Gerhard Scheikl

Extreme Programming bei Einzelentwicklung

MASTERARBEIT

zur Erlangung des akademischen Grades eines

Diplom-Ingenieurs

Masterstudiumstudium Telematik

Technische Universität Graz

Betreuer:

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

Institut für Softwaretechnologie

Graz, im Mai 2012

Eidesstattliche Erklärung

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

Graz, am ....

(Unterschrift)

Kurzfassung

Die Konzepte von Extreme Programming wurden an die Einzelentwicklung angepasst. Es wurde eine Software zur einfachen, GUI-Unterstützen Rücksicherung einzelner Partitionen erstellt. Dies geschah unter Einhaltung der erarbeiteten Konzepte. Die Entwicklung wurde in Iterationen mit einer Zeitspanne von je 2 Wochen aufgeteilt.

Die User-Stories wurden zu Beginn in der Planungsphase erarbeitet und ihre Komplexität wurde nach einem Punktesystem bewertet. Zu Beginn jeder Iteration wurden entsprechend viele Tasks abgeschätzt, um genügend Arbeitsaufwand bis zur nächsten Iteration abzudecken.

Nach Abschluss des Projektes wurde der Verlauf analysiert und die Ergebnisse wurden aufbereitet und ausgewertet.

Stichwörter

Extreme Programming, Softwareentwicklung, Einzelentwicklung

Abstract

The concepts of Extreme Programming have been adjusted to single development. A software for simple, GUI supported image restoring has been created. Single partitions can be chosen for recovery. The development happened under adherence to the previously developed concepts. Development has been divided into several iterations, each with a timespan of two weeks.

At the start of the planning phase, user stories have been developed. Their complexity has been assessed by a point system. The needed tasks have been estimated at the begin of each iteration. (just the right amount, so that there was enough work for every iteration)

The progression has been analyzed at the end of the project. The findings have been processed and analyzed.

Keywords

Extreme Programming, software development, single development

Inhaltsverzeichnis

1 Einleitung

Heutzutage werden oft kleinere Anwendungen von nur einer Person entwickelt. Dazu beigetragen hat unter anderem der Smartphone-Boom. Durch den Trend hin zu kleinen Apps auf Handys, Tablets usw. gibt es aktuell viele Einzelentwickler.

Viele davon starten ihr Projekt nach einem kurzen Abstecken der Ziele direkt mit der Programmierung. Dies hat natürlich Vor- aber auch Nachteile. Man ist zwar relativ schnell auf einem verwendbaren Software-Stand, jedoch auf Kosten der Test- und Wartbarkeit.

Diese Arbeit befasst sich mit der Umsetzung von Extreme-Programming-Konzepten auf Einzelentwicklung. Es werden bewährte Herangehensweisen angepasst bzw. übernommen.

Durch den Einsatz von Test-Driven-Development(TDD) und die Anwendung agiler Prozesse erwartet man sich eine Verbesserung der Software-Qualität, sowie eine Verbesserung bei der Weiterentwicklung. Natürlich ist hierfür Zusätzlicher Aufwand von Nöten.

Die Vorgehensweise wird anhand eines konkreten Projektes gezeigt. Es handelt sich um eine Software, mit deren Hilfe der Benutzer auf einfache Weise Festplatten-Images erstellen und wieder einspielen kann. Zusätzlich bietet das Programm die Möglichkeit, Software-Pakete direkt nach der Wiederherstellung automatisiert einzuspielen.

Es handelt sich hierbei um einen Rewrite und Verbesserung einer bereits be-

stehenden Software. Da die alte Implementierung Design-Schwächen aufweist und nur noch schwer erweiterbar ist, wurde beschlossen, die Anwendung von Grund auf neu zu entwickeln. Zusätzlich gilt es noch, neue Anforderungen einzubauen. Die neue Version soll wesentlich mehr Features aufweisen, als die alte.

2 Angewendete Konzepte

Hier eine kurze Übersicht über die wichtigsten Konzepte:

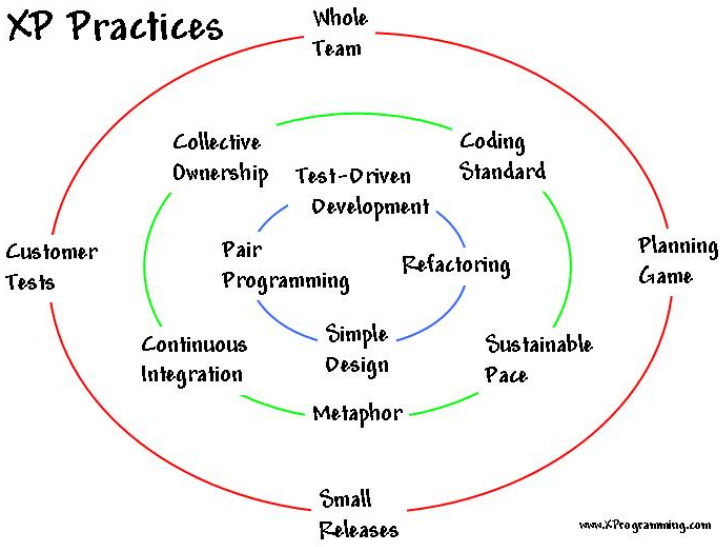

Abbildung 1: XP practices [Jef11]

Weitere Informationen wurden aus [BA04] und [Wel09] bezogen. Folgende Konzepte wurden hierbei übernommen bzw. angepasst:

2.1 Organisatorische Praktiken

Energized Work

"Arbeite nur so lange, wie du produktiv bist." Man soll nicht so lange, wie möglich arbeiten. Es wird hierbei auch eine Maximalarbeitszeit festgelegt. Die 40-Stunden-Woche darf nicht überschritten werden. Wenn man weiter arbeitet, ohne wirklich produktiv zu sein, kann dies durchaus eine negative Auswirkung auf die Code-Qualität haben. Wenn man zu lange arbeitet, ent-

steht oft Software, die am folgenden Tag wieder korrigiert werden muss, weil schlicht und einfach Fehler übersehen wurden. Dies kann dann somit zu mehr Arbeitsaufwand führen, als ohne Zusatzstunden nötig gewesen wäre.

Weekly Cycle

Plane immer nur für eine Woche.

Am Anfang der Woche werden folgende Punkte betrachtet:

- Review des Fortschritts
- Auswahl der User-Stories für die aktuelle Woche
- Aufteilung der User-Stories in Tasks

Dieser Punkt findet sich auch im Agilen Manifest wieder:

"Liefere funktionierende Software regelmäßig innerhalb weniger Wochen oder Monate und bevorzuge dabei die kürzere Zeitspanne." [BBvB+01]

Kürzere Zeitspannen haben sich bewährt. Man kann dadurch viel schneller auf Veränderungen reagieren.

Slack

Füge bei der Planung auch weniger wichtigere Tasks hinzu. Diese können fallen gelassen werden, wenn ein Zeitliches Problem auftritt. Dies ist ein sehr einfaches, aber wesentliches Konzept. Abschätzungen in der Softwareentwicklung erfordern sehr viel Erfahrung und sind oft relativ ungenau. Falls mehr

Zeit benötigt wird, als geplant, so können die zusätzlich abgeschätzten Tasks einfach ignoriert werden.

Continuous Integration

Änderungen werden spätestens nach ein paar Stunden getestet und integriert. Das Frühzeitige Einpflegen der Modifikationen trägt dazu bei, Fehler schneller zu erkennen und die Code-Qualität zu verbessern. Oft werden mit jedem Commit oder in bestimmten Zeitintervallen automatisch Tools gestartet, die die Software neu kompilieren und alle verfügbaren Tests starten. Dadurch erhält man zeitnah Informationen über die Auswirkung der aktuellen Code-Modifikationen.

Test-Driven Development (TDD)

Bevor irgendein Code geschrieben wird, wird entsprechender automatisierter Test-Code geschrieben. Dieser muss bei Ausführung fehlschlagen. Durch diesen Ansatz werden gleich mehrere Probleme vermieden:

- "Scope creep": unkontrollierte Änderungen
- "Coupling and cohesion": Wenn es schwer fällt, einen entsprechenden Test-Case zu schreiben weist das auf ein Design-Problem hin
- "Trust": Es ist schwierig einem Programmierer zu vertrauen, dessen Code nicht funktioniert. Durch die automatisierten Tests kann mehr Vertrauen für einzelne Codeteile gewonnen werden.

- "Rythm": Durch Test-First gibt es eine klarere Vorgabe, was als nächstes zu erledigen ist. Entweder ist ein Test-Case zu schreiben, oder für einen fehlschlagenden Code zu erzeugen.

Incremental Design

Investiere jeden Tag etwas Zeit für das Design. Versuche das Design perfekt für die Aufgaben des jeweiligen Tages zu machen.

Single Code Base

Es gibt nur eine Code-Basis. Man darf in einem temporären Branch entwickeln, allerdings nicht länger, als ein paar Stunden.

Project Velocity

Die Project-Velocity ist ein Wert dafür, wie viel Arbeit im Projekt erledigt wird. Er spiegelt die Produktivität wieder. Hierfür werden die Abschätzungen der in der Iteration abgeschlossen Tasks addiert. Dieser Wert sollte im Idealfall über den gesamten Projektverlauf nur sehr geringen Schwankungen unterliegen.

Die Velocity ist ein wichtiger Anhaltspunkt, der Aussagen über den Projektverlauf zu lässt. Sie wird bei den Planungen herangezogen um eine passende Auswahl an User-Stories für die nächste Iteration zu finden.

Fix XP when it breaks

Zu Beginn sollte man sich noch an alle Regeln des XP halten. Währen des Verlaufes kann sich jedoch herausstellen, dass einige davon nicht praktikabel sind. Ist das der Fall, so soll man nicht zögern und diese fallen lassen, bzw. durch andere, geeignetere Praktiken ersetzen.

2.2 Entwicklungsmethoden

2.2.1 Design

KISS-Prinzip

"Keep it simple, stupid!" [Wik11b] Es soll eine möglichst einfache Lösung für das Problem gefunden werden um die Wartbarkeit und die Wiederverwendbarkeit zu verbessern.

Dies ist ein wirklich wichtiges Grundprinzip, welches man nie aus den Augen verlieren sollte. Oft entstehen in Projekten Konstrukte, die nach längerer Zeit nicht einmal mehr vom Urheber selbst gewartet werden können. Dies ist tunlichst zu vermeiden.

CRC-Cards

CRC steht für "Class, Responsibilities, Collaborators" und spiegelt den Inhalt der Karten wieder. Hiermit wird folgendes definiert:

- Klassenname

- die "Verantwortung", also welche Aufgaben von der Klasse erfüllt werden sollen
- mit welchen anderen Klassen interagiert wird

[BC89]
Hier ein leeres Template:

Class	
Responsibilities	Collaborators

Tabelle 1: CRC-Vorlage

Spike-Solutions

Das Erstellen von Spike-Solutions dient dazu, um Probleme schneller zu erkennen. Hierbei werden kleine und einfache Programmteile entwickelt, die eine mögliche Lösung (oder mehrere) für das selbe Problem darstellen. Dies wird verwendet, um z.B. technische Probleme vorab bewerten zu können. Oft werden Spike-Solutions nicht in den Produktiv-Code eingepflegt, sondern verworfen. Dementsprechend muss dabei auch nicht auf Code-Qualität geachtet werden.

Zusatzfunktionen

Zusatzfunktionen sollen nicht zu früh berücksichtigt werden. Dadurch zu rasches einbringen von nicht unmittelbar benötigten Features wird der Projektfortschritt unnötig gehemmt. Wird dafür zu viel Zeit aufgewendet, so fehlt diese oft für die wesentlicheren Komponenten der Software.

