren leisten.

Vor allem mit der Mono-Compiler Umgebung kann Emacs gut umgehen, ist aber durch seine Flexibilität natürlich auch mit allen anderen Compilern gut zu gebrauchen. In diese IDE lässt sich quasi jedes externe Tool gut einbinden, wobei häufig mit Kommandozeilen-Optionen gearbeitet werden muss, sofern man nicht selbst eine visuelle Schnittstelle implementiert.

Dahingehend kann Emacs wahrscheinlich mehr als so manche andere Umgebung, so auch für Test-getriebene Softwareentwicklung, bedarf aber auch ein großes Maß an Erfahrung sowohl im Umgang mit Emacs selbst, als auch mit den entsprechenden Tools.

Abbildung 5.3[2] zeigt den Aufbau der Entwicklungsumgebung.

Usability: 2/5
Performance: 5/5
Einarbeitungszeit: 2/5

[2]Siehe http://ecb.sourceforge.net/screenshots/ (zuletzt besucht 27.10.2013)

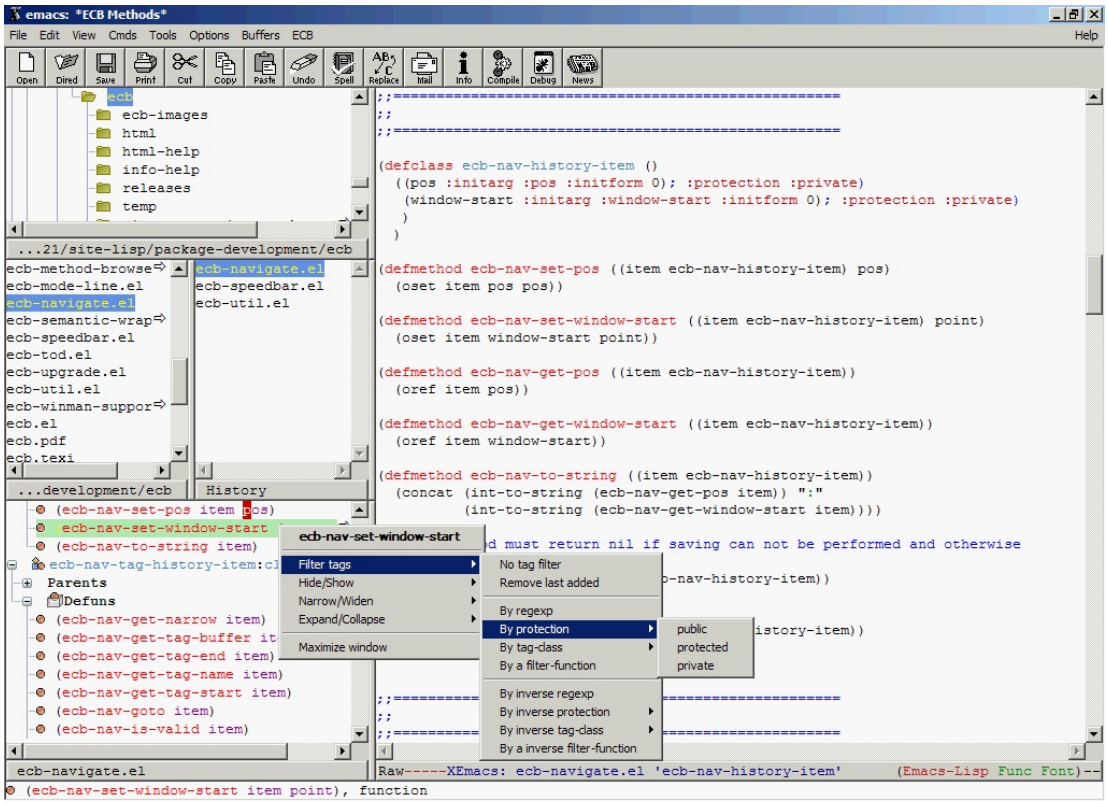

Abbildung 5.3: Emacs (Screenshot)

5.2.1.4 Vim

Ähnlich wie Emacs ist Vim ein Urgestein der Linux-Gemeinde. Vim ist eher als Editor, denn als Entwicklungsumgebung zu bewerten, bietet aber durch seine Flexibilität und Fähigkeit sich durch externe Tools zu erweitern eine durchaus saubere Lösung für die Test-getriebene Softwareentwicklung, sofern der Entwickler sich nicht vor verschiedenen Skriptsprachen und der verzahnten Interaktion mit dem Betriebssystem scheut.
Abbildung 5.4[3] zeigt den Aufbau der Entwicklungsumgebung.

Usability: 2/5
Performance: 5/5
Einarbeitungszeit: 2/5

[3]Siehe http://vim-taglist.sourceforge.net/screenshots.html (zuletzt besucht 27.10.2013)

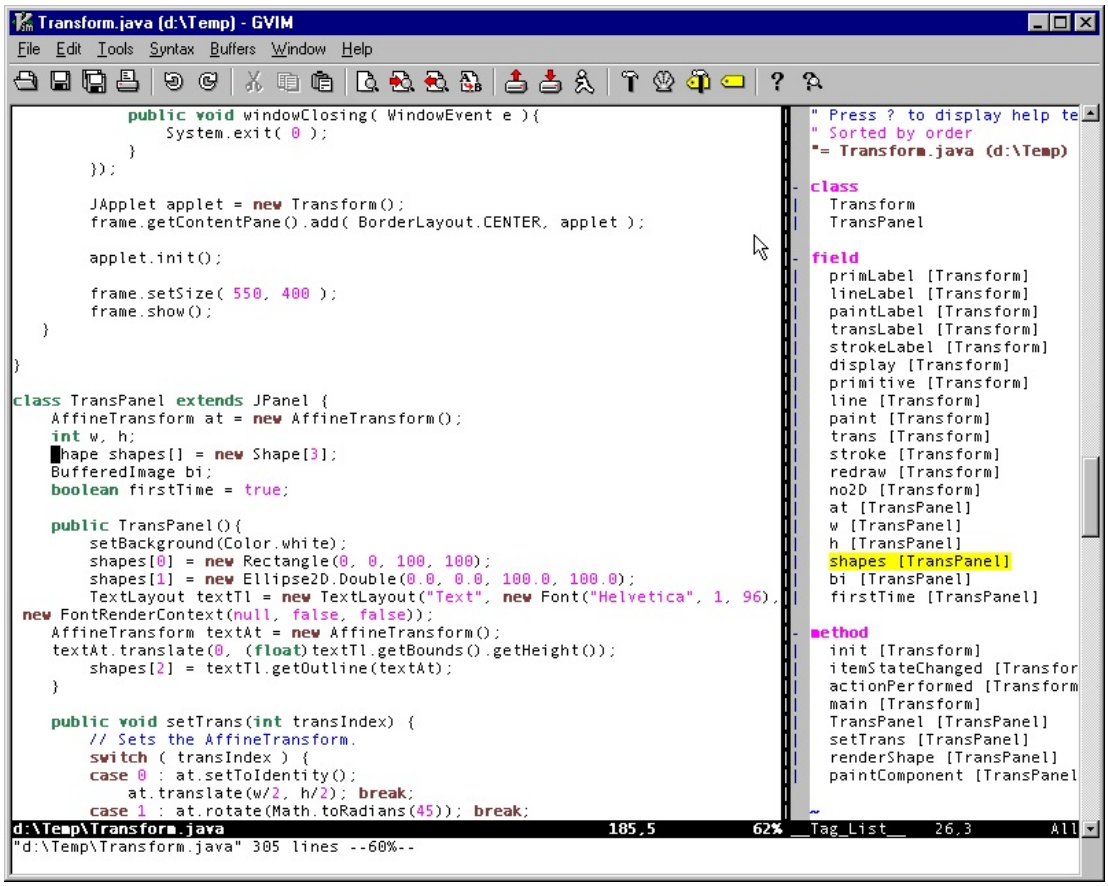

Abbildung 5.4: Vim (Screenshot)

5.2.1.5 Borland DevPartner Studio

Borland liefert mit DevPartner Studio, sowie dem Silk Portfolio eine große Anzahl an Testmöglichkeiten. Als einer der wenigen Anbieter von Load-Testing, aber natürlich auch Regression-Tests, UI-Tests, Performance-Tests und Build Automatisierung liefert DevPartner Studio ein umfassendes Werkzeug für die professionelle Entwicklung von Programmen, die aber auch preislich auf den professionellen Einsatz abzielt.

Abbildung 5.5[4] zeigt den Aufbau der Entwicklungsumgebung.

[4]Siehe http://documentation.microfocus.com/help/index.jsp?topic=%2Fcom.borland.silktest.doc%2FSILKTEST-FB884B57-VISUALNAVIGATOR-CON.html (zuletzt besucht 27.10.2013)

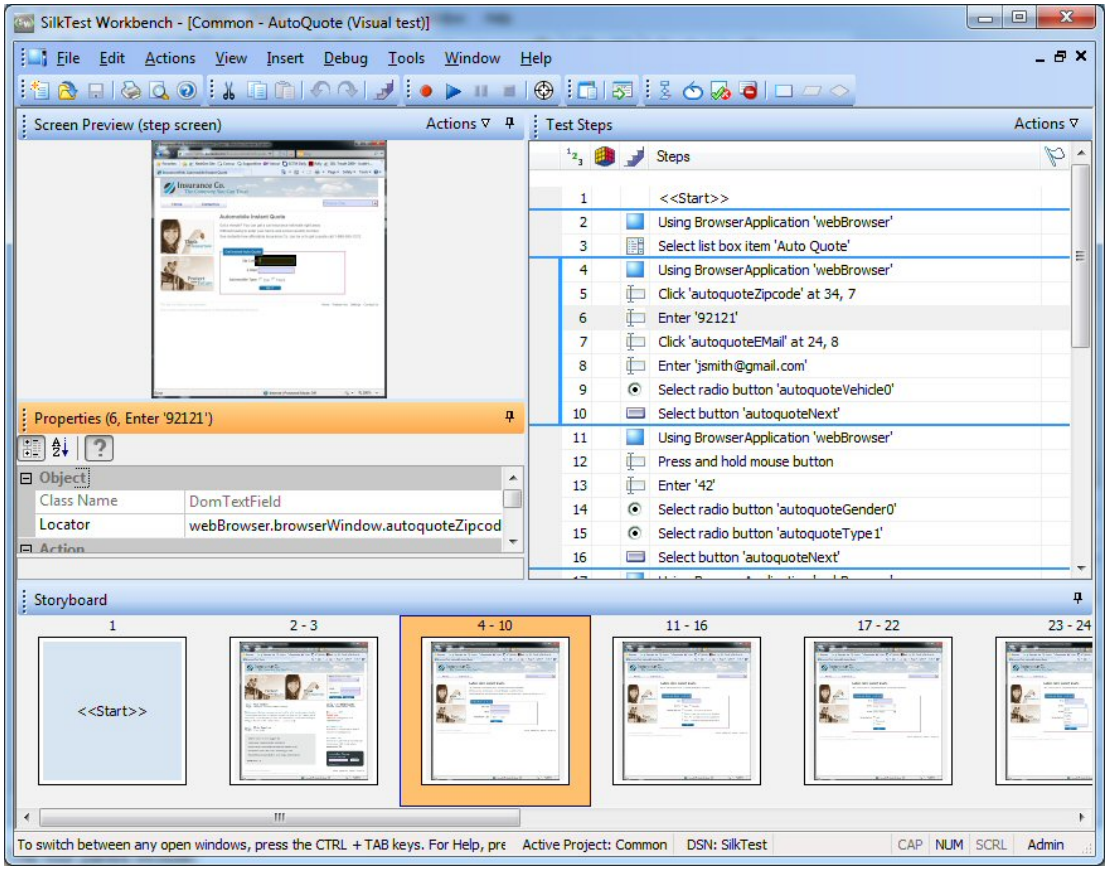

Abbildung 5.5: SilkTest (Screenshot)

Usability: 4/5
Performance: 4/5
Einarbeitungszeit: 4/5

5.2.1.6 MonoDevelop

MonoDevelop ist ein Open Source Projekt und erlaubt die Entwicklung von .NET Programmen auf unterschiedlichen Betriebssystemen. Es stammt vom ebenfalls noch verfügbaren Editor SharpDevelop ab - wurde aber danach noch in Hinblick

auf die Eigenheiten der Mono-Laufzeitumgebung erweitert. Es unterstützt zur Build Automatisierung 'Make' und liefert eine integrierte Umgebung für Unit-Tests. Darüber hinaus gibt es ein offenes API zur Erweiterung von MonoDevelop und erlaubt die Integration von externen Tools. Wem diese Ansprüche genügen kann MonoDevelop mit gutem Gewissen für die Test-getriebene Softwareentwicklung verwenden. Für den professionellen Einsatz bietet diese IDE eher geringe Möglichkeiten.

Abbildung 5.6[5] zeigt den Aufbau der Entwicklungsumgebung.