Refactoring

Refactoring ist ein wichtiger Teil in der Softwareentwicklung. Es hilft, die Komplexität zu verringern und Redundanz zu eliminieren. Es sollte so oft, wie möglich durchgeführt werden, um die Codebasis ständig zu verbessern. Sobald Fehler z.B. im Design erkannt werden muss eingegriffen werden. Refactoring ist somit eine immer wiederkehrende Aufgabe, die maßgeblich zur Erfüllung des Projektziels ist.

2.2.2 Implementierung

Coding-Standard

Die Code-Formatierung soll nach bewährten Standards durchgeführt werden. Dadurch wird der Code konsistent und lesbar gehalten.

TDD

TDD ist sowohl eine organisatorische, als auch eine Programmiertechnische Herangehensweise. Wie bereits erwähnt, darf keine Funktionalität implementiert werden, so lange nicht mindestens ein fehlschlagender Unit-Test existiert.

Häufiges Commiten

Laufendes Commiten ist sehr wichtig, wenn in einem Team gearbeitet wird. Nur so kann verhindert werden, dass die Codebasen der einzelnen Entwickler zu weit auseinander driften. Auch für die Einzelentwicklung ist dies von Bedeutung um Änderungen auch zu späteren Zeitpunkten im Projekt leichter nachvollziehen zu können, bzw. um kleinere Änderungen auch einfach wieder rückgängig zu machen.

2.2.3 Testen

Testabdeckung

Der Code muss zu 100% mittels Unit-Tests abgedeckt sein. Vor dem Release müssen alle Unit-Tests positiv abgeschlossen werden. Beim Finden eines Fehlers muss ein passender Unit-Test angelegt werden. Für Tests, die nicht automatisch ausgeführt werden können, müssen Testlisten erstellt werden. Diese sind entsprechend den Anforderungen abzuarbeiten.

Acceptance

Es gibt ein simples Kriterium, wann eine Implementierung akzeptiert wird: Für jede User-Story müssen die jeweiligen Acceptance-Tests erzeugt bzw. aus den vorhandenen ausgewählt werden.

2.3 Zusammenfassung

Das Projekt wurde mit Einhaltung der vorgestellten Konzepten durchgeführt. Es wurden jene Aspekte von XP ausgewählt, die auch für Einzelentwicklung

praktikabel erschienen.

Es wurde versucht die Auswahl möglichst gering zu halten um den Organisatorischen Aufwand so gering, wie möglich zu halten. Auf die Adaption weiterer Konzepte wurde bewusst verzichtet.

Konzept	Ausschlussgrund
Stand-up-Meeting	Als Einzelperson nicht durchführbar
Pair programming	Als Einzelperson nicht durchführbar
Move people around	Als Einzelperson nicht durchführbar
Dedicated integration computer	unnötiger Ressourcenaufwand
System Metaphor	ist oft schwierig eine gute zu finden, mehr im Team von Bedeutung

Tabelle 2: Nicht umgesetzte Konzepte

Hier der grundlegende Ablauf eines XP-Projekts:

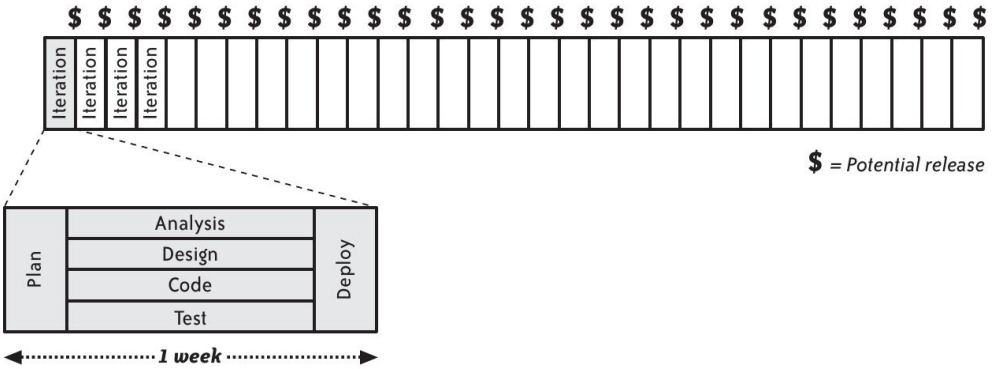

Abbildung 2: XP lifecycle [SW08]

Zu Beginn einer Jeden Iteration steht eine kurze Planungsphase. Danach

wechseln Analyse, Design, Programmierung und Testen einander ab, bis schließlich gegen Ende das Deployment durchgeführt wird. Abschließend sollte nach jeder Iteration ein möglicher Release-Kandidat zur Verfügung stehen.

3 Andere Konzepte

In diesem Kapitel werden weitere agile Ansätze für die Einzelentwicklung vorgestellt.

3.1 Personal Kanban

Hierbei handelt es sich um eine Adaption von Kanban für die Einzelentwicklung.

"Kanban in der IT ist ein Vorgehen, das bei der Softwareentwicklung die Anzahl paralleler Arbeiten, den Work in progress (WiP), reduziert und somit schnellere Durchlaufzeiten erreicht und Probleme – insbesondere Engpässe – schnell sichtbar macht."[Wik11a]

Bei dieser Methode geht es darum, den Fluss der Arbeit aufrecht zu erhalten und zu visualisieren. Eine wichtige Eigenschaft ist auch die Begrenzung der anfallenden Arbeit (WiP). Kanban ist ein Werkzeug um die Arbeit zu visualisieren und zu organisieren. Es gilt das Bestreben, den Arbeitsprozess transparent zu machen, um zu jedem Zeitpunkt zu wissen, wie der aktuelle Stand ist.

In seinem Buch [BB11] bzw. auf einer dazu passenden Homepage [Ben11] stellt Jim Benson die Methode des "Personal Kanban" vor. Es erlaubt die Visualisierung der ausstehenden Arbeit bzw. eine Visualisierung der Art und Weise, in der die Arbeit durchgeführt wird. Personal Kanban soll eher als Pattern und nicht als exakt definierter Entwicklungsprozess gesehen werden. Es liegt an den Anwendern, die Vorgehensweise an die eigenen Gegebenheiten

anzupassen. Man kann dies als eine Parallele zu “fix XP if it breaks” sehen. Personal Kanban skaliert gut und ist somit nicht nur für Einzelentwickler, sondern auch für kleinere Gruppen gedacht.

Es gibt lediglich zwei Grundsätze beim Personal Kanban:

1. Visualisierung der Arbeit
2. Limitierung der WiP

Die Visualisierung ist wichtig, um immer direkt vor Augen zu haben, wie der aktuelle Arbeitsstand ist. Weiters soll dadurch die Priorisierung vereinfacht werden und die Task-Übersicht erleichtert werden.

Die Limitierung der WiP ist ein sehr wichtiger Punkt. Man kann nur schwer viele Dinge gleichzeitig erledigen. Je mehr man sich zu mutet, desto mehr leidet auch die Performance darunter. Multitasking ist ein schlechter Ansatz um die anstehenden Tasks abzuarbeiten.

Als ersten Schritt beim Personal Kanban wird eine Visualisierung für den “value stream” [Wik11d] etabliert.

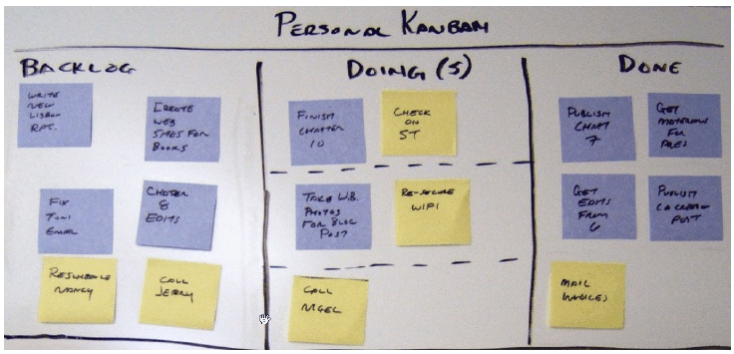

Abbildung 3: Kanban-Board [Ben11]

Hier sieht man ein Beispiel für ein Kanban-Board. Es besteht aus 3 Spalten: Backlog, Doing und Done. Dies stellt die einfachste Art und Weise dar, den value stream zu definieren. Die Tasks werden in einem simplen Workflow von links nach rechts über das Board geschoben. Damit werden drei Ziele erreicht:

1. Auflistung der WiP (Doing)
2. Auflistung der ausstehenden Arbeit (Backlog)
3. Dient als Maß für die Effizienz (Durchsatz)

Man sollte hierfür ein Whiteboard verwenden und nicht Papier. Dies hat den Grund, dass Prozessänderungen auf Papier nur schwer möglich sind. Ein Whiteboard bietet z.B. für das Hinzufügen von Schritten die nötige Flexibilität.

Als nächster Schritt wird der Backlog erstellt. Alles was aktuell an Arbeit ansteht, entspricht dem Backlog. Die ausstehende Arbeit wird in Tasks unterteilt und auf Post-Its niedergeschrieben. Ab einem gewissen Punkt stellt

man meistens fest, dass noch "viel zu viel" zu tun ist. Dies ist anfangs als normal zu betrachten.
Sollte sich heraus stellen, dass einzelne Tasks zu klein geworden sind, ist es notwendig diese zusammen zu fassen. Umgekehrt ist es auch ratsam, zu große Tasks auf mehrere kleine aufzuteilen.
Eine farbliche Trennung kann hierbei auch helfen, unterschiedliche Aufgabenbereiche einfach voneinander zu unterscheiden.

Nun geht man dazu über, die WiP zu limitieren. Man setzt ein Maximum dafür, wieviel man gleichzeitig erledigen kann. Hier die korrekte Zahl zu finden ist anfangs eher schwierig. Als guter Startwert hat sich eine Menge von 5 Tasks etabliert. Dieser Wert ist keineswegs als fixiert zu betrachten. Man kann diesen im Laufe der Entwicklung durchaus öfter anpassen.
Personal Kanban unterstützt einen dabei, die richtige Menge an WiP zu finden. Dadurch nähert man sich mit der Zeit an, die optimale Anzahl an Arbeit mit der optimalen Performance zu erledigen.

Nachdem nun alle nötigen Grundlagen für die Entwicklung gegeben sind, kann mit dieser begonnen werden. Hierbei werden die Tasks von einer Stufe zur nächsten verschoben. Zuerst vom Backlog auf Doing und nach deren Fertigstellung von Doing auf Done.
Wichtig ist hierbei, dass mit dem Verschieben vom Backlog auf Doing auch eine Priorisierung des Tasks einher geht. Wichtigere Tasks müssen natürlich Vorrang bekommen.

Personal Kanban endet nicht mit dem Verschieben der Tasks auf Done. Eine volle Done-Spalte steht natürlich für die Produktivität. Es gilt allerdings

noch festzustellen, wie effektiv gearbeitet wurde.

Hierfür sollte man sich folgende Fragen stellen: [BB11]

- Which tasks did you do particularly well?
- Which tasks made you feel good about yourself?
- Which tasks were difficult to complete?
- Were the right tasks completed at the right time?
- Did the tasks completed provide value?

Die Retrospektive ist ein wichtiges Vorgehen, um festzustellen, was gemacht wurde und wie es gemacht wurde. Weiters dient es, um aufzuzeigen, worin man gut ist und wo noch Verbesserungsbedarf besteht.

Vergleich zu XP

Im vergleich zu XP werden nur sehr wenige Dinge vorgegeben. Kanban ist für eine Anwendung in der Softwareentwicklung möglicherweise zu sehr vereinfacht. Auf der anderen Seite bietet es dadurch natürlich auch ein sehr hohes Maß an Flexibilität. Dadurch, dass nur sehr wenige Konzepte vorgegeben sind, ist ein hohes Maß an Eigeninitiative notwendig. Dies erschwert auch den Vergleich zwischen verschiedenen Kanban-Projekten, da diese sehr unterschiedlich ablaufen können.

3.2 Personal Scrum

Scrum gilt als der älteste, agile Prozess und geht zurück auf ein Paper aus dem Jahre 1986 [TN86]. Für die Umsetzung gibt es ein Set an Methoden und

vordefinierten Regeln. Bei Scrum existieren grundsätzlich sechs Rollen, von denen drei essenziell sind und unten vorgestellt werden. Die verbleibenden drei sind externe (Management, Customer, User) Rollen.

Product-Owner

Der Product-Owner repräsentiert den Kunden. Seine Aufgabe ist es, sicherzustellen, dass das Team auch das liefert, was gewünscht wurde. Er schreibt typischerweise die User-Stories, priorisiert diese und fügt diese dem Product-Backlog hinzu. Es sollte in jedem Scrum-Team einen Product-Owner geben. Dieser kann auch ein Mitglied des Entwickler-Teams sein. Allerdings wird davon abgeraten, diese Rolle mit der des Scrum-Masters zu kombinieren.

Entwickler-Team

Das Entwickler-Team ist verantwortlich für die Erstellung eines potentiell lieferbaren Produkts am Ende eines jeden Sprints. Ein solches Team sollte aus drei bis neun Personen bestehen. Diese sind für das Durchführen der Arbeit verantwortlich (Analyse, Design, Entwicklung, ...). Das Entwickler-Team sollte grundsätzlich selbst-organisierend sein. Eine Verbindung zu einer Projekt-Management-Abteilung ist optional.

Scrum-Master

Der Scrum-Master ist für die erfolgreiche Umsetzung und die Einhaltung des Prozesses zuständig. Er soll Hindernisse aus dem Weg räumen und den Entwicklern die Möglichkeit geben, die Sprint-Ziele zu erreichen. Weiters gilt er

nicht als Team-Leader sondern dient quasi als Puffer zwischen dem Team und ablenkenden, äußeren Einflüssen. Eine wichtige Aufgabe des Scrum-Masters ist somit, das Team zu schützen, damit es sich besser auf die anstehenden Tasks konzentrieren kann.
Er arbeitet zwar mit dem Entwickler-Team zusammen, ist aber nicht Teil davon.

Ähnlich, wie bei XP, wird auch bei Scrum die Entwicklung in einzelnen Iterationen abgearbeitet, welche Sprints genannt werden. Zu Beginn von jedem Sprint gibt es ein Planungs-Meeting in welchem die Tasks für den aktuellen Durchlauf ausgewählt und abgeschätzt werden. Diese Tasks werden dann während eines Sprints abgearbeitet. Am Ende wird ein Review- bzw. Retrospektive-Meeting abgehalten.

Prozess-Evaluierung

Aktuell existieren einige Evaluierungen zur Eignung des Scrum-Prozesses für die Einzelentwicklung: [Sri09], [Pru11], [Woo09] bzw. eine Kombination von Scrum und XP: [Beh08]

Wie auch bei der Anwendung von XP für die Einzelentwicklung, ergibt sich auch beim Personal-Scrum die Herausforderung, den einzelnen Rollen gerecht zu werden. Die grundsätzlich als "Gruppenaktivitäten" vorhandenen Meetings zu Sprint-Planning, Daily-Scrums, Sprint-Reviews und Retrospektive wurden adaptiert und werden als Einzelperson durchgeführt.

Die 3 wichtigsten Artefakte Produkt-Backlog, Sprint-Backlog und Burndown-Chart wurden direkt übernommen und können in der selben Art und Weise eingesetzt werden.

Durch den Einsatz von Personal-Scrum wurde der Fokus auf die wichtigen Arbeiten gelenkt. Das Velocity-Tracking, sowie das Release-Planning halfen dabei, frühzeitig problematische Aufgaben aufzuzeigen. Dadurch hat man rechtzeitig die Möglichkeit, seine Ziele zu überdenken und dementsprechend zu agieren.

Ein wesentlicher Punkt war auch die Erfahrung, dass Zeitabschätzungen sehr oft falsch sind. Gute Abschätzungen zu machen erweist sich als schwierig. Die bestmögliche Genauigkeit erreicht man durch Vergleiche mit sehr ähnlichen Tasks, die bereits abgeschlossen wurden. Zeitabschätzungen benötigen viel Erfahrung. Im Gegensatz dazu erweist sich die Abschätzung von Story-Points als einfach und effektiv.

Weiters wurde festgestellt, dass Sprints in einer Größenordnung von einer Woche beim Personal-Scrum effizienter waren, als länger andauernde. Durch die kürzere Dauer war es einfacher den Überblick über die eigentlichen Ziele bzw. Tasks zu halten.

Die Burndown-Charts werden beim Personal Scrum meist auf Stunden-Basis erstellt, nicht wie bei "normalem" Scrum üblich anhand der Story-Points. Dies hat den Hintergrund, dass eine Einzelperson meist nicht in der Lage ist, entsprechend viele Stories in dem kurzen Zeitraum abzuschließen. Somit wären die Daten anhand von Story-Points nicht aussagekräftig genug.

Vergleich zu XP

Ein Wesentlicher Unterschied zwischen Scrum und XP ist, dass während eines Sprints keiner den Sprint-Backlog verändern darf. Diese Restriktierung kann situationsbedingt zu Problemen führen. Auf geänderte Anforderungen kann somit während eines Sprints nicht reagiert werden.
Bei Scrum wird im Gegensatz zu XP eher auf die Management-Seite und nicht detailliert auf die Entwickler-Seite eingegangen.

3.3 Personal XP

Hier werden andere Einzelentwickler-Ansätze für die Verwendung von XP vorgestellt.

XP for a single person team [AU08]

In diesem Paper wurden die Features von XP evaluiert und versucht, herauszufinden, wie diese für eine Einzelperson anwendbar sind. Auch hierbei geht es darum, die Produktivität und die Qualität bei der Einzelentwicklung zu steigern. Der dabei entstandene Prozess wurde "Personal Extreme Programming" (PXP) genannt. Es handelt sich hierbei um eine Kombination aus XP und dem Personal Software Process (PSP) [Hum96]

Bei PXP wird ebenfalls in Iterationen gearbeitet. Jede Iteration startet mit einer Planungsphase und einem Design-Update. Der Entwickler wartet drei Code-Basen: development, refactor und production. Nach der initialen Entwicklung wird der Code im Development-Bereich verwaltet und versioniert. Am Ende einer Iteration wird der Code getestet (Integrations-Tests) und

anschließend in den Refactoring-Bereich verschoben. Nach abgeschlossenem Refactoring werden die Acceptance-Tests durchlaufen und der Code wird für das Release in den Production-Bereich verschoben.

Anzumerken ist, dass hierbei auch Test-Driven-Development angewendet wird. Sowohl Acceptance- als auch Unit-Tests werden vor der eigentlichen Implementierung geschrieben. Sollten nach der Implementierung nicht alle Unit-Tests fehlerfrei ausführbar sein, so wird ein neuer Task für das Fixen des Codes mit höchster Priorität erstellt. Bei funktionierendem Code wird eine Integration und ein Refactoring durchgeführt. Nach dem Refactoring ist natürlich wieder das Ausführen von Tests notwendig.

Der PXP-Ansatz versucht eine Balance zwischen zu schwergewichtigen und zu einfachen XP-Methoden zu finden. Es wird versucht, das richtige Maß an Strenge einzuführen, ohne zu viel Overhead zu verursachen

Weiteres PXP-Konzept

Zu dieser Thematik existiert ein weiteres, unabhängiges Paper mit dem Titel "Personal Extreme Programming – An Agile Process for Autonomous Developers": [DKI09]

Hier wurde eine ähnliche Vorgehensweise, wie bei [AU08] gewählt. Es wurde eine Modifikation von PSP, kombiniert mit XP-Konzepten vorgestellt.

Es handelt sich dabei auch um einen iterativen Prozess. Dabei soll die Flexi-

bilität erhöht werden und Response-Zeiten sollen verkürzt werden. Die Vorgestellte Methode basiert zu einem großen Teil auf der Automatisierung täglicher Entwickler-Arbeiten. Dies soll die Performance verbessern und kürzere Delivery-Zeiten ermöglichen.

Folgende Prinzipien bilden eine Basis für PXP:

- PXP erfordert Disziplin. Der Entwickler ist verantwortlich für die Einhaltung des Prozesses und die Anwendung von PXP-Praktiken.
- Der Entwickler soll die tägliche Arbeit messen, aufzeichnen und analysieren.
- Der Entwickler soll von seinen Performance-Variationen lernen und anhand der gesammelten Daten den Prozess anpassen.
- PXP erfordert laufendes Testing.
- Fehlerbehebung soll in einer frühen Entwicklungsphase erfolgen. Hier sind die Kosten dafür niedriger.
- Der Entwickler soll versuchen, so viel wie möglich von der täglichen Arbeit zu automatisieren.

Weiters wurden 6 XP-Praktiken übernommen:

- Continuous Integration
- Simple Design
- Small Releases
- Refactoring

- Test-Driven-Development
- Spike-Solutions

Bei PXP basiert die Planung von Tasks hauptsächlich auf Reports aus vergangenen Projekten. Für jedes funktionelle Requirement wird ein Set an Tasks erstellt und jedem Task wird eine Kategorie zugeordnet. Diese Kategorien sind z.B. "Erstellung eines Unit-Tests", "Implementierung einer Klasse", "UI Design" usw. Abschätzungen werden hierbei anhand von vorangegangenen Abschätzungen von Tasks aus der selben Kategorie gemacht. Es soll dabei auch auf Daten von anderen Projekten zugegriffen werden.

Im PXP-Ablauf werden die Requirements-Analyse, sowie die Task-Planung üblicherweise nur ein Mal für das gesamte Projekt durchgeführt. Änderungen an den Requirements bzw. den Tasks sind allerdings innerhalb der Iterationen auch möglich.

Die Tasks, welche in der Planungs-Phase entstanden sind, werden dann in kleinere aufgeteilt und kategorisiert. Nach der Auswahl der Tasks für eine Iteration wird für diese ein Design erstellt. Nach Abschluss der Entwicklung werden die System-Tests durchlaufen. Anschließend wird eine Retrospektive durchgeführt und die nächste Iteration kann beginnen.

In dem vorgestellten Paper wurde auch ein Vergleich von PXP mit der Ad-hoc-Programming gemacht. Hierbei stellte sich heraus, dass für PXP 37.5% mehr an Zeit in die Planung investiert wurde. Auch hierbei war wieder zu erkennen, dass die Abschätzung von Tasks eine sehr schwierige Aufgabe ist. Ca. 64% der geplanten Tasks wurden zu gering bewertet. Wesentlich ist jedoch die Feststellung, dass durch den Einsatz von PXP die Fehleranzahl deutlich

zurück gegangen ist.

Es finden sich noch weitere Ansätze, wie z.B. [Jef05], die auf verschiedene Arten versuchen, XP zumindest teilweise für die Einzelentwicklung zu adaptieren.

4 Projektverlauf

4.1 Planung

Das Vorhaben ist, wie üblich, mit den User-Stories zu beginnen. Diese werden dann in der Planungsphase abgeschätzt und auf möglichst kleine Releases aufgeteilt. Es wurde versucht, sich weitgehend an die Konzepte aus [BF00] zu halten.

4.2 Initiale User-Stories

Im folgenden sind die User-Stories und ihre zugehörigen Tasks aufgelistet, welche bei der ersten Analyse erstellt wurden.

Es wurden Story-Points (StP) vergeben, welche die Komplexität wieder spiegeln. Ein Wert von 5 steht für die Höchste und 1 für die Niedrigste. Als komplex wird auch bewertet, was einen Hohen Aufwand zu Folge hat. Weiters wurde für die Stories eine entsprechende Priorität (PRI) vergeben. Hierbei bedeutet 1 die höchste Priorität. Somit gilt: je höher die angegebene Zahl, desto weniger wichtig wurde die jeweilige Story bewertet.