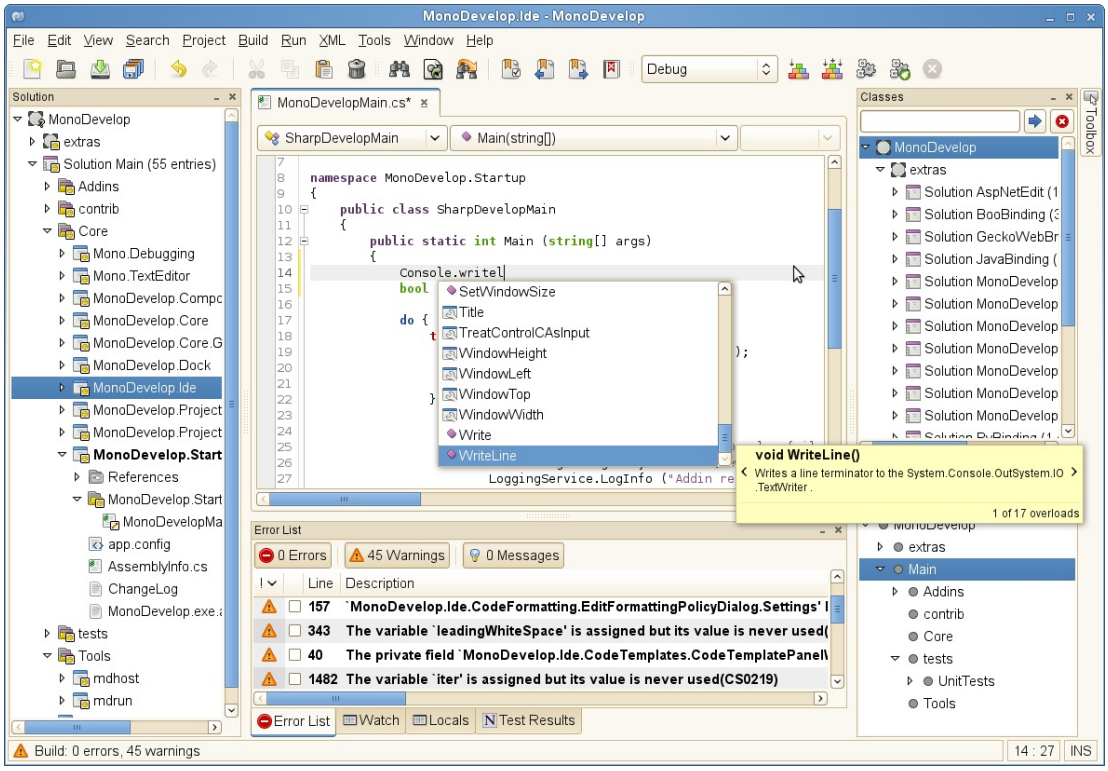

Abbildung 5.6: MonoDevelop (Screenshot)

Usability: 3/5
Performance: 3/5
Einarbeitungszeit: 5/5

[5]Siehe http://monodevelop.com/screenshots (zuletzt besucht 27.10.2013)

5.2.1.7 Sonstige Editoren

Natürlich kann die Entwicklung mit jedem beliebigen Editor erfolgen. Unter Windows mit Cygwin bzw. in den diversen Shells von Linux und Mac OS X ist dies auch durchaus auf produktiver Basis möglich, erfordert jedoch viel Erfahrung im Umgang mit Komandozeilen-Tools bzw. Skriptsprachen und empfiehlt sich nur, wenn die gewünschten Tools zur Test-getriebenen Entwicklung den Einzug in bestehende IDEs noch nicht geschafft haben, bzw. nur schwer möglich sind.

5.2.2 Compiler

Im Bereich der Compiler ist die relevante Auswahl relativ gering. Die meisten Entwickler werden wohl auf die Referenzimplementation von Microsoft selbst zurückgreifen. Auf Open Source Basis entwickelten sich zwei Strömungen, die es wert sind erwähnt zu werden. Diese sind

- dotGNU (GNU Compiler)
- DMCS (Mono Compiler)

Der Einsatz dieser Compiler hängt vor allem von der Laufzeitumgebung ab. Sie werden nur der Vollständigkeit halber erwähnt und spielen für diese Arbeit keine wesentliche Rolle bzw. machen für Test-getriebene Softwareentwicklung keinen Unterschied.

5.2.3 Analyse Tools

Im Grunde gibt es zwei unterschiedliche Arten von Analyse Tools

1. Statische Code-Analyse
2. Dynamische Code-Analyse

Bei der statischen Code-Analyse handelt es sich um Analyse-Methoden, die zur Compilier-Zeit Anwendung finden. Darunter fallen unter anderem die Erkennung von Race Conditions, Format-String Angriffen, Pufferüberläufen und Speicherlecks, aber auch - vor allem für Test-getriebene Softwareentwicklung - hilfreiche Methoden, wie die Erstellung von Objekt-Code zur Durchführung von Testabdeckung und -frequentierung, Einhaltung von Code-Convention (Programmierstil), Refactoring Tools und Werkzeuge zur Navigation und dem automatischen Erzeugen und Verifizieren von Quellcode.

Die dynamische Code-Analyse hingegen braucht für seine Funktionsweise das ausgeführte Programm. Regressions-Tests, Performance-Tests, Load-Tests, Memory-Checker und ähnliche Tests werden auf diese Weise durchgeführt.

5.2.4 Projekt Management Systeme

Während der Entwicklung von Software kann auf diverse Projektmanagementmethoden und -systeme zurückgegriffen werden. Dabei können zwei Arten von Softwaretypen unteschieden werden:

1. Issue Tracking Systeme, die als Informationsdatenbanken bei der Wissensverarbeitung während dem Projekt- und Aufgabenmanagement verwendet werden.

2. Echte Projekt Management Systeme, die beim operativen Ablauf der Projektmanagement Methode unterstützend wirken.

Unter einem Issue Tracking System versteht man ein Softwaresystem, das Angelegenheiten unterschiedlichster Herkunft bearbeiten, klassifizieren und bestätigen kann. Diese Ticket genannten Angelegenheiten können in der Softwareentwicklung sowohl Fehler, allgemeine Informationen, Dokumentation, und Funktionalitätswünsche darstellen.

Unabhängig vom tatsächlichen Einsatzgebiet dieser Software haben alle Systeme diverse Funktionalitäten gemeinsam:

- Zuweisung eines Ticket zu einer Person
- Status-Anzeige eines Tickets
- Bearbeitungs-Funktionalität für ein Ticket
- Klassifizierung von Tickets

Viele Issue Tracking Systeme bieten zusätzlich noch die Funktionalität, per Email oder andere Medien über Veränderungung zu informieren. Auch Kommentarfunktionalitäten sind ein häufiger Bestandteil solcher Software.

Für die Test-getriebene Softwareentwicklung ist ein solches System zwar nicht essentiell - jedoch allgemein für die agile Softwareentwicklung zu empfehlen.

In den folgenden Abschnitten werden einige solcher Issue Tracking Systeme exemplarisch vorgestellt. Auch hier spielt die Usability eines solchen Systems eine entscheidende Rolle, wobei die Entscheidung welches System geeignet ist stark von den Bedürfnissen der Benutzer abhängt.

Echte Projekte Management Systeme bilden den zeitlichen und Phasen-orientierten Ablauf einer Projektmanagement Methode ab.
Sie helfen dem Benutzer bzw. dessen Managern die Aufgaben während der Entwicklung zum richtigen Zeitpunkt und in angemessener Weise durchzuführen.

Der Übergang zwischen reinen Issue Tracking Sytemen und echten Projekt Management Systemen ist schwer abzugrenzen und hängt vor allem auch von der eingesetzen Projektmanagement Methode ab.

5.2.4.1 Asana

Asana ist ein relativ junges Issue Tracking System. Es beschränkt sich in seiner Funktionalität auf das reine Erstellen, Zuordnen, Kommentieren und Schließen von Tickets. Der Aufbau ist vergleichbar mit einem Wiki und versucht durch

seine schlichte Darstellung zu überzeugen anstatt den Benutzer durch Funktionalitäten zu überschwemmen.

Abbildung 5.7[6] zeigt einen Screenshot dieses Systems.

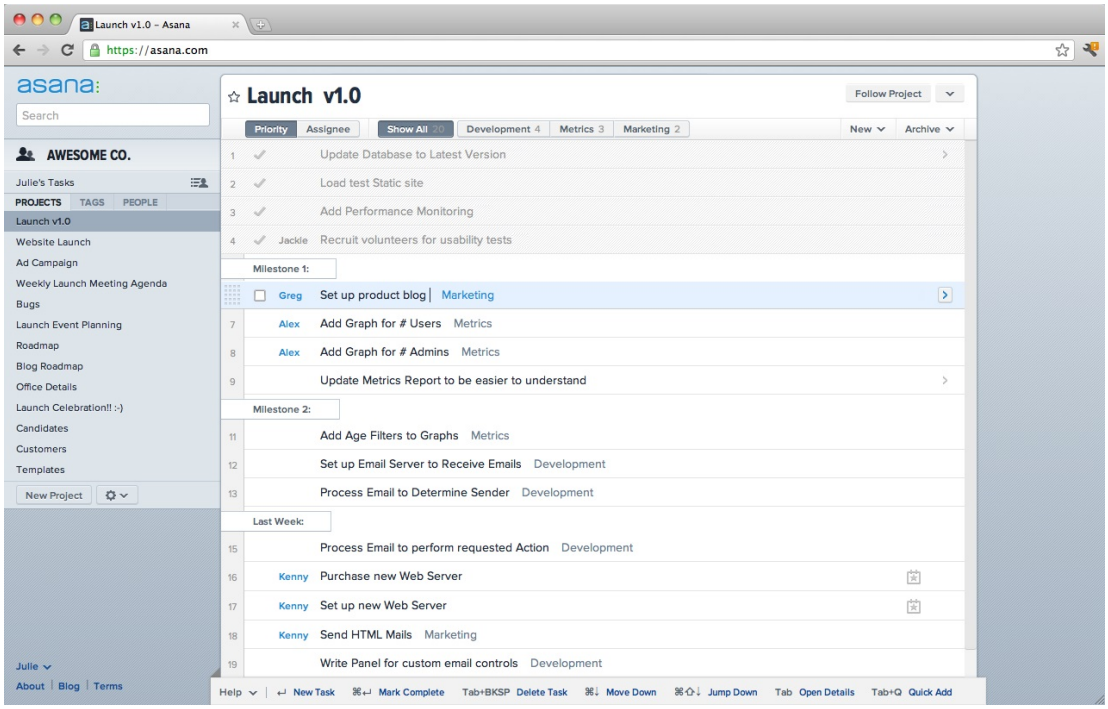

Abbildung 5.7: Asana (Screenshot)

Usability: 5/5
Performance: 5/5
Einarbeitungszeit: 5/5

[6]Siehe http://www.crunchbase.com/company/asana (zuletzt besucht 27.10.2013)

5.2.4.2 Bugzilla

Bugzilla ist ein Issue Tracking System, das hauptsächlich auf den Einsatz in der Softwareentwicklung ausgelegt wurde. Es erfordert die Installation auf einem eigenen Server und bietet eine Vielzahl an Erweiterungsmöglichkeiten an. Da es ein bereits seit geraumer Zeit existierendes Open Source System ist, gibt es eine große Community, die bei der stetigen Weiterentwicklung behilflich und auch in diversen Support-Foren tätig ist.

Die folgende Abbildung 5.8[7] zeigt das System in einem KDE-Browser.

Usability: 4/5
Performance: 4/5
Einarbeitungszeit: 3/5

5.2.4.3 Team Foundation Server

Team Foundation Server (TFS) ist eigentlich mehr als nur ein Issue Tracking System. Neben den 'einfachen' Aufgaben eines solchen Systems gestattet TFS die Planung und Durchführung gesamter Projektmanagement- und Testabläufe.
Durch die tiefe Verzahnung mit der Entwicklungsumgebung möchte damit Microsoft ein System etablieren, das alle Managementaufgaben, die im Zuge der Durchführung eines Softwareprojekts entstehen, abdeckt.

Abbildung 5.9[8] zeigt einen Screenshot dieses Systems.