Wie bei XP vorgesehen wurden im 1. Schritt die User-Stories geschrieben. Danach können zu Beginn der jeweiligen Iteration aus selbigen dann die Tasks abgeleitet werden. Nach Abschätzung der Tasks in Stunden (EST) wurde der Gesamtaufwand in die jeweilige User-Story übertragen. Es wurden entsprechend viele Tasks abgeleitet und abgeschätzt, wie für die Durchführung einer Iteration benötigt wurden. Hierbei hat man vorsorglich eine Reserve eingerechnet, falls die Entwicklung schneller als geplant voran schreitet.

Die User-Stories wurden bereits nach den vergebenen Prioritäten sortiert.

4.2.1 Basissystem

Für die Auswahl des Basissystems wurden verschiedenste Linux-Distributionen evaluiert. Aufgrund der einfachen Möglichkeit ein Minimalsystem zu erzeugen bzw. der Aktualität der Pakete wurde hierfür Ubuntu gewählt. Da KDE sehr viele Anpassungsmöglichkeiten bietet, wurde es als Desktop-Oberfläche verwendet.

ID	Story	PRI	StP
US01	Es wird ein Basissystem benötigt, welches eine minimale Desktop-Oberfläche startet	1	4

Tabelle 3: User-Story US01

ID	Task	EST
T01	Erstellen eines Minimalsystems mittels debootstrap	30
T02	Entsprechende Pakete auswählen und installieren	5
T03	Basiskonfiguration (locale, inputrc, ...)	10
T04	Startscript/Anwendung erzeugen, wodurch die Netzverfügbarkeit überprüft wird und anschließend entweder das Reparaturprogramm gestartet wird oder eine Fehlermeldung ausgegeben wird	5
T05	Anlegen einer VirtualBox-VM	2

Tabelle 4: Tasks zu US01

ID	Story	PRI	StP
US02	Das System sollte innerhalb möglichst kurzer Zeit auf einem Standard-PC gestartet sein	2	3

Tabelle 5: User-Story US02

ID	Task	EST
T06	Standardmäßig installierte Pakete, die nicht benötigt werden herausfinden und entfernen	10
T07	Startup-Scripts anpassen um das Laden von nicht unmittelbar benötigten Komponenten zu verhindern	15

Tabelle 6: Tasks zu US02

ID	Story	PRI	StP
US03	Beim Starten der Desktop-Umgebung soll automatisch ein Standard-User eingelogged werden	3	1

Tabelle 7: User-Story US03

ID	Task	EST
T08	Auto-Login konfigurieren	5

Tabelle 8: Tasks zu US03

ID	Story	PRI	StP
US04	Aus dem Grundsystem soll automatisch die aktuelle Anwendung für das Rücksichern vom Server gestartet werden	4	2

Tabelle 9: User-Story US04

ID	Task	EST
T09	Loader-Anwendung programmieren	10
T10	Server aufsetzen	20
T11	Scripts und Anwendung von Server-Share in einen lokalen Ordner kopieren und entsprechende starten	15

Tabelle 10: Tasks zu US04

ID	Story	PRI	StP
US05	Eine Konsole soll sich per Tastendruck ein- und ausblenden lassen	5	1

Tabelle 11: User-Story US05

ID	Task	EST
T12	Entsprechendes Tool auswählen, installieren und konfigurieren	5

Tabelle 12: Tasks zu US05

ID	Story	PRI	StP
US06	Bei einem Netzwerkproblem soll eine entsprechende Fehlermeldung angezeigt werden. Dem Benutzer sollen Möglichkeiten zur Fehlerbeseitigung geboten werden	6	2

Tabelle 13: User-Story US06

ID	Task	EST
T13	GUI-Tool programmieren, welches Fehler anzeigt wird	15
T14	Möglichkeiten zur Fehlerbehandlung einbauen	5

Tabelle 14: Tasks zu US06

ID	Story	PRI	StP
US07	Das System soll von einer Live-CD bootbar sein	7	4

Tabelle 15: User-Story US07

ID	Task	EST
T15	Einbinden von casper, um eine Live-Umgebung zu booten	10
T16	Änderung der Init-Scripts, sodass nur die gewünschten Livesystem-Anpassungen beim booten vorgenommen werden	20
T17	Ramdisk und Kernel für Livesystem auswählen und modifizieren	15
T18	Erstellen eines scripts zur ISO-Generierung	5

Tabelle 16: Tasks zu US07

ID	Story	PRI	StP
US08	Es soll möglich sein, das System auf einer Partition direkt am Rechner zu installieren	8	4

Tabelle 17: User-Story US08

ID	Task	EST
T19	Script zum Anlegen der Partition erstellen	15
T20	Script zum Kopieren der Daten aus dem Livesystem erstellen	20
T21	Livesystem anpassen, um von einer Partition am Rechner zu starten	20
T22	Script zum Einspielen der Reparaturpartition erzeugen	15
T23	Script zum Einspielen des Bootloaders erzeugen	5

Tabelle 18: Tasks zu US08

ID	Story	PRI	StP
US09	Ein Internet-Zugang per Browser soll möglich sein	9	1

Tabelle 19: User-Story US09

ID	Task	EST
T24	Browser auswählen, installieren und entsprechend verknüpfen	2

Tabelle 20: Tasks zu US09

ID	Story	PRI	StP
US10	Das System soll per VNC Remote-Administrierbar sein	10	2

Tabelle 21: User-Story US10

ID	Task	EST
T25	Entsprechendes Tool installieren und konfigurieren	5

Tabelle 22: Tasks zu US10

4.2.2 User-GUI

ID	Story	PRI	StP
US11	Das Rücksetzten einzelner Partitionen soll vom Benutzer gestartet werden können	11	3

Tabelle 23: User-Story US11

ID	Task	EST
T26	Script zum Rücksetzen einer Partition erzeugen	20
T27	Datenbankschema anpassen	5
T28	Datenbank auslesen und GUI-Elemente für die jeweilige Partitions-Konfiguration anlegen	15
T29	Bei Rücksetzen einer Windows-XP-Partition passende boot.ini schreiben	10
T30	Partitions-Bootsektor schreiben	5
T31	Partitions-Info in Bootsektor updaten	5
T32	Entsprechende Bootloader-Konfiguration erzeugen und Bootloader updaten	10

Tabelle 24: Tasks zu US11

ID	Story	PRI	StP
US12	Nach dem Rücksetzen der ausgewählten Partition sollen je nach Konfiguration die entsprechenden Software-Pakete eingespielt werden	12	2

Tabelle 25: User-Story US12

ID	Task	EST
T33	Datenbankschema anpassen	5
T34	Script zum Einspielen der Software-Pakete auf die jeweilige Partition erstellen	15
T35	Datenbank auslesen und entsprechendes Script ausführen	15
T36	Post-Install-Script erzeugen, welches das Setup der einzelnen Pakete beim ersten Hochfahren startet	15

Tabelle 26: Tasks zu US12

ID	Story	PRI	StP
US13	Für das Kopieren bzw. Extrahieren soll ein Fortschrittsbalken angezeigt werden	13	1

Tabelle 27: User-Story US13

ID	Task	EST
T37	Script für Kopierfortschritt anlegen und den Fortschrittsbalken entsprechend setzen	10
T38	Script für das den Extraktionsfortschritt einer Partition anlegen und den Fortschrittsbalken entsprechend setzen	10
T39	Script für das den Extraktionsfortschritt eines Softwarepaketes anlegen und den Fortschrittsbalken entsprechend setzen	5

Tabelle 28: Tasks zu US13

ID	Story	PRI	StP
US14	Im User-Interface sollen Rechnername, Rechnernummer, MAC-Adresse, IPv4-Adresse und Raum angezeigt werden	14	1

Tabelle 29: User-Story US14

ID	Task	EST
T40	Entsprechendes Anzeigeelement in der GUI anlegen (z.B. Tabelle)	5
T41	Script zum Auslesen der MAC-Adresse anlegen	5
T42	Script zum Auslesen der aktuellen IP-Adresse anlegen	5
T43	Datenbankschema anpassen um entsprechende Daten zu speichern	10
T44	Rechnername, Rechnernummer und Raum anhand der MAC-Adresse in der Datenbank suchen und anzeigen	10

Tabelle 30: Tasks zu US14

ID	Story	PRI	StP
US15	Die benötigten Images und Software-Pakete sollen lokal am Rechner gecached werden, sodass keine zusätzliche Netzwerk-Last erzeugt wird, wenn das Image am Server nicht geändert wurde	15	2

Tabelle 31: User-Story US15

ID	Task	EST
T45	Caching-Algorithmus wählen und implementieren	20
T46	Script für das Caching erzeugen	10

Tabelle 32: Tasks zu US15

ID	Story	PRI	StP
US16	Nach erfolgreichem Rücksetzen soll der PC automatisch neu gestartet werden	16	1

Tabelle 33: User-Story US16

ID	Task	EST
T47	Reboot-Script erzeugen (Aufräumarbeiten, "sanftes" Beenden)	15
T48	Reboot-Script in das Script für das Rücksetzen einbauen	1

Tabelle 34: Tasks zu US16

ID	Story	PRI	StP
US17	Die GUI soll einen Textbereich besitzen, in dem die Statusmeldungen der einzelnen Befehle ausgegeben werden	17	1

Tabelle 35: User-Story US17

ID	Task	EST
T49	GUI-Element für LOG-Bereich anlegen	2
T50	Output der einzelnen Scripts sammeln und im LOG-Bereich ausgeben	10

Tabelle 36: Tasks zu US17

ID	Story	PRI	StP
US18	Ein Admin-Dialog soll über einen Login-Dialog (PAM) verfügbar gemacht werden	18	3

Tabelle 37: User-Story US18

ID	Task	EST
T51	Anlegen eines Admin-Dialogs	10
T52	Erstellen eines Login-Dialogs mit PAM-Wrapper	20
T53	Anlegen einer entsprechenden PAM-Konfiguration im System	10

Tabelle 38: Tasks zu US18

ID	Story	PRI	StP
US19	Ein GUI-Tool zur Festplattendiagnose soll verfügbar sein	19	1

Tabelle 39: User-Story US19

ID	Task	EST
T54	Passende Tools evaluieren	5
T55	Gewähltes Tool installieren und entsprechend verknüpfen	1

Tabelle 40: Tasks zu US19

4.2.3 Admin-GUI

ID	Story	PRI	StP
US20	Das manuelle Einspielen beliebiger Images auf beliebige Partitionen soll möglich sein (kein automatischer Reboot)	20	2

Tabelle 41: User-Story US20

ID	Task	EST
T56	GUI-Elemente anlegen (Auswahl von Image, Partition, Dateisystem)	10
T57	Script für manuelles Rücksetzen erzeugen (kein automatischer Reboot)	15

Tabelle 42: Tasks zu US20

ID	Story	PRI	StP
US21	Das Erstellen neuer Images soll möglich sein	21	2

Tabelle 43: User-Story US21

ID	Task	EST
T58	GUI-Elemente für Image-Erstellung anlegen	2
T59	Script für Image-Erzeugung anlegen	10
T60	Erzeugtes Image auf Server und in Cache ablegen	5
T61	Datenbankschema anpassen und extrahierte Größe in Datenbank ablegen (für Fortschrittsbalken)	10

Tabelle 44: Tasks zu US21

ID	Story	PRI	StP
US22	Das Erstellen von Software-Paketen soll möglich sein	22	3

Tabelle 45: User-Story US22

ID	Task	EST
T62	GUI-Elemente für das Erzeugen von Softwarepaketen anlegen	5
T63	Script für das Erzeugen von Softwarepaketen anlegen	15
T64	Softwarepakete auf Server und in Cache ablegen	5
T65	Datenbankschema anpassen und extrahierte Größe in Datenbank ablegen (für Fortschrittsbalken)	10

Tabelle 46: Tasks zu US22

ID	Story	PRI	StP
US23	Das automatische Rücksetzen aller Partitionen soll möglich sein (anschließend Reboot)	23	2

Tabelle 47: User-Story US23

ID	Task	EST
T66	GUI-Element für das automatische Rücksetzen anlegen	1
T67	Script für automatisches Rücksetzen aller Partitionen erzeugen	10
T68	Bootmanager, Partitions-Sektoren und eventuell boot.ini schreiben	10
T69	Reboot nach Abschluss aller Aktionen einrichten	1

Tabelle 48: Tasks zu US23

ID	Story	PRI	StP
US24	Es soll eine Möglichkeit geben, die Festplatte automatisch zu partitionieren	24	3

Tabelle 49: User-Story US24

ID	Task	EST
T70	GUI-Element für automatische Partitionierung anlegen	1
T71	Script für automatische Partitionierung erzeugen	10
T72	Nach Partitionierung je nach Konfiguration Reparatursystem wieder lokal einspielen	15

Tabelle 50: Tasks zu US24

ID	Story	PRI	StP
US25	Das Reparatur-System soll in eine beliebige Partition eingespielt werden können	25	3

Tabelle 51: User-Story US25

ID	Task	EST
T73	Entsprechende GUI-Elemente anlegen (Auswahl der Partition)	-
T74	Script erzeugen, welches das Reparatur-System in eine beliebige Partition einspielt	-
T75	Anschließend Bootmanager-Konfiguration anlegen und Bootmanager neu schreiben	-

Tabelle 52: Tasks zu US25

ID	Story	PRI	StP
US26	Es soll eine Möglichkeit geben, den Bootmanager manuell neu zu schreiben	26	1

Tabelle 53: User-Story US26

ID	Task	EST
T76	Entsprechendes GUI-Element anlegen	1
T77	Script zum aktualisieren des Bootmanagers erzeugen	5

Tabelle 54: Tasks zu US26

4.2.4 Allgemeine Tasks

Hier finden sich Tasks, die aufgrund gewisser Randbedingungen entstanden sind und somit keiner konkreten User-Story zugeordnet werden können.

ID	Story	PRI	StP
US27	Gemeinsam genutzte Funktionalitäten	-	-

Tabelle 55: User-Story US27

ID	Task	EST
T78	Partitionen nach dem Schema /mnt/disc?/part? mounten	5
T79	Sqlite-Datenbank inklusive Schema für Testzwecke anlegen	15
T80	MySQL-Datenbank inklusive Schema für Echtbetrieb anlegen	10
T81	Samba-Share mit Scripts, Images und Softwarepaketen anlegen	5
T82	Samba-Share am Client automatisch mounten	5
T83	Minimal notwendige scripts erzeugen und am Server ablegen	5

Tabelle 56: Tasks zu US27

4.3 Iterationen

In diesem Abschnitt werden einzelne Iterationen, bei denen es besondere Vorkommnisse gab herausgehoben.

Iteration 1

Zum Projektstart wurden entsprechend Stories ausgewählt und Tasks abgeschätzt, die einen Gesamtaufwand von 79 Stunden besitzen.

Während des Entwickelns gab es einige unerwartete Hindernisse. Kleinere Bugs im aktuellen Distributionsstand mussten behoben werden. Weiters stellte sich heraus, dass einzelne Tasks nicht wie technologisch angedacht umsetzen ließen.
Daraus resultiert auch die relativ geringe Velocity nach Abschluss der 1. Iteration.

Anfangs wurden auch bereits Tasks aus der allgemeinen User-Story US27 implementiert.

Iterationen 5-7

Hier ist hervorzuheben, dass die Story US10 anfangs auch dafür gedacht war, um Tests über die VNC-Verbindung laufen zu lassen. Im Projektverlauf stellte sich allerdings heraus, dass diese Methode nicht praktikabel ist. Aus diesem Grund wurde US10 neu priorisiert und erst in einer späteren Iteration abgeschlossen.

Iteration 10

In der 10. und letzten Iteration wurden lediglich Tasks im Ausmaß von 42 Stunden eingeplant und abgeschlossen. Dies resultiert einfach daraus, dass keine weiteren User-Stories mehr vorhanden sind. Mit Ende dieser Iteration wurde die Software in Version 4.0.0 fertig gestellt. Version 4 deshalb, da die Vorgänger-Software die Major-Versions-Nummer 3 besaß.

Anmerkung US24: Die automatische Partitionierung ist prinzipbedingt nur beim Booten von einer Live-CD ratsam. Grundsätzlich wird dem Benutzer diese Möglichkeit auch bei der lokal installierten Version angeboten, allerdings hängt der positive Ausgang davon ab, welche Daten aktuell im RAM verfügbar sind.

Anmerkung US25: Diese Story wurde während des Projektverlaufs verworfen. Es hat sich heraus gestellt, dass die fixe Zuordnung der Reparaturpartition auf die Nummer 1 einige Vorteile mit sich bringt. Es wurde somit entschieden, US25 fallen zu lassen und nicht zu implementieren.

5 Testvorgang

Das Testen ist ein wichtiger Bestandteil der Softwareentwicklung. TDD war eine wesentliche Komponente in diesem Projekt. Durch die Anwendung dieser Entwicklungsmethode wird verhindert, dass Code entsteht, welcher nicht mehr mittels einfacher Unit-Tests überprüfbar ist. Man wird dazu gezwungen, das Software-Design dementsprechend zu gestalten.

Die Tests dienen nicht nur zur Überprüfung des Source-Codes. Sie dienen auch als Dokumentation. Die Testfälle beschreiben die Art und Weise, wie sich die entsprechende Funktionalität verhält bzw. wie die Verwendung einzelner Klassen angedacht wurde. Man hat dadurch direkt ein Beispiel für die korrekte Benutzung vorgegeben. Es muss für einen beliebigen Programmierer möglich sein, anhand dieser Vorgaben die passende Implementierung zu schreiben.

Für die Entwicklungs- bzw. Testumgebung stellte sich das Problem, dass aktuelle PCs mit entsprechender RAM-Bestückung ein 64-Bit-System erfordern. Die Software sollte allerdings auch auf 32-Bit-CPUs lauffähig sein. Für dieses Projekt wurde deshalb eine 32-Bit-Chroot-Umgebung angelegt. In dieser wurden dann auch die entsprechenden Tools, wie das Qt-SDK usw. installiert.

Um z.B. für das Testen des Rücksichern einer Partition keine gesonderte Festplatte heranziehen zu müssen, wurde eine spezielle Funktion von Linux verwendet. Es bietet die Möglichkeit, eine Datei als FFestplatteßu partitionieren. Dies führt dazu, dass man alle Tests, für die Hard-Disks erforderlich wären, mit Hilfe von Dateien abgewickelt werden können. Nach Ende eines

Unit-Tests können diese einfach wieder gelöscht werden. Dies vereinfacht die Wiederherstellung einer immer gleichen Ausgangsbasis wesentlich.

Weiters wurde für das Testen eine Virtual-Box-VM angelegt. Von dieser aus kann eine Live-CD gestartet werden. Zusätzlich ist es auch möglich, das System in eine Partition auf einer Festplatte zu installieren.

Im folgenden werden die verwendeten Test-Arten behandelt.

5.1 Test-Driven-Development

"Test-Driven Development is one way, an effective way, to weave testing into the fabric of software development. It's Kevlar for your code." [Gre11]

Testgetriebene Entwicklung ist ein Konzept, welches die Fehleranfälligkeit und die Qualität von Software in einem möglichst frühem Stadium verbessern soll. Wichtig ist, dass nicht erst im Projektverlauf auf TDD umgestellt wird, sondern, dass dies bereits von Anfang an durchgeführt wird. Zu einem späteren Zeitpunkt kann es erheblichen Zusatzaufwand bedeuten. Oft existiert bereits Code, der nicht mit Bedacht auf die Testbarkeit entwickelt worden ist. Dieser muss dann dementsprechend Verändert bzw. ausgetauscht werden.

In der klassischen Entwicklung werden oft lediglich Code-Reviews zur Verbesserung der Qualität eingesetzt. Diese sind zwar gut und auch notwendig, allerdings kann damit nur sehr schwer eine 100%ige Aussage über die geschriebenen Zeilen getroffen werden. Will man das tatsächliche Verhalten überprüfen, so ist dies nur durch die Ausführung des Codes möglich.

Tests werden beim TDD immer vor der eigentlichen Entwicklung von Produktiv-Code geschrieben. Wesentlich ist hierbei, dass diese automatisiert ausgeführt werden können. Die Software wird dann in kleinen Schritten Element für Element aufgebaut. Hier ein schöner Vergleich mit dem Klettern:

"Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken. [...] Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. [...] Der bis zum Haken erkletterte Weg gehört ihm in jedem Fall, selbst wenn ihm ein Fehler unterläuft." [Wes05]

Übertragen auf die Softwareentwicklung bedeutet das folgendes:
Wie das Setzten der Haken, bedeutet auch das Schreiben von Test-Cases einen zusätzlichen Aufwand. Diese gewähren jedoch ein großes Maß an Sicherheit. Wird neuer Code geschrieben, kann er durch einfaches Ausführen der zugehörigen Tests überprüft werden. Die Fallhöhe bei einem Sturz wird durch die Haken wesentlich verringert: Sollte ein Fehler vorhanden sein, kann dieser somit auf einfache Art und Weise gefunden und behoben werden.

Beim Test-Driven-Development hat sich das Mantra "red/green/refactor" etabliert (siehe Abbildung 4). Hierbei steht "red" für den fehlschlagenden Test-Case und "green" für korrekt implementierten Produktiv-Code. Es wird versucht, jeden einzelnen Test mit möglichst wenig Coding-Aufwand positiv ausführbar zu machen. Auf das Refactoring wird im folgenden Abschnitt genauer eingegangen.

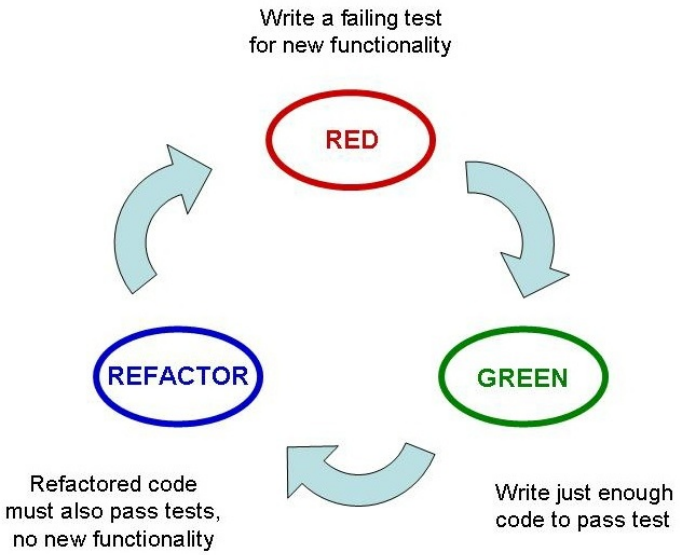

Abbildung 4: TDD-Zyklus [Obj10]

Produktivität

Im Jahr 2005 wurde eine Studie zur Effektivität der Test-First-Entwicklung veröffentlicht: [EMT05]

Eine Gruppe schrieb hierbei die Tests vor der Implementierung, die andere erst danach. Beide Gruppen folgen einem inkrementellen Prozess. Features wurden erst nach und nach hinzugefügt.

Bei diesem Vorgehen stellte sich heraus, dass die Verwendung des Test-First-Konzepts darin resultierte, dass mehr Tests geschrieben wurden. Allerdings fand man auch heraus, dass Programmierer, welche mehr Tests geschrieben hatten auch zu einer höheren Produktivität tendierten.