[7] Siehe http://commons.wikimedia.org/wiki/File:Kde-bugtracking-via-bugzilla-firefox-1.0.6-kde-3.4.2-de.png (zuletzt besucht 27.10.2013)
[8] Siehe http://blog.ianuy.com/2010/01/24/laziness-impatience-and-hubris-or-how-to-download-work-item-attachments-programatically-using-tfs-sdk (zuletzt besucht 27.10.2013)

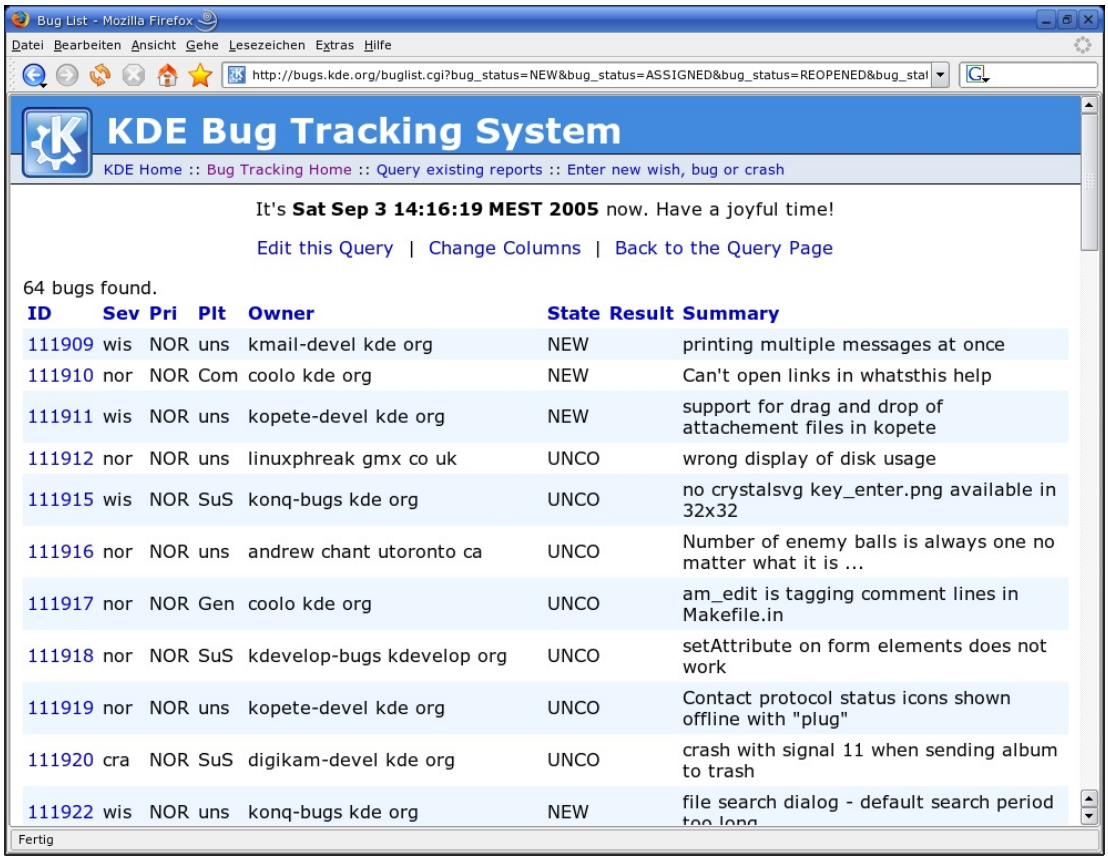

Abbildung 5.8: Bugzilla (Screenshot)

Usability: 3/5
Performance: 3/5
Einarbeitungszeit: 3/5

5.2.4.4 Track+

Track+ ist ein leichtgewichtiges, kommerzielles Issue Tracking System mit zusätzlichen Funktionalitäten für das Projektmanagement.

Durch Integrationsmöglichkeiten an diverse Versionsverwaltungssysteme kann es Aufgaben und Fehler direkt an den Codestellen markieren und dort Diskussionen

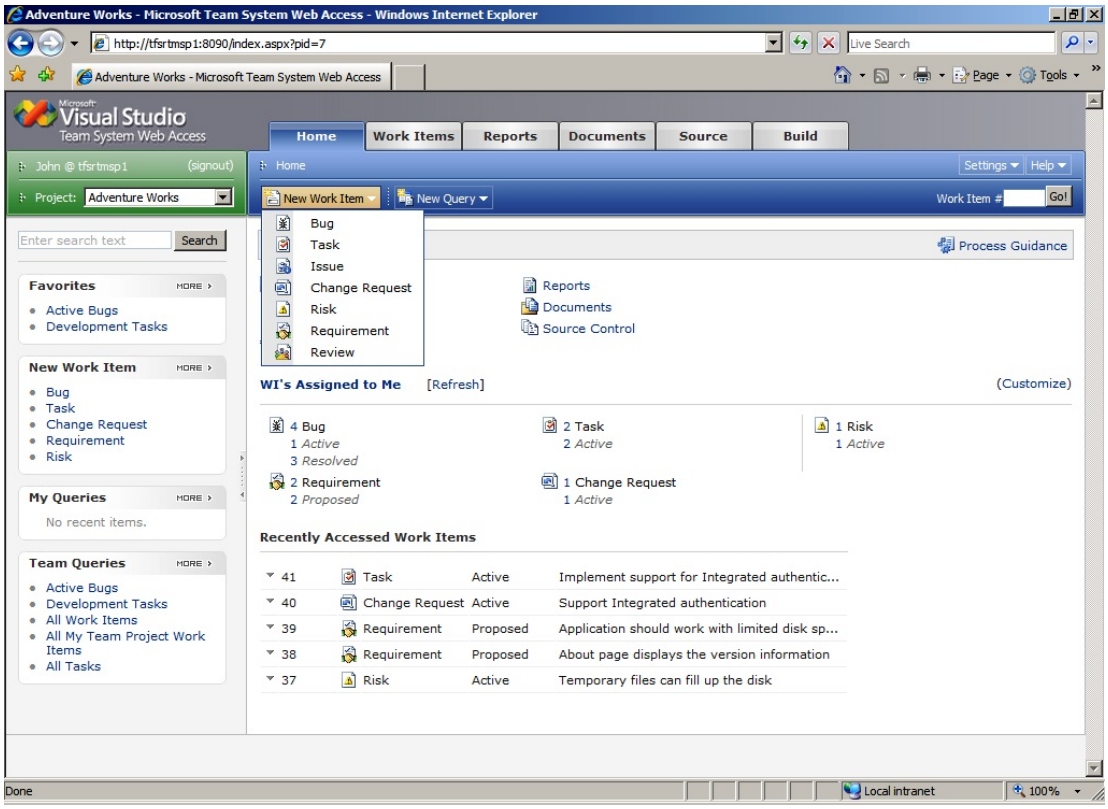

Abbildung 5.9: Team Foundation Server (Screenshot)

erzeugen, wo die Problemstellung relevant erscheint.

Track+ richtet sich vor allem an professionele Softwareentwickler, die interne Wissensdatenbanken und Projektmanagement-Workflows mit einem Issue Tracking System verbinden wollen.

Abbildung 5.10[9] zeigt einen Screenshot dieses Systems.

Usability: 4/5
Performance: 5/5
Einarbeitungszeit: 4/5

[9] Siehe http://www.trackplus.de/Funktionen/teamorientiertes-aufgabenmanagement.html (zuletzt besucht 27.10.2013)

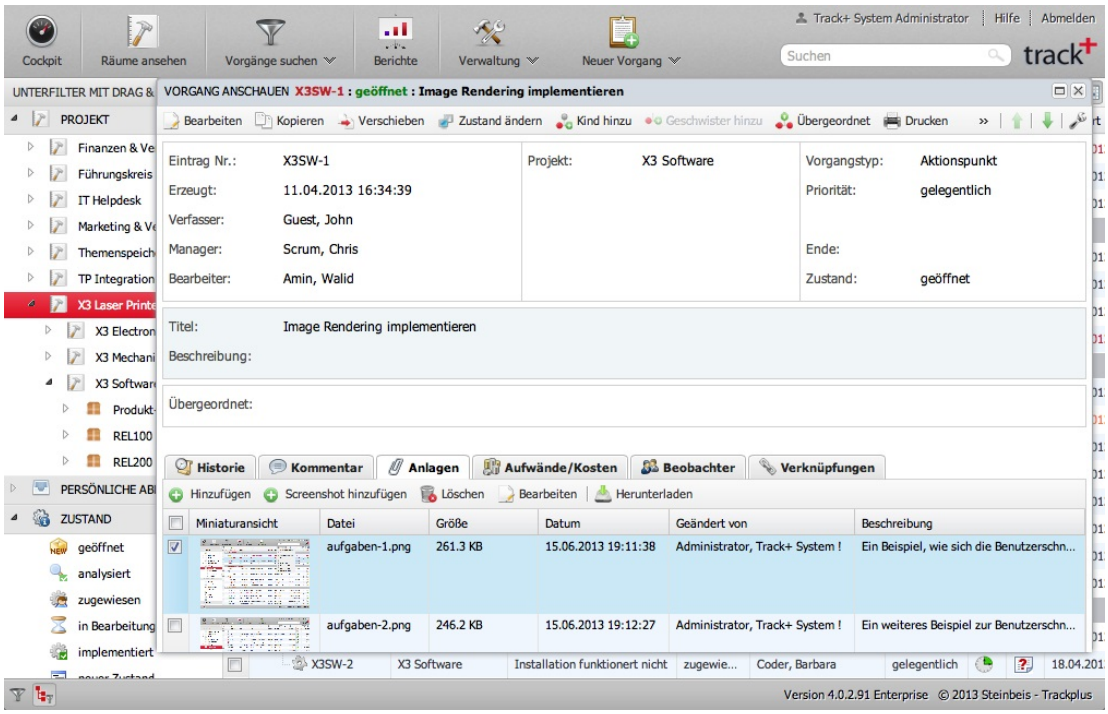

Abbildung 5.10: Track+ (Screenshot)

5.2.4.5 OpenProject

OpenProject versteht sich als Projekt-Management-System mit einer durchgehenden Integration als Issue Tracking System. Durch seine starke Community und den vielen Funktionalitäten ist es vor allem für große nicht-kommerzielle Projekte geeignet, für die ein einfacher Issue Tracker nicht ausreichend ist, aber die Kosten einer kommerziellen Lösung nicht getragen werden können.

Abbildung 5.11 zeigt einen Screenshot dieses Systems.

Usability: 4/5
Performance: 2/5
Einarbeitungszeit: 4/5

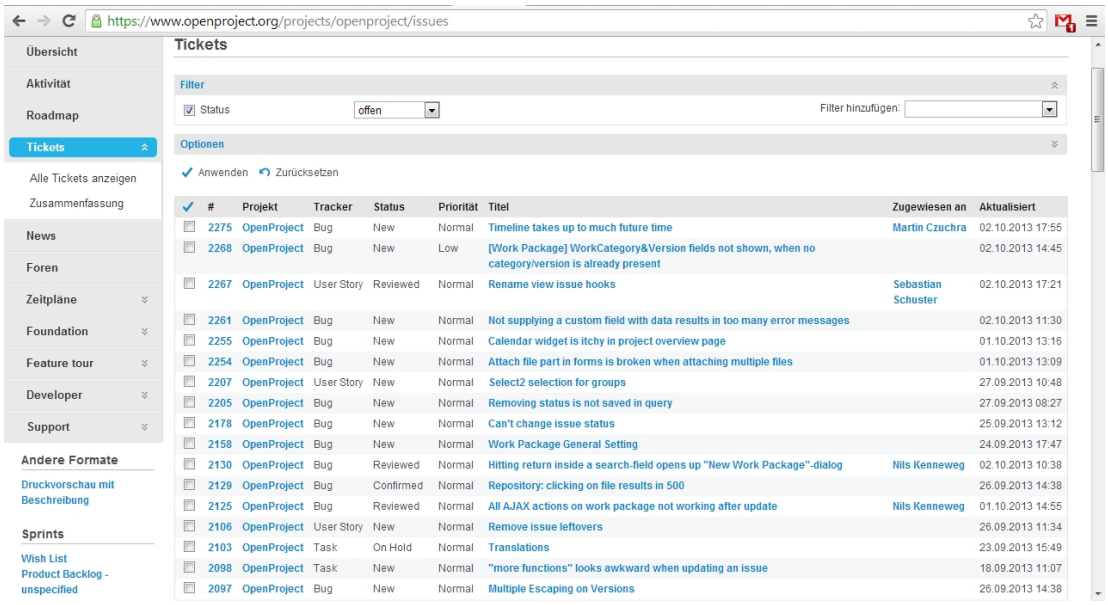

Abbildung 5.11: OpenProject (Screenshot)

5.2.5 Versionsverwaltungssysteme

Unter einem Versionsverwaltungssystem versteht man eine Kategorie von Software, die Änderungen von Dokumenten bzw. Dateien erfassen kann. Dabei wird jede Version einer Datei mit einem Zeitstempel versehen und bei den meisten Systemen auch mit einer Benutzererkennung, um eine gewünschte Version zu einem späteren Zeitpunkt wieder herstellen zu können.

In der Softwareentwicklung werden solche Systeme meist zur Verwaltung von Quelltexten verwendet, wobei in Büroanwendungen oder Content-Management-Systemen dies meist für alle Arten der damit zu verarbeitenden Dateien gilt.

Versionsverwaltungssysteme sind häufig nicht gleich als solche erkennbar und zeigen sich dezent, zum Beispiel in der Funktionalität der Änderungsnachverfolgung im Programm Word von Microsoft. Dabei werden Verfasser und Uhrzeit der Änderung in den Datei-Metadaten gespeichert.

Auch inkrementelle Backup-Systeme speichern häufig Redundanzen und Versio-

nen von Dateien aller Arten und zeigen diese Versionsverwaltung teilweise nur sehr unterschwellig (wie zB. bei Apple TimeMachine).

Als Sonderform des Variantenmanagements ist die Versionsverwaltung meist dazu geeignet unterschiedliche Varianten eines Softwareprodukts zu verwalten.