Vorteile von TDD

Durch die Verwendung von TDD ist es meist weniger oft notwendig, Code zu

debuggen. Fehler können durch die hohe Anzahl an Tests einfacher lokalisiert werden. TDD gibt eine gewisse Richtung beim Design vor. Dieses muss so gewählt werden, dass der programmierte Code gut testbar ist. Die Testfälle spiegeln somit die Verwendung des Codes wieder. Hierbei werden die einzelnen Software-Teile genau so getestet, wie sie später auch verwendet werden sollen. Dadurch, dass die Tests vor der Implementierung geschrieben werden, muss man sich auch bereits vorher Gedanken über Interfaces usw. machen. Ohne ein Konzept kann somit die Entwicklung nicht begonnen werden.

Nachteile von TDD

Die Verwendung von TDD stößt an ihre Grenzen, wenn es schwierig wird, den Erfolg eines Test-Cases zu überprüfen. Dies ist der Fall, wenn z.B. ein User-Interface oder Daten einer Netzwerkverbindung den Ausgang des Tests beeinflussen. Hierbei ist es dann oft notwendig, die Verbindung zur Außenwelt zu simulieren (mocking).

Da sowohl der Test, als auch die Implementierung vom selben Entwickler geschrieben wird, kann es durchaus passieren, dass dadurch Fehlerfälle ungetestet bleiben. Vergisst man beim Schreiben der Test-Cases auf eine Randbedingung, so kann es vorkommen, dass man dies auch bei der Implementierung übersieht. Um dieses Problem weitgehend zu verhindern, gibt es Methoden wie z.B. Ping-Pong-Pair-Programming. [CC11]
Hierbei schreibt ein Entwickler den fehlschlagenden Test, die Implementierung wird jedoch von einem anderen umgesetzt. Dieses Konzept ist bei der Einzelentwicklung mangels an Programmierern leider nicht anwendbar. Es ist somit wichtig, sich im Vorhinein sehr genau zu überlegen, wie der Code

später verwendet werden soll.

TDD spielt mit einigen anderen XP-Konzepten direkt zusammen:

5.1.1 Refactoring

Refactoring bedeutet, interne Code-Änderungen zu machen, ohne dass diese externe Auswirkungen zu Folge haben. Dabei soll z.B. das Design oder die Erweiterbarkeit der Software verbessert werden. Hierbei können auch nichtfunktionale Anforderungen wie Verkürzung von Antwortzeiten abgehandelt werden. Auch die Verbesserung der Lesbarkeit und Wartbarkeit des Codes fällt ebenfalls darunter. Dieses Vorgehen kann man mit der algebraischen Umformung aus der Mathematik vergleichen. Hierbei werden Terme abgeändert um die Leserlichkeit zu erhöhen. Deren Bedeutung bleibt jedoch die selbe.

Durch die Verwendung von TDD sind Refactorings wesentlich einfacher möglich. Jede Code-Änderung kann natürlich neue Fehler beinhalten. Die testgetriebene Entwicklung stellt allerdings die korrekte Funktionalität der einzelnen Module sicher. Somit kann nach einem Refactoring ganz einfach durch das Starten von Test-Cases überprüft werden, ob sich die Software sich noch korrekt verhält.

Wie in Abbildung 4 zu erkennen ist, ist das Refactoring ein wesentlicher Bestandteil des TDD-Zyklus, welcher laufend durchgeführt werden soll. Auf jeden Fall aber ist es notwendig, wenn z.B. doppelter Code existiert, Methoden zu lang sind, zu viele Parameter an eine Funktion übergeben werden, oder einfach die Benennung von Variablen, Funktionen usw. inkorrekt ist.

5.1.2 Continuous integration

Laufende Integration ist notwendig um sicherzustellen, dass die durchgeführten Änderungen keine Seiteneffekte hervorrufen. Deshalb ist es auch wichtig, dass diese so oft wie möglich passiert. Durch die testgetriebene Entwicklung wird zumindest sichergestellt, dass das jeweilige Modul für sich gesehen korrekt arbeitet.

Wird neuer Code in das Source-Code-Verwaltungssystem eingecheckt, so sollten damit auch gleich automatisiert die jeweiligen Tests durchgeführt werden. Dabei werden alle verfügbaren Tests durchlaufen, nicht nur jene, die unmittelbar den geänderten Code betreffen. Dies stellt sicher, dass auch andere Module noch korrekt funktionieren.

5.1.3 Incremental Design

In der klassischen Software-Entwicklung wird zu Beginn ein Design festgelegt, welches im Projektverlauf grundsätzlich nicht mehr geändert werden soll. Beim inkrementellen Design werden zwar ständig Design-Änderung, allerdings nur in kleinen Schritten vorgenommen. Im Deutschen wird dies oft auch als “evolutionäres Design” bezeichnet.

Die Anwendung von TDD unterstützt dieses Konzept. Nach einer Änderung kann wiederum durch simples Ausführen der Test-Cases sichergestellt werden, dass die Software noch korrekt funktioniert. Hier spielt auch das Konzept der laufenden Integration mit. Dadurch wird wieder Modulübergreifend getestet und man erkennt direkt die Auswirkungen der Änderung auf andere Codeteile.

5.2 Unit-Tests

Für jede Funktionalität wurde vor der eigentlichen Implementierung mindestens ein Unit-Test für den gewünschten Gut-Fall geschrieben. Dieser muss, wie bereits erwähnt, fehl schlagen. Dadurch erreicht man, dass zumindest der gewünschte Output bei korrekter Verwendung sichergestellt wird. Es ergibt sich daraus allerdings keine 100%ige Testabdeckung. Diese muss gesondert betrachtet werden.

Bei der gewählten Vorgehensweise ist zu erwähnen, dass durchaus Code für einzelne Klassen vor den Unit-Tests geschrieben werden darf. Es ist z.B. praktikabler, die Klassen-Rümpfe bzw. Interfaces in erster Instanz anzulegen. Es existieren verschiedenste Tools, die dann die Möglichkeit bieten, aus diesen die zugehörigen, fehlschlagenden Unit-Tests zu generieren.

Für den Einsatz von TDD ist eine Unterstützung mittels entsprechendem Tooling unumgänglich. Es ist notwendig, dass es eine Möglichkeit gibt, für eine beliebige Klasse Test-Stubs zu erstellen. Bei diesem Projekt wurde dafür QAutoTestGenerator [Mey10] verwendet. Damit ist es möglich, nach dem schreiben eines Header-Files den dazu passenden Test-Code automatisch zu generieren. Dies automatisiert das Erstellen eines Unit-Test so weit, dass für einen Großteil der Funktionen nur mehr das Spezifizieren der Übergabeparameter, sowie des dazu erwarteten Return-Wertes notwendig ist. Ohne diese Unterstützung würde das Schreiben von Unit-Tests einen wesentlichen Mehraufwand bedeuten.

Soll eine Funktionalität geändert werden, so muss vorher der entsprechende Unit-Test angepasst werden. Selbiges gilt auch für das Hinzufügen von Funk-

tionen. Es darf somit kein Code verändert werden, so lange kein Test dafür existiert.

Eine Funktionalität wird als implementiert gewertet, sobald alle zugehörigen Test-Cases ohne einen Fehler ausgeführt werden können.

Da das für die Entwicklung verwendete Qt-Framework bereits eine Unit-Test-Komponente (QTestLib) besitzt, wurde diese auch herangezogen.
Ein Beispiel für die Verwendung der ConfigReader-Klasse:

```
1 void ConfigTest::databaseTestCase()
2 {
3   ConfigReader cfgReader("test.cfg");
4   QVERIFY2(cfgReader.getDatabaseType() == "SQLITE",
5     "Wrong database type.");
6 }
```

Listing 1: Einfacher Unit-Test

Hierbei ist anzumerken, dass das Qt-Framework nicht, wie z.B. in JUnit üblich, verschiedenste Assert-Funktionen bietet. Es wird lediglich das Ergebnis von einer beliebigen, boolschen Operation ausgewertet. Liefert der aktuelle Verify-Code "true", so wird mit der Ausführung fortgefahren. Andernfalls wird der Fehler gelogged und der Test wird abgebrochen. Anschließend führt das Framework automatisch den nächsten Unit-Test aus.

Vor jedem Test wird, sofern vorhanden, die Funktion "init" ausgeführt. Diese wird verwendet, um eine definierte Umgebung für den Testfall herzustellen. (z.B. das Erzeugen einer Datei mit entsprechender Partitionierung)

Alle hier angelegten Ressourcen müssen auch wieder freigegeben werden. Hierfür wird nach Ende eines Tests die Funktion “cleanup” aufgerufen. Diese ist ebenfalls optional. Wurde sie definiert, so wird sie immer ausgeführt. Es ist dabei irrelevant, ob die Ausführung des Tests fehlerfrei war, oder nicht.

Die Überprüfung des C++-Codes lässt sich somit sehr einfach durchführen. Es wurde beschlossen, für die Bash-Scripte ebenfalls QTestLib zu verwenden. Somit ist es möglich, mit einem Tool die gesamte Code-Basis zu testen. Anders als beim C++-Teil ist es dadurch nicht mehr möglich, auf Funktionsebene zu Testen. Aufgrund dessen wurde versucht, die Scripts von ihrem Umfang her eher klein zu halten. Einzelne Bash-Dateien liefern somit nur relativ wenig Output. Dieser wird dann mittels Test-Cases überprüft:

```
1 void ScriptTest::getServerImageSize()
2 {
3   QProcess scriptProcess;
4   scriptProcess.start("scripts/get_server_imagesize.sh",
5     QStringList() << "test_img");
6   scriptProcess.waitForFinished();
7   QVERIFY2(scriptProcess.readAll() == "100000",
8     "Wrong image size returned.");
9 }
```

Listing 2: Unit-Test Bash-Script

5.3 Integrationstests

Integrationstests dienen dazu, das Zusammenspiel einzelner Komponenten zu überprüfen. Sie können auch auf nicht-funktionale Anforderungen, wie z.B. Antwortzeiten testen. Unit-Tests validieren lediglich, ob eine einzelne

Komponente für sich selbst gesehen korrekt arbeitet. Somit wird bei Integrationstests z.B. der korrekte Datenaustausch zwischen mehreren Modulen kontrolliert. Dabei werden nur mehr die entsprechenden Interfaces verwendet und Internas bleiben unbekannt. Es handelt sich somit um Black-Box-Tests. Hier ein kurzes Beispiel:

```
1 void IntegrationTests::readComputerData()
2 {
3   ConfigReader cfgReader("test.cfg");
4   QVERIFY2(SqliteDB::open(cfgReader.getDatabaseName()),
5     "Could not open database.");
6   ComputerInfo computerInfo;
7   QVERIFY2(computerInfo.getName() == "Test_PC",
8     "Wrong computer name.");
9 ...
```

Listing 3: Einfacher Integrationstest

Die Datei "test.cfg" wird von ConfigReader eingelesen. Der Datenbankname wird ausgelesen und mit Hilfe der Klasse SqliteDB wird eine neue Datenbankverbindung angelegt. Anschließend wird ein Objekt vom Typ ComputerInfo mit Hilfe des Default-Konstruktors erzeugt. Ohne übergebene Parameter verwendet dieser die zuletzt erzeugte Datenbankverbindung. Nun können die einzelnen Computer-Daten ausgelesen werden.

5.4 Systemtests

Wie der Name bereits sagt, wird hierbei das gesamte System getestet. Darunter fallen nicht nur Test, welche die Funktionalität überprüfen, sondern auch z.B. Performance- oder Lasttests. Es werden somit auch nicht-funktionale

Anforderungen getestet.