Das Archiv der versionierten Dateien wird Repository genannt. Die entsprechenden Programme arbeiten mit dem Repository und beziehen von dort aktuelle oder ältere Dateien bzw. fügen neue Versionen hinzu.

Bezieht ein Benutzer einen vollen Verzeichnisbaum versionierter Dateien, so wird die lokale Kopie als Arbeitskopie bezeichnet. Typischerweise stellt das entsprechende Versionierungssystem eine Programm bereit, das in der Lage ist den Verzeichnisbaum bzw. die Arbeitskopie mit dem Repository zu synchronisieren.

Unabhängig davon, ob das Programm kommandozeilenorientiert oder auf Basis einer grafischen Benutzeroberfläche arbeitet wird das Beziehen der Arbeitskopie als Checkout (Auschecken) und das Übertragen von Änderungen in das Repository als Check-in (Einchecken) bzw. Commit bezeichnet.

Im allgemeinen werden drei Arten der Versionsverwaltung unterschieden:

1. lokal: die Versionierung findet auf einem einzigen Arbeitsgerät statt

2. zentral: ein Client-Server System, wobei der Server die Versionierung vornimmt und der Client Änderungen durchführt

3. dezentral: jedes Arbeitsgerät hält ein Repository bereit, das mit anderen Repositories abgeglichen werden kann

Die folgenden Abschnitte erklären die Arten der Versionsverwaltung im Detail.

5.2.5.1 Lokale Versionsverwaltungssysteme

Die lokale Versionsverwaltung wird heutzutage fast ausschließlich in Form einzelner Dateien durchgeführt. Dabei dienen Metadaten innerhalb der Datei auch meist als Repository. Berühmte Vertreter dieser Versionierungsart sind:

- Microsoft Office Dateien
- Versions ab OS X Lion
- Autodesk 3D-Studio Max Dateien

5.2.5.2 Zentrale Versionsverwaltungssysteme

Die zentrale Versionsverwaltung besteht aus einem Client-Server-System und erlaubt damit den Zugriff auf das Repository über ein Netzwerk. Solche Systeme integrieren meist auch eine Rechteverwaltung, sodass je nach Berechtigung ein Benutzer das Repository bzw. Verzeichnisse und Dateien innerhalb dieser lesen, schreiben oder beides kann.

Berühmte Vertreter dieser Versionsverwaltungsart sind:

- CVS
- Subversion
- Team Foundation Server
- Alien Brain

5.2.5.3 Verteilte Versionsverwaltungssysteme

Die verteilte Versionsverwaltung stellt quasi eine Mischung aus den beiden vorhergehenden Arten dar. Jedes Arbeitsgerät hält selbst ein Repository, welches

nach belieben verändert werden kann. Bei Bedarf kann ein Repository mit einem anderen abgeglichen werden.

Bekannte Vertreter dieser Versionsverwaltungsart sind:

- GIT
- Bazaar
- Mercurial
- BitKeeper

6 Projektmanagementmethoden im agilen Entwicklungsprozess

Im Laufe der letzten Jahre haben sich eine Reihe von Projektmanagementmethoden etabliert, die ihrem Aufbau nach besonders geeignet sind im agilen Entwicklungsprozess Anwendung zu finden. Die folgenden Abschnitte sollen einen kurzen Überblick über etablierte Modelle bieten und dem Leser eine kleine Einführung in diese Projektmanagementmethoden geben.[1]

Abbildung 6.1 [2] soll schematisch den Ablauf eines agil-durchgeführten Softwareprojekts veranschaulichen.

6.1 Scrum

Scrum bedeutet auf Deutsch 'Gedränge' und bezeichnet eigentlich die Planungsphase im American-Football, in der die Spieler mit ihren Helmen zwischen zwei Spielzügen die Köpfe zusammenstecken und das weitere Vorgehen für den nächsten Spielzug besprechen.

Ähnlich wie beim Football geht es auch in der Softwareentwicklung um ein kleines verschworenes Team. Eine Grundregel beschreibt die ideale Anzahl der Teammitglieder als 7 +/- 2.

[1] Vgl. (Schneider, 2012) Seite 206-226

[2] Siehe http://www.it-informatik.de/factorplus/services/software_engineering/index.html (zuletzt besucht 27.10.2013)

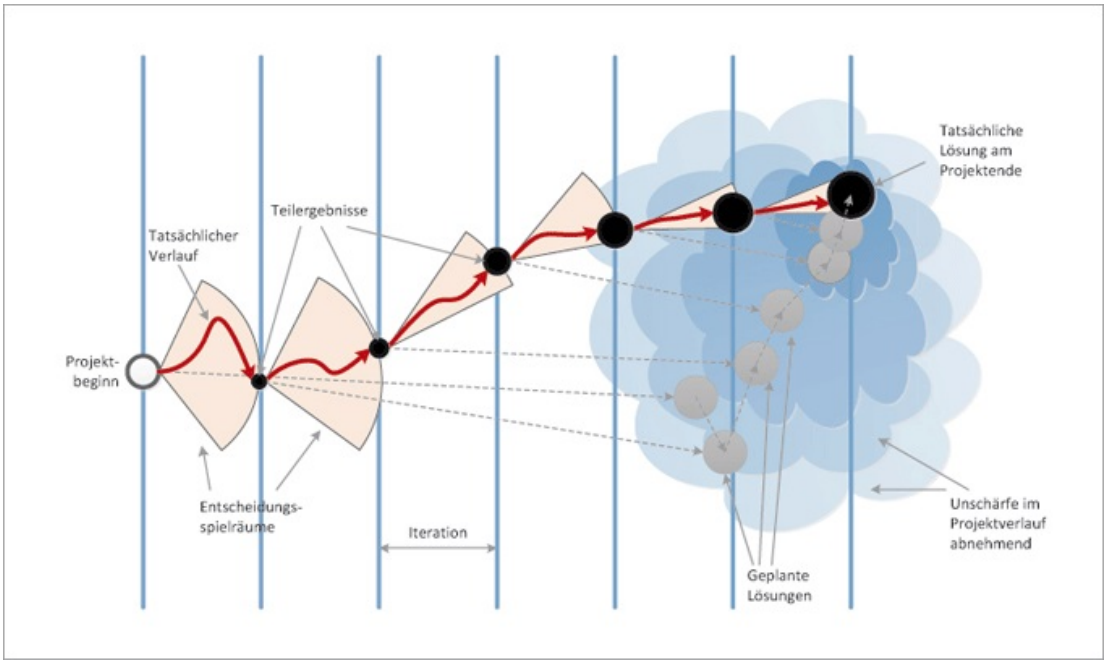

Abbildung 6.1: Agile Softwareentwicklung

Abgesehen vom Team selbst gibt es auch für weitere Stakeholder eine Rollenverteilung. Diese sind:

- Product Owner: führt ein Product Backlog, das die Aufgaben bzw. Wünsche für bzw. an das Team jederzeit in priorisierter Reihenfolge bereit hält.

- Scrum Master: führt in regelmäßigen Abständen Sprint-Meetings durch und sorgt für den reibungslosen Ablauf des Projekts

Der Product Owner hat gewisse Ähnlichkeiten mit dem On-Site Customer aus Extreme Programming, es ist jedoch nicht zwangsläufig erforderlich, dass dieser vor Ort sein muss (es zeigt sich jedoch hier eine angenehme Synergie). Seine Aufgabe ist es jederzeit nach seinem derzeitigen Wissensstand eine Liste von Produktfunktionalitäten (dem Product Backlog) bereit zu halten. Diese Liste ist seinem subjektiven Empfinden nach priorisiert und zeigt dem Team, wie wichtig dem Kunden welche Funktionalität in welcher Reihenfolge ist.

Alle 30 Tage beginnt mit dem sogenannten Sprint-Meeting eine neue Iteration, bei dem sich Kunde (Product Owner) und das Team zusammensetzen und die Kundenwünsche besprechen. Daraus entstehen für das Team eine Reihe von Aufgaben, die in einem Sprint-Backlog festgehalten werden. Diese Aufgaben sind bereits auf die Entwickler ausgerichtet und können nicht mehr zwangsläufig vom Kunden verstanden werden - sie stellen die eigentlichen Implementationsaufgaben dar. Jedes Sprint-Backlog soll Aufgaben im Umfang von einer Woche für das Team bereithalten, wobei jeder Entwickler für jede einzelene Aufgabe nicht länger als einen Tag brauchen sollte.

Jeden Tag treffen sich alle Teammitglieder zu einem 15-minütigen Daily Scrum, bei dem ein schneller Wissensaustausch stattfindet. Da es keine Diskussionsrunde ist, sondern jeder Entwickler nur seinen aktuellen Status kommuniziert, wird das Team nicht lange aufgehalten. Sollte es notwendig sein, Fragen zu beantworten oder Probleme zu besprechen, so können sich die betroffenen Entwickler nach dem Daily Scrum zusammensetzen um das Problem gemeinsam zu lösen.

Am Ende der 30 Tage dauernden Iteration trifft sich das Team erneut mit dem Product Owner, der idealerweise durch einen täglich neu verfügbaren Build bereits den aktuellen Status des Projekts kennt und seine Wünsche für den Verlauf des Sprints im nächsten Sprint-Meeting äussert bzw. im Product Backlog festhält, um die nächste Iteration zu starten.

Bei Scrum wird das Backlog häufig auch durch einen Burndown-Chart über die Zeit hinweg dargestellt. Das gibt dem Kunden die Möglichkeit selbst eine gute Abschätzung für die Restdauer des Projekt zu erhalten.

Der Ablauf einer Iteration von Scrum ist in Abbildung 6.2 [3] dargestellt.

Kritiker von Scrum bemängeln an dieser agilen Projektmanagementmethode vor allem die Iterationszeit von 30 Tagen. Diese ist im Zuge der zielgerichteten Arbeit mit dem Kunden zu lange, um Fehlentwicklungen des Produkts ausreichend zu

[3] Siehe http://commons.wikimedia.org/wiki/File:Scrum_process-de.svg (zuletzt besucht 27.10.2013)

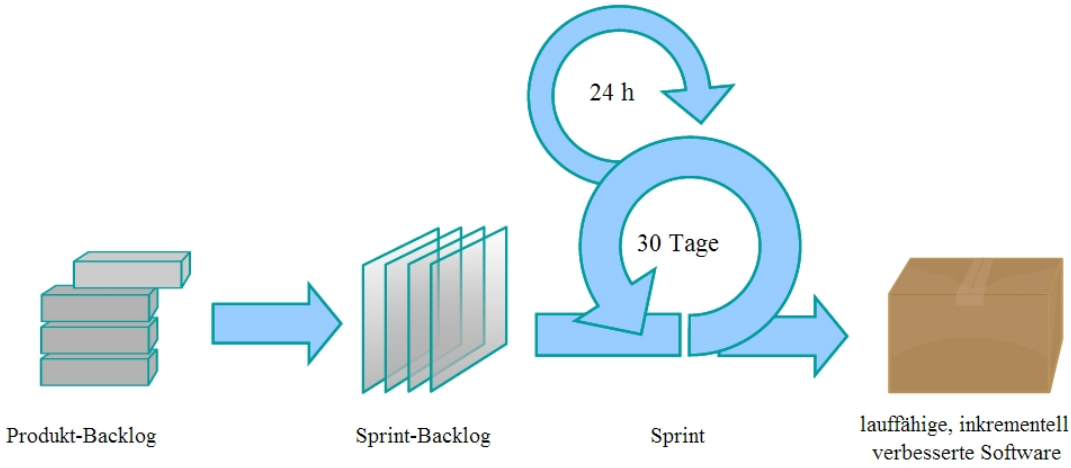

Abbildung 6.2: Scrum Ablauf

verhindern.

6.2 Lean Software Development

Lean Software Development bedeutet die Softwareentwicklung so schlank wie möglich zu halten. Im Gegensatz zu vielen anderen Projektmanagementmethoden greift dieser Prozess also auf der Management-Seite, während die meisten anderen agilen Methoden bei der Entwicklung selbst ansetzen.

Lean Software Development wurde im Jahr 2006 durch das gleichnamige Buch von Mary und Tom Poppendieck vorgeschlagen und entstammt ursprünglich der Automobilindustrie. Dort konnte der japanische Autohersteller Toyota durch hohe Materialpreise und geringe Absätze mit der amerikanischen Massenindustrie nicht mithalten. Im Buch Toyota Production System von Ohno wird das Prinzip beschrieben, jegliche Verschwendung so weit als möglich zu eliminieren. Ohno zählt hier 7 Arten der Verschwendung auf:

1. Überproduktion

2. unnötiger Transport
3. Zwischenlagerbestände
4. Bewegung
5. Fehler
6. Überarbeitung
7. Warten

Diese Arten der Verschwendung sind zwar nicht alle direkt auf die Softwareentwicklung anzuwenden, es lassen sich jedoch leicht Analogien finden.

Lean Software Development stellt 8 Prinzipien vor, die dabei helfen sollen die Entwicklung effizienter zu gestalten:

- Eliminieren von Verschwendung
- Visualisieren der Wertschöpfungskette
- Fördern von Qualität
- Fördern von systematischem Lernen
- Festlegungen hinausschieben
- schnell entwickeln
- Respekt gegenüber anderen Personen
- bei Optimierung das Ganze im Auge behalten

Durch die ständige Beachtung dieser Prinzipien soll nach dieser Methode in kürzerer Zeit ein höherer Kundennutzen entstehen.