Als Testumgebung wurde eine virtuelle Maschine gewählt. Dies hat den Vorteil, dass man durch die Verwendung der Snapshot-Funktion jederzeit zu einem definierten Ausgangszustand zurückkehren kann, egal welche Dateisystemänderungen durch den Testvorgang verursacht wurden. Das folgende Beispiel überprüft, ob die Reparatur-Applikation vom Basissystem automatisch gestartet wurde.

```
1  void RepairsystemTest::startTestCase()
2  {
3    QProcess::execute("/etc/init.d/vboxdrv",
4      QStringList() << "start");
5    QProcess::startDetached("VirtualBox");
6    QProcess::execute("VBoxManage",
7      QStringList() << "startvm" << "Startup_Test");
8    QTcpServer tcpserver;
9    tcpserver.listen(QHostAddress::Any, 32123);
10   QVERIFY2(tcpserver.waitForNewConnection(100000),
11    "ARS-Client did not connect.");
12 }
```

Listing 4: GUI-Start-Test

Bei diesem Testfall werden zu Beginn die Virtualisierungs-Treiber geladen. Anschließend wird VirtualBox gestartet. Mittels dem VBoxManage-Kommando kann nun die VM gestartet werden. Die Maschine ist so konfiguriert, dass nach dem Einschalten automatisch von der Live-CD gebootet wird. Aus diesem System wird anschließend automatisch die eigentliche GUI-Applikation gestartet. Selbige versucht dann, eine Verbindung zu dem Server

mit IP-Adresse 10.0.2.2 auf Port 32123 aufzubauen. 10.0.2.2 repräsentiert in einer VirtualBox-VM, deren Netzwerkkonfiguration auf NAT gestellt ist, immer den Host. Es wird damit sichergestellt, dass die Anfrage immer zum Test-System gelangt.

Obiger Test-Case wartet auf die Verbindungsanfrage durch die Reparatur-Applikation. Wurde sie durchgeführt, gilt der Test als positiv abgeschlossen. Sollte ein Timeout auftreten, so wurde die GUI-Applikation nicht korrekt gestartet und der Test schlägt fehl.

5.5 Manuelle Tests

Der Aufwand, komplexe Abläufe automatisiert zu Testen, kann mitunter sehr groß werden. Oft ist es einfacher, einzelne Funktionalitäten manuell zu Testen. Es wurden entsprechende Listen für den Ablauf solcher Tests erstellt.

Folgendes Beispiel behandelt einen Kompletten Durchlauf für das Reparatursystem, beginnend mit dem Start der VM, bis zum Start des Wiederhergestellten Systems:

Aufgabe/Erwartung	OK?
Test-VM starten	
Basissystem bootet	
Auto-Login funktioniert	
Reparatur-GUI wird automatisch gestartet	
Reparatur-Button für 1. Partition anclicken (WinXP)	
Image wird von Server kopiert (Anzeige)	
Image wird extrahiert (Anzeige)	
System startet automatisch neu	
Live-CD wurde ausgeworfen	
Bootmanager von HDD wird korrekt angezeigt	
[ENTER] Um erstes System zu starten	
Windows XP startet erfolgreich (Desktop geladen)	
Fehlerbeschreibung:	

Tabelle 57: Manueller Test

5.6 Abnahme-Tests

Abnahme- bzw. User-Acceptance-Tests sind solche, die voraussetzend für die Übergabe an den Kunden sind. Es wird explizit auf die Anforderungen des Users hin getestet. Oft werden hierfür eigene Tests erstellt. Da bei der Einzelentwicklung der Entwickler und der Repräsentant des Kunden oft ein und die selbe Person sind, wurde auf eine Entwicklung gesonderter Tests verzichtet. Für dieses Projekt wurde aus den vorhandenen ein entsprechendes Sub-Set

ausgewählt.

Eine User-Story galt somit dann als fertiggestellt, wenn alle Tests, die für diese vorhanden waren auch positiv abgeschlossen werden konnten. Es wurde hierbei versucht, die User-Sicht so gut als möglich abzubilden. Hierfür wurden möglichst umfangreiche Tests herangezogen, die einen größtmöglichen Bereich der Software abdecken. Ein Systemtest wäre ein Beispiel hierfür.

Es wurde somit für jede Story eine Liste der zugehörigen Acceptance-Tests erstellt.

6 Auswertung

Das Projekt verlief über den Zeitraum von ca. einem Jahr. Da nicht durchgehend an der Entwicklung gearbeitet wurde sind die angegebenen Werte die Summe dessen, was an Arbeitsaufwand investiert wurde.

6.1 Daten

Stories	26
Tasks	83
Wochen	19
Stunden	760
Iterationen	10

Tabelle 58: Eckdaten

In 19 Wochen wurden 10 Iterationen abgewickelt, 26 User-Stories und 83 dazugehörige Tasks erledigt. Der Gesamtaufwand hierfür betrug 760 Stunden. Die letzte Iteration nahm nur eine Woche in Anspruch.

Iteration	Stories	Stunden	Velocity
1	US01, US02, US03	79	64
2	US02, US03, US04	85	75
3	US04, US05, US06, US07	85	77
4	US07, US08	85	80
5	US08, US09, US10, US11	82	77
6	US10, US12, US13	85	80
7	US10, US14, US15, US16	86	79
8	US15, US17, US18, US19, US20	88	82
9	US19, US21, US22, US23	90	80
10	US22, US24, US25, US26	42	42

Tabelle 59: Iterationen

In obiger Tabelle ist zu erkennen, dass einige User-Stories in mehreren Iterationen vorkommen. Angegeben sind hier jene, die eingeplant wurden. Stories, die nicht erledigt werden konnten, wurden natürlich in die nächste Iteration übernommen. Weiters ist anzumerken, dass zu Beginn nur Tasks im Gesamtausmaß von 79 Stunden eingerechnet wurden. Erfahrungsgemäß liegen die ersten Abschätzungen unter dem tatsächlichen Aufwand. Dies sollte sich auch in diesem Projekt wieder bewahrheiten. (zu erkennen an der Velocity von 64 nach Iteration 1)

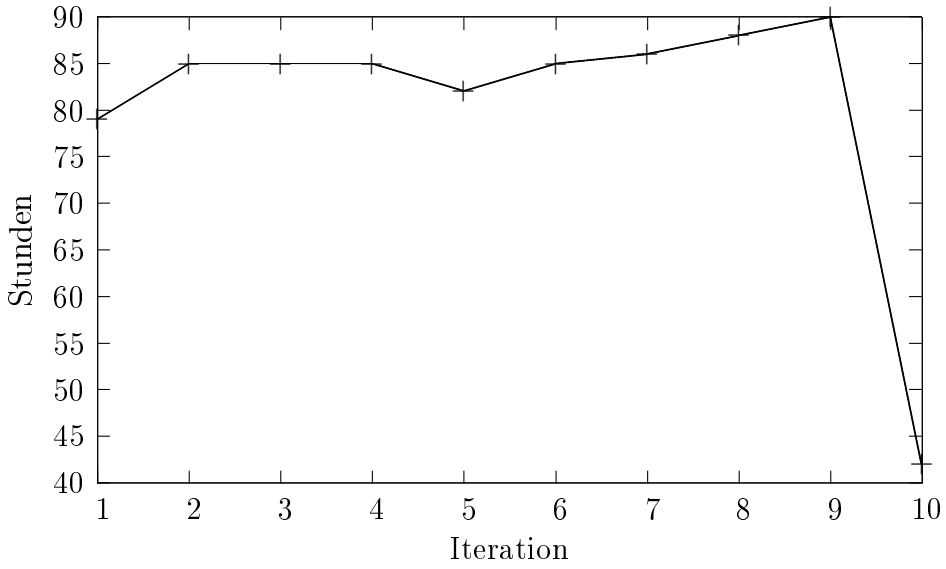

Abbildung 5: Stunden geplanter Tasks

Die geplanten Tasks starten absichtlich, wie bereits erwähnt, mit einem relativ geringen Wert. Es wurde im weiteren Verlauf versucht, die Stunden bei einem Wert von ca. 85 pro Iteration zu halten. Dies soll verhindern, dass während einer Iteration neue Stories abgeschätzt werden müssen.

Bei der letzten Iteration ergibt sich der geringe Stunden-Wert daraus, dass schlicht und einfach keine offenen Tasks/Stories in größerem Umfang mehr existierten.

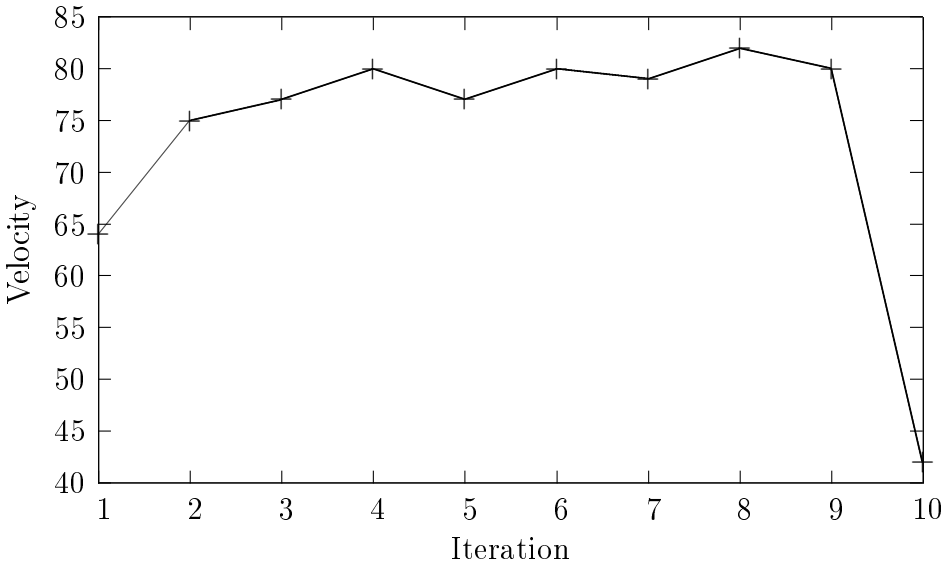

Abbildung 6: Velocity-Verlauf

An der Velocity ist gut zu erkennen, dass der Arbeitsaufwand vor allem in der Projekt-Eingangsphase überschätzt wurde. Mit der 2. Iteration gab es einen erheblichen Anstieg. Im weiteren Verlauf ist nur mehr eine leichte Steigerung zu erkennen. Naturlich ist auch hier zu Projektende nur ein sehr geringer Wert festzustellen.

Codezeilen

Sprache	Codezeilen	Anteil
C++	3437	61,8%
BASH-Scripts	881	15,9%
Test-Code	1239	22,3%

Tabelle 60: LOC-Auswertung

Bei der Auswertung der Codezeilen ist zu erkennen, dass die BASH-Scripts nur einen relativ geringen Anteil ausmachen. Diese leisten allerdings den Hauptteil der "Arbeit". Dies resultiert daraus, dass die BASH-Scripts dafür verwendet wurden um mittels einfacher Abfragen verschiedenste externe Prozesse zu starten. Komplexere Logik und die Auswertung der diversen Prozess-Outputs wurden im C++-Teil realisiert.

6.2 Erfahrungen

Im Falle der Einzelentwicklung nimmt man natürlich die Rolle des Customers und des Entwicklers ein. Es hat sich gezeigt, dass dadurch User-Stories entstehen, die weniger genau spezifiziert sind. Dies wird jedoch als weniger problematisch erachtet, da kein Wissenstransfer nötig ist, um die Implementierung entsprechend er Kundenwünsche abzuwickeln.
Aus diesem Grund wurde auch auf die Verwendung des oft gängigen Templates "Als <Rolle>, möchte ich <Ziel/Wunsch>, um <Nutzen>" [Wik11c] verzichtet.

Der Versuch, die User-Stories nicht allzu technisch zu halten ist leider nur teilweise gelungen. Es erwies sich als schwierig, beim Entwerfen der Stories nur die Kundensicht zu vertreten. Man hat sofort immer auch die technischen Aspekte und Implementierungsdetails im Hinterkopf und bringt diese automatisch ein.