6.3 Kanban

So wie Lean Software Development entstammt auch Kanban der japanischen Automobilindustrie. Dabei greift Kanban jedoch stärker auf Praktiken zurück als auf Prinzipien.

Zentrales Element dieser Methode stellt das sogenannte Kanban-Board dar, ein Whiteboard, welches Arbeitsabläufe visualisieren soll. Kanban geht hier davon aus, dass die Umsetzung von mehreren Aufgaben gleichzeitig durch eine Person unwirtschaftlicher durchgeführt wird, da ständige Umdenkprozesse erfolgen.

Durch das Einschränken dieser gleichzeitigen Aufgaben soll ein flüssiger und durchgehender Prozess für den Entwickler entstehen. Die zugeteilten und gerade zu bearbeitenden Aufgaben werden 'Work in Progress' (kurz WIP) genannt.

Bei der Visualisierung des Prozesses einer Person auf dem Kanban-Board wird also zu jedem einzelen Schritt (Anforderung, Entwurf, Implementierung, Testen, Fertig) die Anzahl der WIP-Items begrenzt.

Das Pull-Prinzip bedeutet, dass einem gesamten Prozess keine weitere Aufgaben hinzugefügt werden, da dieses sonst 'verstopfen' könnte. Die Aufgaben wandern also Schritt für Schritt in Richtung 'Fertig' bis Platz genug für weitere Aufgaben besteht.

Letztlich ist das Prinzip Lernen noch gesondert in dieser Methode hervorzuheben: Da das System einem Evolutionsprozess unterliegt, können und sollen alle Beteiligte Verbesserungsvorschläge und Probleme so früh als möglich melden um entsprechend reagieren zu können.

Abbildung 6.3 [4] zeigt das Pull-Prinzip von Kanban auf eine Produktionslinie angewandt.

[4] Siehe http://www.manufactus.com/portfolio/push-vs-pull/ (zuletzt besucht 27.10.2013)

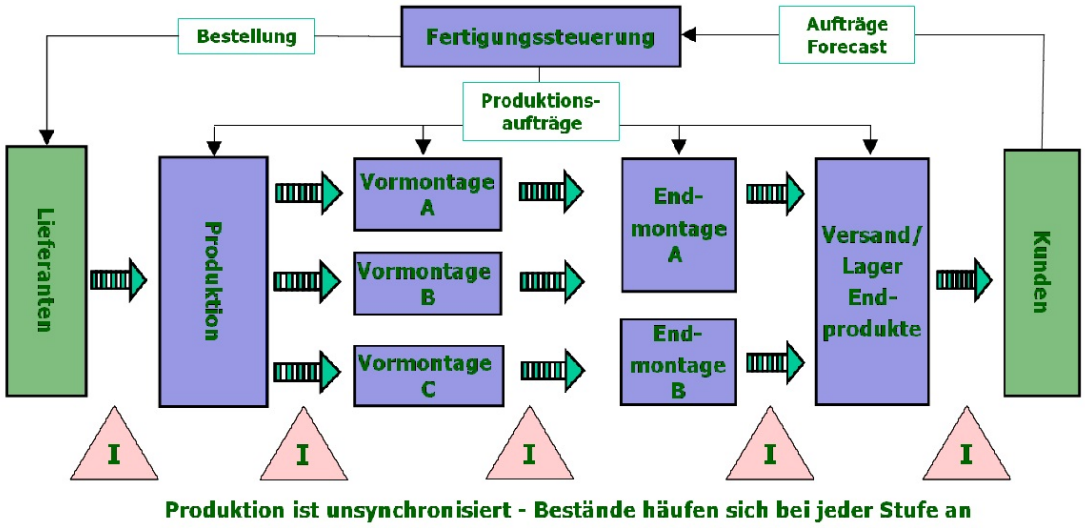

Abbildung 6.3: Pull-Prinzip von Kanban

6.4 Feature Driven Development

Feature Driven Development (kurz FDD) ist eine agile Softwareentwicklungsmethode, die vor allem in Unternehmen eingesetzt wird bzw. werden kann, die einen Umstieg auf Scrum oder ähnliche größere Eingriffe in die Unternehmenskultur noch nicht wagen wollen.

FDD ist schnell erklärt und relativ leicht durchgeführt. Dabei verläuft die Umsetzung in 5 Phasen:

1. Gesamtmodell entwickeln
2. Feature-Liste erstellen
3. jedes Feature planen
4. jedes Feature entwerfen
5. jedes Feature konstruieren

Im ersten Schritt treffen sich alle Projektbeteiligte und definieren gemeinsam ein Gesamtmodell, u.z. den Umfang und Inhalt der zu erstellenden Software. Sobald ein Bereich als gesondertes Subsystem definiert wird, spaltet sich die Gruppe in Teilgruppen auf, wobei je eine Teilgruppe den Umfang und Inhalt des Subsystems definiert - nach dem Prinzip 'Divide and Conquer'.

Im zweiten Schritt oder Phasenabschnitt, erstellen alle beteiligten Chefentwickler für das Gesamtsystem bzw. auch für die Subsysteme eine Feature-Liste nach einem dreistufigen Schema:

1. Fachgebiete extrahieren

2. Fachgebiete in Geschäftstätigkeiten unterteilen

3. Geschäftstätigkeiten durch Schritte zur Umsetzung definieren

Nach diesem Schema entspricht jeder Schritt zur Umsetzung einem Feature.
Um eine definierte Granularität zu erreichen wird den Chefentwicklern die Vorgabe gestellt, dass jedes resultierende Feature nicht länger als zwei Wochen zur Implementation benötigen darf.
Das Resultat dieser Phase ist eine kategorisierte Feature-Liste.

In der dritten Prozessphase setzen sich Projektleiter und Chefentwickler zusammen, um jedem Feature einen Besitzer zuzuordnen. Dabei nehmen sie Bedacht auf die Auslastung des Besitzers, dem Umfang des Features und Abhängigkeiten zwischen einzelnen Features.

Die vierte Phase steht für den Entwurf jedes einzelnen Features und beschäftigt sich mit deren Verteilung auf die Teams und Teammitglieder. Dabei muss jeder Entwickler im Team die ihm zugeordneten Features durch Sequenzdiagramme, Modul- und Klassendefinitionen beschreiben, um den Entwurf des Features im Anschluss wieder mit dem zugeordneten Chefentwickler zu besprechen, bzw. nachzubessern.

In der letzten Prozessphase erfolgt die Umsetzung eines jeden Features in der be-

sprochenen Reihenfolge. Zur Qualitätssicherung werden Unit-Tests und Reviews eingesetzt.

Die letzten beiden Phasen müssen nicht immer für alle einem Programmierer zugeordneten Features passieren, sondern können auch einzeln entworfen und dann umgesetzt werden.

In diesem Fall durchläuft der Entwickler zyklisch immer wieder die 4. und 5. Prozessphase.

Der Ablauf der Prozessphasen ist in Abbildung 6.4 dargestellt.

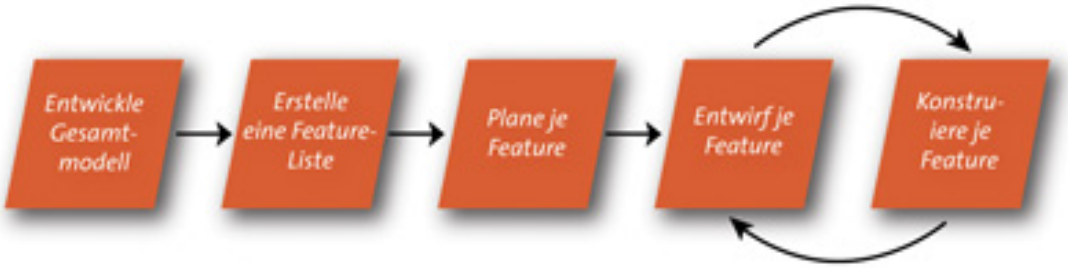

Abbildung 6.4: Feature Driven Development Prozess

6.5 Prototyping

Die Softwareentwicklungsmethode Prototyping ist eher als Sub-Projektmanagementmethode, statt als vollständiger Softwareentwicklungsprozess zu verstehen.

Unter Prototyping können unterschiedliche Arten verstanden werden:

- Exploratives Prototyping
- Evolutionäres Prototyping
- Experimentelles Prototyping

- Rapid Control Prototyping

Beim explorativen Prototyping wird Software geschrieben, um eine Orientierung über die Anforderungen des zu erstellenden Produkts zu bekommen.

Das evolutionäre Prototyping beschreibt die Entwicklung eines teilweise vollständigen Softwaresystems und kann bis zur Fertigstellung des endgültigen Produkts durchgeführt werden.

Unter experimentellem Prototyping versteht man die Evaluierung von möglichen Problemlösungen. Im Unterschied zum explorativen Prototyping ist der experimentelle Ansatz noch kürzer und zielgerichteter. Er wird vor allem in der Forschung oder als Proof-of-Conecpt eingesetzt.

Das Rapid Control Prototyping ist die Entwicklung von Software mithilfe grafischer Hilfsmittel. Sie wird für allem für Regel- und Steuerungssysteme verwendet.

Bis auf die evolutionäre Methode, entsteht normalerweise aus dem Prototypen kein fertiges Produkt - die Erzeugnisse dieser Methoden werden daher auch häufig als Wegwerf-Prototypen bezeichnet.

Alle funktionalen Prototypen können auf zwei verschiedene Arten produziert werden:

- horizontal: dabei wird nur ein Teil des Systems umgesetzt - dieses jedoch vollständig. Dadurch kann mit der geplanten Implementation des fertig evaluierten Systems begonnen werden, während andere Implementationsfragen noch nicht geklärt wurden.

- vertikal: ein System wird zur Gänze umgesetzt, aber nur in einer gewissen Ebene, d.h., grundlegende Funktionalitäten fehlen häufig und sind teilweise auch noch nicht geplant. Übliche Vertreter vertikaler Prototypen sind die Entwicklung grafischer Oberflächen zur Vorlage für den Kunden, ohne dass die eigentliche Funktionalität besteht (Proof-of-Design).

Abbildung 6.5[5] veranschaulicht die Zusammenhänge zwischen vertikalen und horizontalen Prototypen in Bezug auf ihre Funktionalitäten.

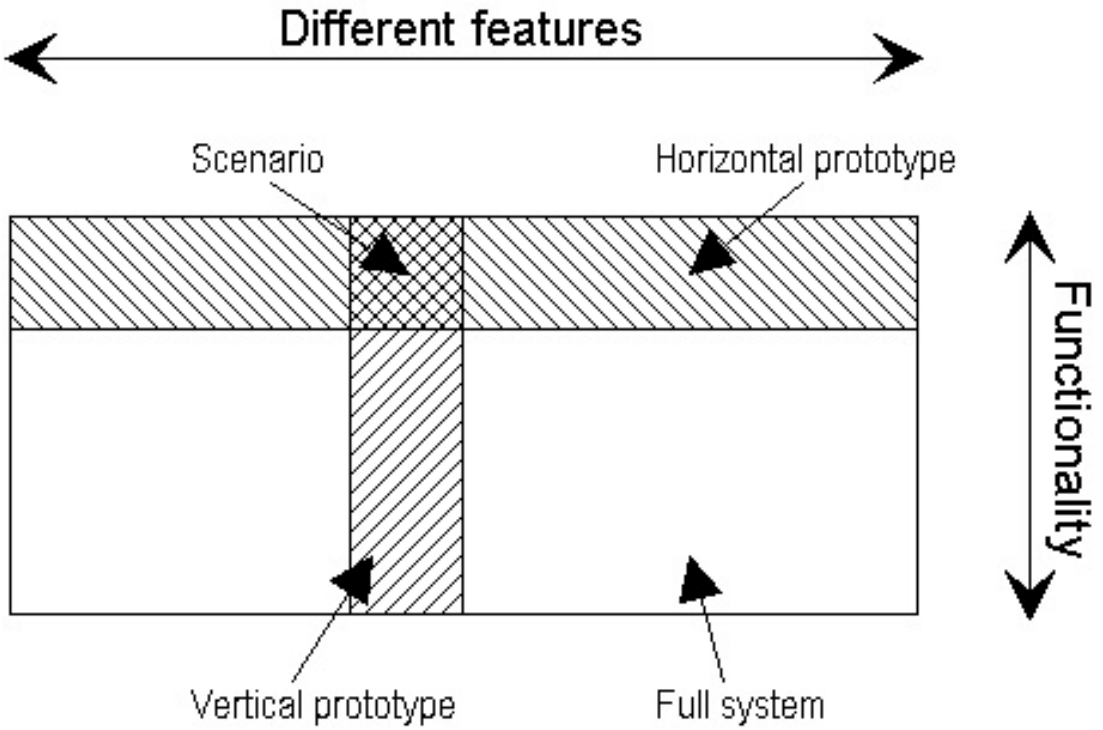

Abbildung 6.5: Vertikales und Horizontales Prototyping

[5]Siehe http://cartoon.iguw.tuwien.ac.at/fit/fit01/prototyping/konzepte.html (zuletzt besucht 27.10.2013)

7 Feldstudie VisioCAPAD

7.1 Projektbeschreibung

Die folgenden Abschnitte beschreiben den Zweck und die Zielgruppe des entwickelten Produkts und dienen dem Leser zur Einführung und Übersicht in die produzierte Software.[1]

7.1.1 Zweck des Produkts

Der Kunde und zukünftige Benutzer des zu erstellenden Tools VisioCAPAD arbeitet im industriellen Umfeld der Analyse und Optimierung von Produktionsprozessen.

Mit Hilfe von linearen Gleichungssystemen ist die Lösung von Masse- bzw. Energiesystemen möglich, die im Zuge einer solchen Produktion häufig Grundlage der Evaluierung eines solchen Prozesssystems sind. Um einen Ausgangspunkt für die Aufstellung einer Optimierungsanalyse zu ermöglichen, enthält VisioCAPAD eine Reihe von branchenüblichen Vorlagen.

VisioCAPAD ermöglicht dem Benutzer in Microsoft Visio ein Abbild eines bestehenden Energie- oder Masseprozess zu erstellen. Durch erweiterte Eingabemasken kann direkt in Visio der Benutzer dieses Abbild mit gemessenen oder exakt verfügbaren Daten befüllen, um unbekannte Masseströme, Energien oder Temperaturen

[1] Vgl. Patrick Hofmann (2013)

zu ermitteln.
Das Tool trifft selbst keine Annahmen – Entscheidungen sind vom Nutzer zu treffen. Werte, die aus Rechenschritten folgen, welche über die Bilanzierung hinausgehen, sind ebenfalls vom Benutzer anzugeben.
Microsoft Visio wird um zusätzliche Schablonen für einige Branchen erweitert.

7.1.2 Zielgruppe (vorausgesetzte Fähigkeiten und Kenntnisse)

Als wahrscheinlichste Zielgruppen gelten: Auditoren, Wartungspersonal, Energieberater und Lebensmitteltechnologen, sowie Personen mit besonderem Interesse an Masse- und Energieflüssen und deren Visualisierung in der Industrie.

Um mit VisioCAPAD arbeiten zu können sollten Benutzer grundsätzliche Erfahrungen mit dem Tool Microsoft Visio haben, sowie ein Basiswissen über lineare Gleichungen, spezifische Wärmekapazität und Bildungsenthalpien besitzen.

7.2 Projektplanung

Bei der Planung des Projekts wurde in einem ersten gemeinsamen Meeting die eigentliche Spezifikation und Projektgröße definiert. Die nachfolgenden Abschnitte sollen einen Einblick in diese geben.

7.2.1 Spezifikation

Ausgehend von obiger Produktbeschreibung wurde dem Entwickler die Software 'Stenum' und 'Procede' vorgeführt. Die eigentliche Problematik und Hintergrund der Entwicklung bestand in der Unfähigkeit, diese Software auf aktuellen Betriebssystemen ausführen zu können. Der Quellcode von Stenum lag in einer veralteten Version bereit - zu Procede gab es nur eine Dokumentation in ausgedruckter Form.

Die damit sehr direkte und daher einfache Spezifikation lag in der Neu-Implementierung der Funktionalitäten beider Programme, wobei auch Microsoft Visio 2010 zur Umsetzung von Fließdiagrammen benutzt werden konnte.

Der Umfang der Projektgröße wurde somit weitestgehend durch vorliegende Software vorgegeben und die Abschätzung der Entwicklungszeit wurde dem Entwickler überlassen.

Der Auftraggeber stellte einen wissenschaftlichen Mitarbeiter zur Verfügung, der sowohl für das Testen der Software auf Kundenseite (also für die Akzeptanztests) zuständig war, als auch Fragen zum Algorithmus zur Lösung eines Problems beantworten sollte.

Zur Beschreibung der verwendeten Algorithmen wurde dem Entwickler die Dissertation von Florian von Linde, dem ursprünglichen Entwickler von Stenum vorgelegt.

7.2.2 Interaktion mit dem Kunden

Die Umsetzung des Projekts wurde als Werkvertrag definiert, wobei wöchentliche Meetings den fortlaufenden Projektfortschritt evaluieren sollten.

Neben diesen wöchentlichen Meetings wurde beschlossen, zusätzliche Diskussionen per Email durchzuführen und monatlich ein größeres Meeting, welches Ausblick über die weitere Entwicklung des Projekts geben sollte.

Die Zahlung wurde in zwei Tranchen aufgeteilt, wobei der erste Teil mit Projektstart erfolgte und der zweite Teil mit Projektende.

7.3 Projektverlauf

Die folgenden Kapitel sollen dem Leser Einblick in den Prozess der Entwicklung des beschriebenen Produkts geben.

7.3.1 Analyse bestehender Software

VisioCAPAD ist im Wesentlichen ein Folgeprojekt des aus einer Doktorarbeit hervorgegangenen Produkts 'Stenum' (Bezeichnung für SToff-ENergie-UMwelt). Diese von Florian von Linde im Jahr 1993 durchgeführte Arbeit war einzigartig auf ihrem Gebiet und blieb es bis zum Abschluss dieser Arbeit. Abbildung 7.1 zeigt einen Screenshot des aus dieser Arbeit resultierten Produkts.

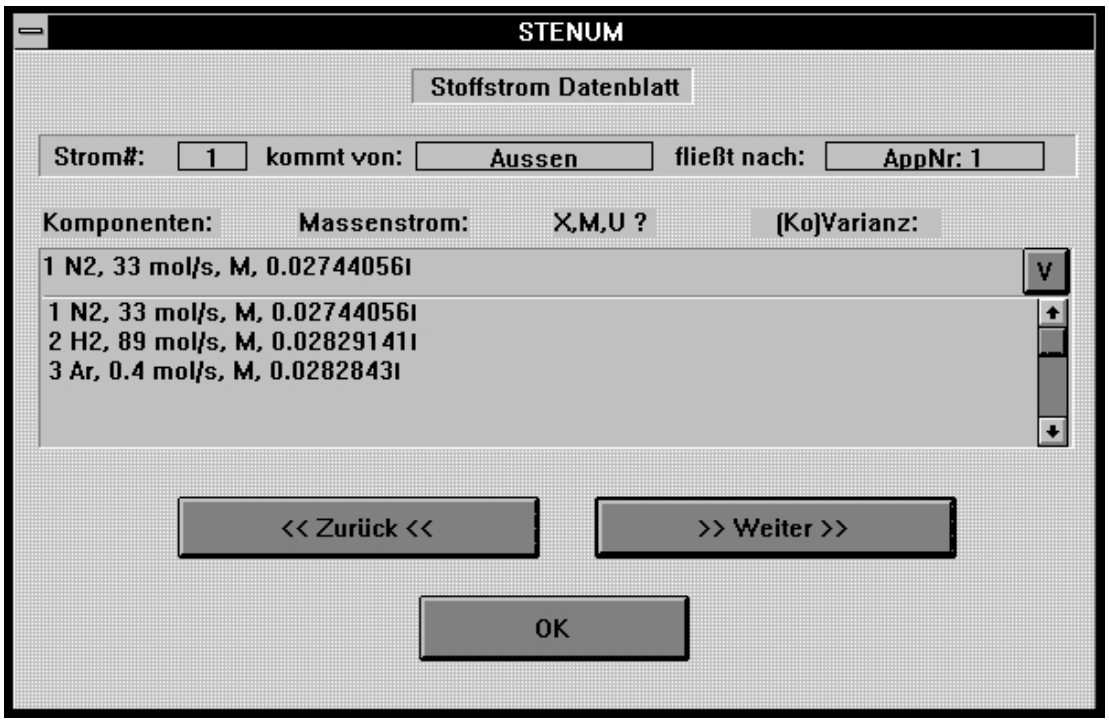

Abbildung 7.1: Stenum (Screenshot)

Da Stenum nur zur grundsätzlichen mathematischen Lösung gedacht war, wurde zur Erstellung von Fließbildern das Programm 'Procede' benutzt. Abbildung 7.2 zeigt einen Ausschnitt aus diesem Programm.

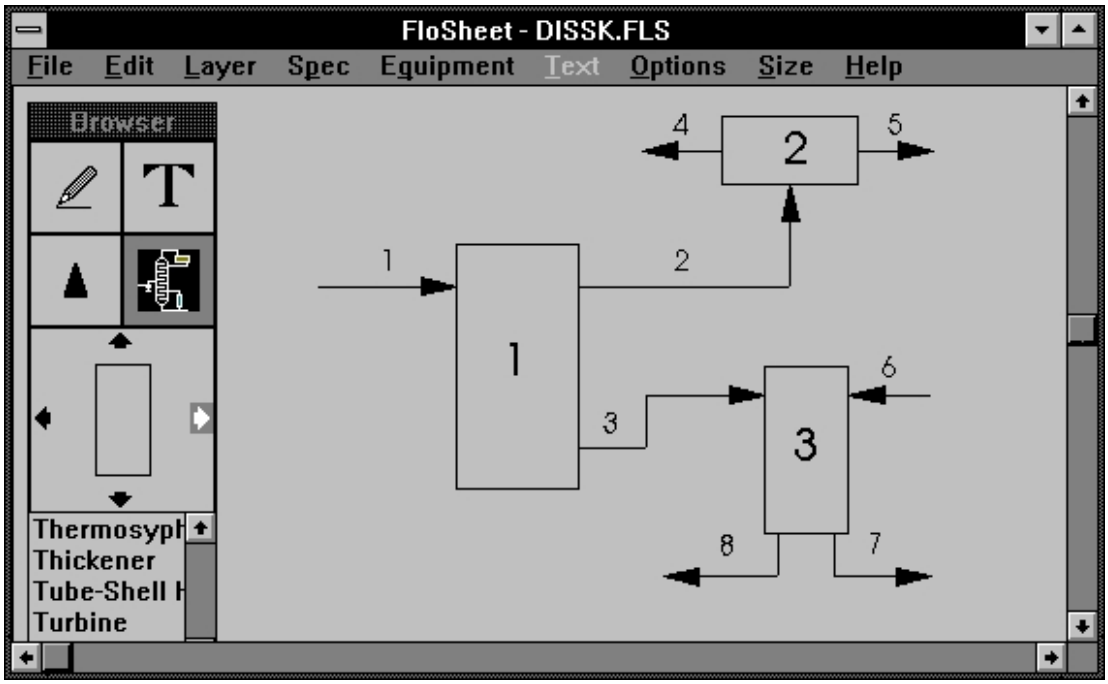

Abbildung 7.2: Procede (Screenshot)

Stenum basiert grundsätzlich auf der algorithmischen Lösung unter-, exakt- und überbestimmter Gleichungssysteme. Dahingehend - jedoch natürlich ohne spezialisiertem User-Interface - gibt es diverse wissenschaftliche Software, die der Lösung solcher Gleichungssysteme mächtig sind. Populäre Vertreter dieser Softwareprodukte sind:

- Mathematica
- MatLab
- Maple

Teilweise sind auch integrierbare Software-Bibliotheken zur Umsetzung von Gleichungssystemen verfügbar. Aufgrund des ausdrücklichen Kundenwunsches auf Spezialfälle einzeln reagieren zu können, wurde jedoch beschlossen die algorithmisch bekannte Lösung selbst zu implementieren.

7.3.2 Argumentation und Analyse verwendeter Entwicklungssoftware

Der Kunde wollte eine in Microsoft Visio 2010 integrierte Umsetzung - dem quasi Industriestandard für Diagramm-Software. Die Entwicklung hierfür erforderte den Einsatz der Office Developer Tools for Visual Studio 2012.

Wie hier der Name schon suggeriert, ist dies eine Erweiterung der Entwicklungsumgebung Visual Studio 2012, welche dahingehend auch als IDE verwendet wurde. Das Kapitel 5.2.1.1 beschreibt bereits umfassend die in dieser Umgebung integrierten Werkzeuge für Test-getriebene Softwareentwicklung, welche zur Umsetzung von VisioCAPAD ebenfalls verwendet wurden.

Zur Versionierung wurde das Programm 'Subversion' verwendet, da bereits ein externer Server hierfür zur Verfügung stand und andere Versionierungssysteme bei einer Ein-Mann-Entwicklung keine bedeutenden Vorteile erkennen ließen.
Die Kommunikation mit dem Kunden wurde auf drei Wegen durchgeführt:

- E-Mail zur Rücksprache bei Unklarheiten, algorithmischer Eigenheiten und 'spontanen' Funktionalitätswünschen
- Asana zur Diskussion von Fehlern
- Meetings zur Präsentation des aktuellen Standes und Diskussion der durch den Entwickler nicht reproduzierbaren Fehler.

Die Installation wurde durch den in Visual Studio 2012 integrierten Microsoft ClickOnce Installer implementiert, da andere Installer-Software entweder mit Kosten verbunden wären oder nur erhöhten Aufwand dargestellt hätten.

Zur Übermittlung der Releases wurde Dropbox verwendet, da dies jeweils den aktuellen Stand bei allen Beteiligten synchron halten konnte und gleichzeitig ein Informationssystem enthält, um die beteiligten Personen über Aktualisierungen zu informieren.

7.3.3 Setup der Entwicklungsumgebung

Die Installationen der im vorhergehenden Kapitel beschriebenen Software verlief erwartungsgemäß einfach - zum Einen, da die Software bereits einem breiten Publikum zugänglich ist und kommerziell vielfach eingesetzt wird, zum Anderen, da der Entwickler bereits mit allen dieser Softwarepakete gearbeitet hat und dahingehend auf einen gewissen Erfahrungsschatz zurückgreifen konnte.

7.3.4 Recherche der notwendigen Algorithmen

Wie bereits im Kapitel 7.3.1 erwähnt, basiert das Produkt auf der bestehenden Doktorarbeit 'EDV-gestützte Verfahrensbilanzierung mit Bilanzausgleich als Bewertungbasis von Abfallvermeidung in industriellen Prozessen des oben erwähnten Autors 'Florian von Linde'.

Die Darstellung des verwendeten Algorithmus gibt in dieser Arbeit zwar Aufschluss über die verwendeten Gedankengänge - war jedoch nicht ausreichend (vor allem im Bereich der überbestimmen Gleichungssysteme bei gleichzeitiger Verwendung exakt definierter Constraints), um eine exakte Neuprogrammierung des Algorithmus durchführen zu können.

Es mussten daher diverse Recherchen zur Lösung durchgeführt werden. Diese bezogen sich weitestgehend auf Internet-Recherchen von Gleichungssystemen, unter Bedachtnahme auf die Minimierung des quadratischen Fehlers bei über- und fehl-bestimmten Gleichungssystemen.

Als Folge dieser Recherchen entstand eine Programm-Bibliothek (CAPAD Library) mit entsprechenden Testfällen gemäß der Test-getriebenen Softwareentwicklung.

7.3.5 Datenbasiertes Modul zur Lösung der Algorithmen (CAPAD Library)

Die CAPAD Library besteht aus drei Modulen, die zur Umsetzung des gewünschten Produkts sinnvoll erschienen:

- EquationSolver ein String-basiertes Modul zur Lösung von Linearen Gleichungen
- Matrix zur Durchführung benötigter Matrix-Operationen
- CAPAD zur Lösung der strukturellen Zusammenhänge von Knoten, Strömen und Reaktionen, wie sie im finalen Produkt Anwendung finden

Die Module konnten mit einer Testabdeckung (Code-Coverage) von 100% erstellt werden, wobei im Zuge der grafischen Anbindung und den damit verbunden Refactorings diese auf ca. 91% sank. Dabei wurden durch den eingeschränkten Programmfluss teilweise Codestellen nicht mehr erreicht - jedoch nicht aus dem Quellcode entfernt, da es sich hierbei um Ausnahmebehandlungen handelte, die durchaus bei falschem Programmfluss wieder Einfluss nehmen könnten.

Ähnliches gilt für die schließlich verwendete und im Quellcode vorhandene Programmbibliothek 'Mehroz', welche Funktionalitäten unterstützt, die im finalen Produkt keine Anwendung fanden und daher ungetestet blieben.

7.3.6 Grafische Anbindung

Im Unterschied zu Programmbibliotheken ist das Testen der grafischen Funktionalität bedeutend schwieriger, da sich das Unit-Test-Framework nicht direkt gegen das finale Produkt linken lässt. Dementsprechend wurden diverse User-Interface-Test Frameworks getestet, die jedoch alle (inklusive dem Microsoft eigenen UI-Test-Tool) nicht verlässlich Funktionen innerhalb von Microsoft Visio 2010 ausführen ließen.

Daher wurde beschlossen ein eigenen Framework zur Ausführung der im Programmcode aufgerufenen Funktionen zu gestalten und diese mittels Makros und vorgefertigten Test-Files zu verifizieren. Abbildung 7.3 zeigt das Ergebnis von Testfällen durch dieses eigene Framework.

Die Testfälle wurden direkt in Zusammenarbeit mit dem Kunden ausgearbeitet und sowohl vom Kunden als auch vom Entwickler erweitert, um eine möglichst hohe Testabdeckung zu erreichen. Leider war durch das eigene Framework die Ermittlung der Testabdeckung in Prozent nicht möglich, sodass dieser Wert hier nicht bestimmt werden konnte.

Die Oberfläche selbst wurde durch die Erweiterung 'Office Developer Tools for Visual Studio 2012' durchgeführt und im User-Interface-Designer von Visual Studio 2012 nach dem Prinzip WYSIWYG (What you see is what you get) erstellt. Es wurde versucht die Programm-Anweisungen als Reaktionen auf die zugehörigen GUI-Funktionen so direkt als möglich an das CAPAD Library Modul weiter zu leiten. In den meisten Fällen gelang dieses Vorhaben auch gut. Ausnahme bildeten hier die dynamischen Felder in der grafischen Anbindung, die auch diversen Quellcode für diese Vorgänge beinhalten.

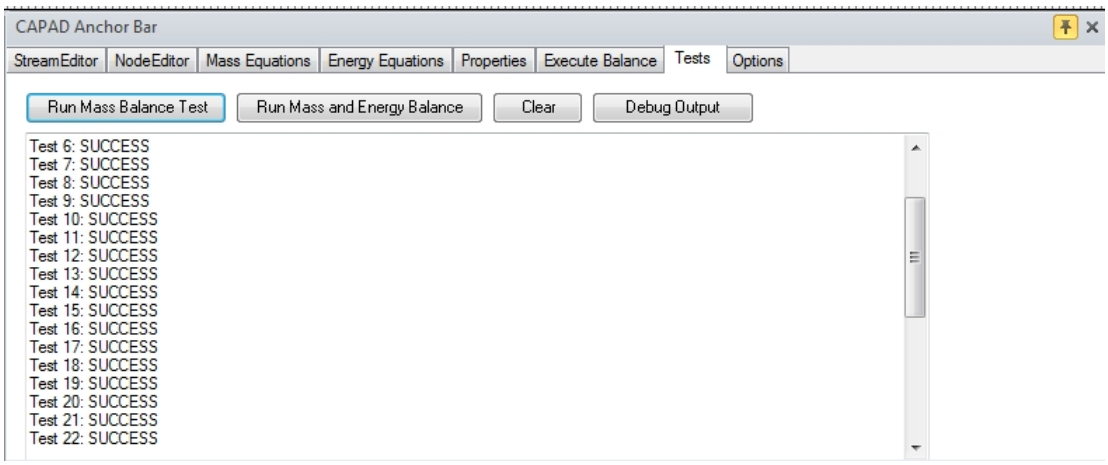

Abbildung 7.3: VisioCAPAD Tests (Screenshot)

7.3.7 Usability Anpassungen

Nachdem eine erste Version mit vollständiger Funktionalität verfügbar war, wurde das Produkt auf mehreren Rechnern manuell installiert um das Produkt manuell von mehreren Personen testen lassen zu können. Die Benutzer wurden daher gebeten sowohl allgemeine Software-Fehler zu melden, als auch die Benutzbarkeit selbst zu beurteilen und im Issue Tracker 'Asana' fest zu halten.

Um den praktischen Nutzen der Software evaluieren zu können wurden 'echte' Problemfälle aus der Industrie herangezogen. Hierbei zeigten sich neben kleinen Usability-Problemen auch fehlende Funktionalität zur Behebung von Problemfällen - obwohl diese nicht als Fehler des Programms selbst sondern als Funktionalitätserweiterung betrachtet wurden.

7.3.8 Nicht bedachte Problemfälle

Es stellte sich heraus, dass die Software für das Anwendungsgebiet in der Industrie eine entscheidende Programmfunktionalität vermisste - eine Stoffdatenbank bzw. die Umrechnung zwischen Energie und Masse, die nicht zwangsläufig linear erfolgt und daher nicht einfach durch einen Algorithmus für lineare Gleichungen zu lösen ist.

In Zusammenarbeit mit der Technischen Universität Graz wurde ein Algorithmus entwickelt, der in Iterationen auch dieses nicht lineare Gleichungssystem lösen konnte und User-Interface und Bibliotheken-Module wurde für eine Stoffdatenbank erweitert.

Hier spielte die Test-getriebene Softwareentwicklung ihre Vorteile aus und schaffte vor allem für strukturelle Veränderungen in großen Ausmaßen Sicherheit, damit die bestehende Funktionalität des Programms durch diese Änderungen keine Einbußen erfährt.

7.3.9 Installer-Setup

Die Installer Recherche wurde auf den Abschluss des Projekts verlegt. Es standen unterschiedlichste kommerzielle und nicht-kommerzielle Installer zur Debatte. Vorerst wurde auf das kommerziell verfügbare Produkt 'InstallShield' in einer Trial-Version zurückgegriffen. Aufgrund einer Budget-Entscheidung musste dann jedoch darauf verzichtet werden. In Folge entstand der Wunsch nach frei verfügbarer Installationssoftware. Hierfür bot sich dann der Microsoft ClickOnce Installer, der direkt in Visual Studio 2012 integriert ist, an.

Leider war diese Vorgangsweise nur schwer automatisiert testbar, sodass eine Reihe manueller Tests auf Rechnern mit unterschiedlichen Betriebssystemen und Rechte-Einstellungen durchgeführt wurden - all das entpuppte sich als ein langwieriger Prozess.

7.3.10 Finales Release

Schließlich konnten alle Bedürfnisse durch Einstellungen im Installer-Setup befriedigt werden und die Software im März 2013 in den produktiven Einsatz gehen.

Abbildung 7.4 zeigt das fertige Produkt im Einsatz.

7.4 Probleme und Erfolge bei der Umsetzung

Das folgende Kapitel spiegelt die subjektiven Eindrücke des Entwicklers bei der Umsetzung des Projekts VisioCAPAD wider und soll einen Überblick über die empfundenen Vor- und Nachteile von Test-getriebener Softwareentwicklung vermitteln.

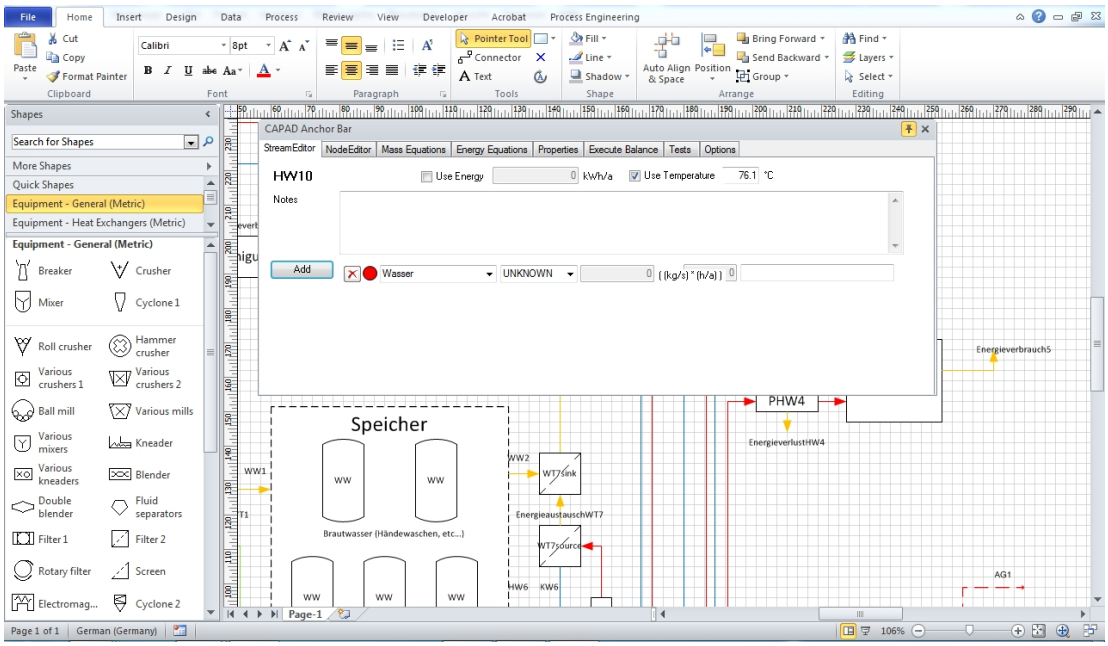

Abbildung 7.4: VisioCAPAD (Screenshot) im produktiven Einsatz

7.4.1 Positive Eindrücke

Da die Entwicklung von VisioCAPAD nicht durchgehend erfolgte, sondern mit zwischenzeitlich langen Pausen, in denen der Entwickler mit anderen Projekten beschäftigt war, zeigte sich vor allem eine unerwartet große Sicherheit beim erneuten Einarbeiten in das Projekt. Dies lässt darauf schließen, dass Test-getriebene Softwareentwicklung in Unternehmen die Einarbeitung von neuen Mitarbeitern erleichtern könnte, aber auch dass diese Methode nicht nur für große Teams, sondern auch für einzelne Entwickler sinnvoll ist. Subjektiv konnte hier sogar eine Produktivitätsverbesserung vernommen werden, da andernfalls die Veränderungen einzelnen manuellen Tests unterzogen werden müssten.

Es konnten ebenfalls Release-Versionen des Programms in kürzeren Abständen erzeugt werden, da das Produkt vor der Übertragung zum Kunden nicht nochmals einem intensiven Test unterzogen wurde. Hier war die Sicherheit beim Entwickler so groß, dass ein weiterer manueller ganzheitlicher Systemtest nicht wichtig erschien - was sich über die Projektdauer hinweg auch als richtig erwies.

7.4.2 Negative Eindrücke

Vor allem in den Anfangsphasen der Test-getriebenen Softwareentwicklung zeigten sich große Produktivitätseinbußen. Selbst Aufgaben, die nach dem 'alten' Ansatz schnell erledigt gewesen wären, forderten ein Umdenken im Workflow.

Dieser neue Workflow zog nicht nur die Erstellung von Testfällen selbst nach sich - sondern auch eine neue Vorstellung im Design des Quellcodes selbst. Diese Phase hielt ca. zwei bis drei Wochen an, bis sich eine Normalität des Denkens für diese neue Methode einstellte. Innerhalb dieser Einarbeitungszeit kann durchaus von einer mehr als verdoppelten Entwicklungszeit gesprochen werden. Danach entsprach der Produktivitätsverlust subjektiv eher 10%, der wahrscheinlich durch die häufigeren Releases und daher agileren Softwareentwicklung relativiert werden kann.

Zur Mitte des Projekts hin trat ein neues Phänomen auf, das in dieser Form ohne Test-getriebene Entwicklung nicht bestand: Die Verwirrung, ob ein Testfall bereits existiert oder noch nicht.

Auch mittels Code-Coverage (die in solchen Fällen vor allem in Visual Studio die nicht-getesteten Code-Stellen sehr gut anzeigt) konnte nicht ermittelt werden, ob ein Grenzfall als Testfall bereits implementiert ist oder nicht.

Hier zeigte es sich, vor allem nach längeren Pausen, dass Testfälle doppelt geschrieben wurden.

8 Fazit

Das Fazit soll eine synoptische Darstellung aus wissenschaftlicher Arbeit, verfügbarer Literatur, sowie einer Einschätzung der Verwendbarkeit von Test-getriebener Softwareentwicklung bieten.

8.1 Evaluierung von Test-getriebener Softwareentwicklung

Vor allem die Einarbeitung in die Test-getriebene Softwareentwicklung machen den produktiven Einsatz in Unternehmen zu einem schwierigen Unterfangen. Die Anwendung ist dem subjektiven Empfinden nach grundsätzlich kontraintuitiv - ähnlich der Pairprogramming-Methode.

Die wissenschaftlichen Studien über die Effizienz divergieren sehr und fanden hauptsächlich in kleinen Fallstudien über kurze Entwicklungszeiträume statt, eine Problematik, die den Einsatz im Unternehmen zur Feldforschung erklärt. Im Verständnis von Projektmanagement kann Test-getriebene Softwareentwicklung auch als Weitergabe von Wissen verstanden werden. In diesem Fall würden die Vorteile erst dann richtig ausgespielt werden, wenn unterschiedliche Entwickler an der selben Code-Basis arbeiten oder der Entwickler über mehrere Projekte hinweg am selben Code arbeitet.

Test-getriebene Entwicklung scheint vor allem in Gebieten schwierig, in denen es nur unscharfe Formulierungen der Aufgabenstellungen gibt. Das Gegenstück wird

durch Wegwerf-Prototypen repräsentiert, die in unterschiedlichen Geschäftsfeldern durchaus ihre Berechtigung haben. Hier scheint es sogar, dass Test-getriebene Softwareentwicklung niemals Anwendung finden wird. Als problematisch stellt sich der Fall dar, wenn aus einem geplanten Wegwerf-Prototyp das finale Produkt ohne Neuentwicklung entstehen soll. In diesem Fall liegen keinerlei Testfälle vor.

Das Einsatzgebiet scheint auch sehr stark von den absoluten Qualitätsansprüchen der Softwareentwicklung abzuhängen. Hier sei das Beispiel 'Proof-of-concept' erwähnt, bei dem ja nur die grundsätzliche Analyse der Umsetzbarkeit durchgeführt wird.

Auf der anderen Seite scheint es auch Paradebeispiele für Test-getriebene Softwareentwicklung zu geben, nämlich genau dort, wo zu keiner Zeit Fehler im System auftreten dürfen. Hier sei die laufende Entwicklung an Software im produktiven Einsatz im Internet, die Softwareentwicklung in Bereichen, in denen Leben auf dem Spiel steht (zB. Raumfahrt oder Personentransport) oder Fehler mit enormen Verlusten gekoppelt sein können (zB. Software im Bankenwesen oder für Glücksspielautomaten). Hier profitieren vor allem die Unternehmen von der besseren Abschätzbarkeit der Kosten für Qualitätssicherung.

8.2 Umstieg auf Test-getriebene Softwareentwicklung

Damit Unternehmen einen Einstieg in die Test-getriebene Softwareentwicklung wagen können bzw. sollten, bedarf es mehrerer Schritte:

- Überzeugung oder Experimentierfreude des Management
- Aufklärung und Informationen für die Entwickler
- Schrittweiser Integration von Test-getriebener Entwicklung

Entwickler brauchen sowohl die notwendigen Informationen als auch technischen Hilfsmittel, um Test-getriebene Softwareentwicklung angemessen durchführen zu können. Ohne geeignete Zustimmung und teilweise auch Unterstützung ist das ein schwieriges Unterfangen - sodass bei Eigenengagement eines oder mehrerer Entwickler eine gewisse Überzeugungsarbeit beim Management im Vorfeld erfolgen sollte.

Der Argumentation dienlich könnte die Aussicht auf höhere Qualität und den damit verbundenen geringeren Folgekosten eines Projekts sein. Weiters bietet Test-getriebene Entwicklung dem Management die Möglichkeit auf bessere bzw. interaktivere Kundenzusammenarbeit, ein Vorgehen, das sich nicht nur hinsichtlich der Produktion selbst, sondern auch dem Miteinander von Auftraggeber und Auftragsnehmer gegenüber als angenehm erweisen kann. Durch Test-getriebene Softwareentwicklung gibt es weiters die Möglichkeit neue Zahlungsmodalitäten mit dem Kunden zu entwickeln.

Während traditionelle Vorgangsweisen auf Werkverträgen, mit Pflichtenheften und Milestonezahlungen beruhen, besteht durch die Etablierung von Test-getriebener Softwareentwicklung die Möglichkeit als echter Dienstleistungsbetrieb zu fungieren und dem Kunden den laufenden Fortschritt der Entwicklung zu präsentieren.

Kommt die Initiative zum Umstieg auf eine Test-getriebene Entwicklung vom Management selbst, so ist es wichtig den Entwicklern im Vorfeld ausreichend Informationen über die Vorteile und Umsetzungsarten zukommen zu lassen. Hierfür bietet die Literatur eine umfangreiche Anzahl von Büchern und Fachartikel. In größeren Unternehmen wird auch die Inanspruchnahme von Seminaren und Workshops eine nicht zu unterschätzende Rolle spielen.

In der Praxis wird ein sauberer Schnitt zur Test-getriebenen Softwareentwicklung kaum möglich sein. Durch laufende Projekte und der in Unternehmen immer fehlenden Zeit sollte daher ein schrittweiser Übergang erfolgen.

Dabei stellt sich auch die Frage nach der aktuellen Situation des Unternehmens. Viele - insbesondere kleinere Softwareentwicklungsunternehmen - testen ihre Soft-

ware noch immer händisch und in unregelmäßigen Abständen. Hier lohnt sich eventuell die erste Integration von internen Akzeptanztests. Dadurch wird eine Bewusstseinsbildung erreicht, die den notwendigen Einsatz von Tests in früheren Entwicklungsphasen erst veranschaulicht.

Als nächster Schritt folgt natürlich auch gleich die Automatisierung dieser Tests - die Entwickler selbst kommen daher zu einem schnelleren Feedback über die von ihnen geleistete Arbeit. Da die häufig bereits bestehende Software noch nicht auf die Durchführung von Modultests ausgelegt ist, gilt es nun relativ rasch (um die Entwickler nicht durch häufig fehlschlagende Funktionstests zu demotivieren) eine Arbeitsumgebung für die Programmierer zu schaffen, die es ihnen erlaubt selbstständig Tests ihrer Implementationen durchzuführen. Dies sollte anfangs keineswegs obligatorisch erfolgen, jedoch den Entwicklern die Möglichkeit geben Tests in einer definierten Umgebung einfügen zu können.

Durch das langsame aber (hoffentlich) stetige Wachstum der Testfälle bekommen auch Entwickler, die selbst keine Testfälle geschrieben haben, eine gewisse Sicherheit in der Umsetzung solcher Testfälle bzw. ihrer Implementationen und beginnen daher selbst neue Tests einzufügen. Hier ist die Basis geschaffen, dass auch eine Test-getriebener Softwareentwicklungsansatz verfolgt werden kann.

Mit der wachsenden Anzahl von Testfällen wird mit der Zeit ein gewisses Testmanagement notwendig werden. Hier ist wieder die Managementebene gefragt um den Entwicklern die notwendigen Kompetenzen zu geben und die Strukturierung der Testfälle entweder selbst oder durch einzelne qualifizierte Personen durchführen zu lassen.

8.3 Zukünftige Entwicklung

Aus unternehmerischer Sicht werden Softwareentwickler mehr Forschung auf diesem Gebiet fordern, bevor der Einsatz eine breite Fan-Gemeinde bilden wird. Es fehlen wissenschaftliche Arbeiten über die Haupteinsatzgebiete von Test-getriebener

Softwareentwicklung, sowie Analysen zum Thema 'Return on Invest', d.h., ab welchen Projektgrößen und bei welchem Kapitaleinsatz sich der Umstieg auf diese Methode der Softwareentwicklung auszahlt.

Es besteht die Hoffnung, dass Großkonzerne wie Microsoft oder IBM aufgrund der sehr hohen Kosten im Bereich Softwareentwicklung hier ebenfalls ein starkes Augenmerk auf Langzeitstudien legen, wobei sich wahrscheinlich auch hier die Ergebnisse nur auf Unternehmen in deren Größe umlegen lassen wird. Generell wäre ein Design für eine aussagekräftige Studie zur Evaluierung von Test-getriebener Softwareentwicklung wünschenswert.

Auf universitärer Ebene bieten sich Analysen mit großer Teilnehmerzahl (vielleicht mehr als Hundert pro Lehrveranstaltung) an, wobei hier auch eine nicht Team-basierte jedoch nach Erfahrung klassifizierte Analyse wünschenswert wäre.

In diesem Fall könnte ebenfalls eine Bewusstseinsbildung über die Vor- und Nachteile von Test-getriebener Softwareentwicklung erreicht - bzw. die Erfahrungen durch soziale, psychologische und anthropologische Studien wissenschaftlich verarbeitet werden - da auch Mitarbeitermotivation und deren Verständnis für Unternehmen einen erheblich großen Wert haben.

Literaturverzeichnis

Beck K., 2002: *Test Driven Development: By Example*, Addison-Wesley Professional.

Beck K., 2004: *Extreme Programming Explained*, Addison-Wesley.

Bhat T., Nagappan N., 2006: *Evaluating the efficacy of test-driven development: industrial case studies*, in: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ACM, S. 356–363.

Canfora G., Cimitile A., Garcia F., Piattini M., Visaggio C.A., 2006: *Evaluating advantages of test driven development: a controlled experiment with professionals*, in: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ACM, S. 364–371.

George B., Williams L., 2003: *A structured experiment of test-driven development*, in: *Elsevier*, Inf Softw Techn (IST), S. 337–342.

Janzen D., Saiedian H., 2005: *Test-driven development concepts, taxonomy, and future direction*, in: *Computer*, 38(9), S. 43–50.

Kaufmann R., Janzen D., 2003: *Implications of test-driven development: a pilot study*, in: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, S. 298–299.

Martin R.C., 2008: *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall International.

Martin R.C., 2011: *Clean Coder*, Addison-Wesley.

Maximilien E.M., Williams L., 2003: *Assessing test-driven development at IBM*, in: *Software Engineering, 2003. Proceedings. 25th International Conference on*, IEEE, S. 564–569.

Nachiappan Nagappan T.B. E. Michael Maximilien, Williams L., 2008: *Realizing*

quality improvements through test driven development: results and experiences of four industrial teams, in: *Empir Software Eng*, Springer Science + Business Media, S. 289–302.

Patrick Hofmann F.H., 2013: *Manual für Bilanzierungstool VisioCAPAD Solution 1.0.1.42*, nicht publiziert; Manual liegt dem Produkt bei.

Sanchez J., Williams L., Maximilien E., 2007: *On the Sustained Use of a Test-Driven Development Practice at IBM*, in: *Agile Conference (AGILE), 2007*, S. 5–14.

Schneider K., 2012: *Abenteuer Softwarequalität*, dpunkt.verlag.

Williams L., Maximilien E.M., Vouk M., 2003: *Test-driven development as a defect-reduction practice*, in: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, IEEE, S. 34–45.