Einige Tasks wurden bewusst nicht exakt definiert. Durch eine entsprechende Formulierung wurde erreicht, dass Detailentscheidungen erst später im Projektverlauf getroffen werden konnten. Diese Gegebenheit wurde genutzt um den "Customer on site" zu ersetzen. Oft sind die exakten Ansprüche nicht im

Vorhinein bekannt und ergeben sich erst im Projektverlauf.
Bei US10 wurde z.B. nicht explizit angegebenen, welches Tool zu verwenden ist.

Tasks, bei denen die Implementierungsdetails von Vorne herein feststanden, wurden natürlich auch dementsprechend formuliert.

Die Aufwände für die erste Planungsphase wurden nicht in die Entwicklungszeit eingerechnet. Aufgaben, wie das Erstellen der initialen User-Stories oder die Betriebssystemauswahl wurden bereits im Vorfeld erledigt.

Das Auseinanderhalten der verschiedenen Rollen gestaltete sich schwieriger, als anfangs gedacht. Durch die Einzelentwicklung verschwimmt die Grenze zwischen User-Anforderungen und Programmierer-Sichtweise etwas. Es ist nicht Einfach, hier eine wirkliche Trennung zu schaffen. Man muss sich im Projektverlauf immer wieder die einzelnen Rollen ins Gedächtnis rufen und versuchen, dem so gut als möglich zu entsprechen.

Refactorings bzw. Design-Änderungen erwiesen sich durch die Verwendung von TDD als wesentlich komfortabler. Nach der Entwicklung konnte durch simples Anstoßen der entsprechenden Unit-Tests sofort wieder die Funktionalität des restlichen Systems überprüft werden. Somit konnte auf einfache Art und Weise sichergestellt werden, dass eine Änderung keine Seiteneffekte aufgewiesen hat.

7 Entwickelte Software

In diesem Kapitel werden in kurzer Form die entwickelten Komponenten bzw. das Fertige System vorgestellt.

Das Testen der Software wurde, neben der Installation auf echter Hardware, auch in einer virtuellen Maschine durchgeführt. Hierfür wurde auf die Oracle-Software VirtualBox zurück gegriffen. Diese wurde in erster Linie deshalb ausgewählt, weil sie sowohl gratis, als auch in einer Open-Source-Version verfügbar ist.

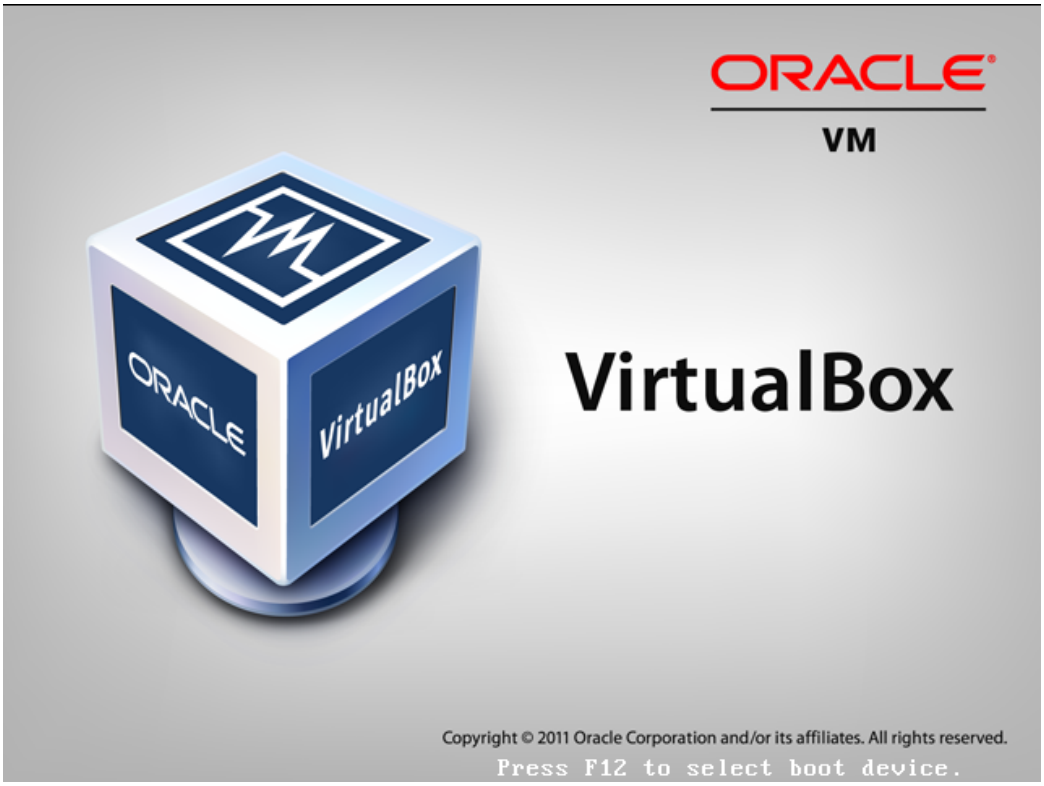

Abbildung 7: VirtualBox

Basissystem

Als Grundlage wurde eine angepasste Version der Linux-Distribution Kubuntu verwendet. Es gibt die Möglichkeit dieses von einer Live-CD zu starten und dieses auf der Festplatte zu installieren.

Abbildung 8: Bootsplash

Das Live-System besteht aus einem squashfs, welches beim Booten als unionfs mit dem RAM gemounted wird. Dadurch werden jegliche Änderungen, die nach dem Starten des Systems im Filesystem gemacht werden, in den RAM geschrieben. Diese werden nicht gespeichert und gehen bei einem Neustart verloren. Als Bootmanager wurde Grub in der Version 1 ausgewählt. Zusätzlich wurden in die Ramdisk modifizierte casper-Scripts eingebaut, welche für das korrekte Starten der Live-Umgebung zuständig sind.

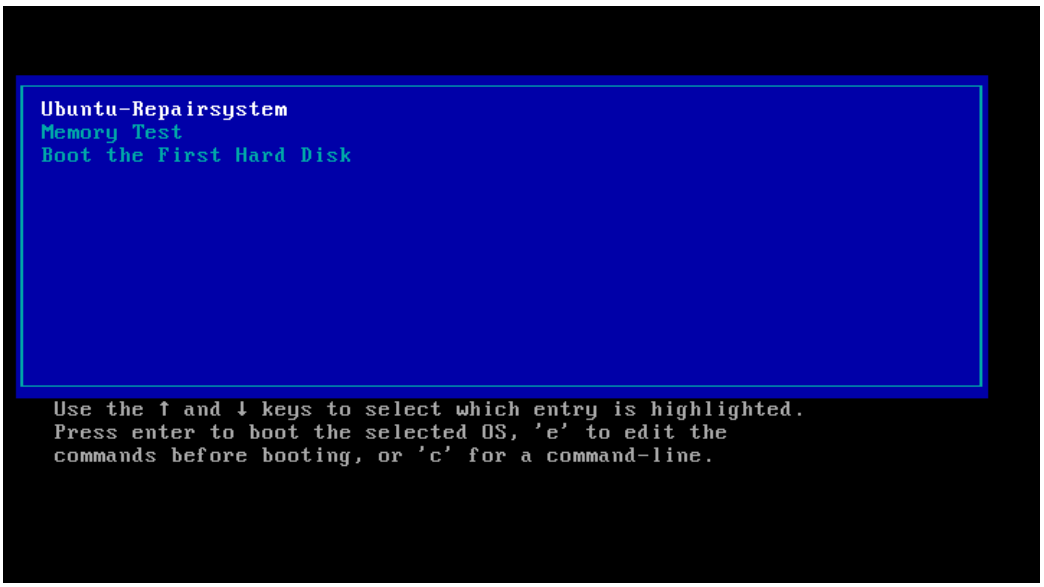

Abbildung 9: Grub Live-CD

Das selbe Prinzip wird auch bei der Festplatteninstallation verwendet. Dadurch wird diese wesentlich robuster gegen externe Ereignisse. (z.B. Netz-Stecker während dem Betrieb entfernen)

Abbildung 10: Grub HDD

Aus dem System werden dann die mit Hilfe von Qt implementierten GUI-Anwendungen gestartet. In erster Instanz ein kleines Tool, welches den Netzwerkstatus überprüft. Dieses lädt dann bei vorhandener Server-Verbindung die eigentliche Anwendung.

Die benötigte Datenbank, sowie die Scripts und die Haupt-GUI liegen auf einem Server. Als Datenbank wurde Sqlite, sowie MySQL getestet. Für den Datei-Zugriff wurde ein Samba-Share angelegt. Image-Dateien und Software-Pakete liegen ebenfalls am Samba-Share. Als Image-Format wurde aktuell ein einfaches Tar-Gzip gewählt.

Grundsätzlich wurde versucht, das System so flexibel wie möglich zu gestalten. Es ist z.B. auf einfache Weise möglich, durch anpassen eines Scripts die Art der Server-Verbindung bzw. den Image-Typ zu ändern.

Als Desktop-Oberfläche wurde ein minimaler KDE-Desktop gewählt.

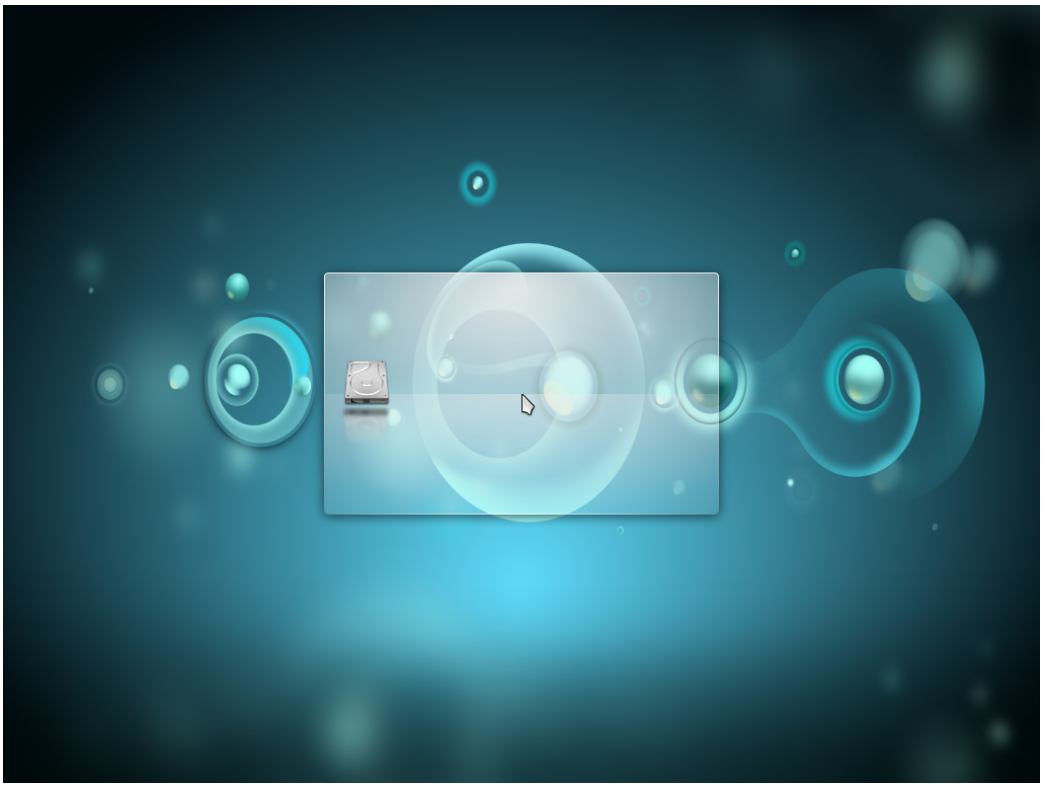

Abbildung 11: KDE

Von diesem ausgehend, wird ein Launcher-Tool gestartet, welches, wie erwähnt, die Netzwerkverbindung überprüft.

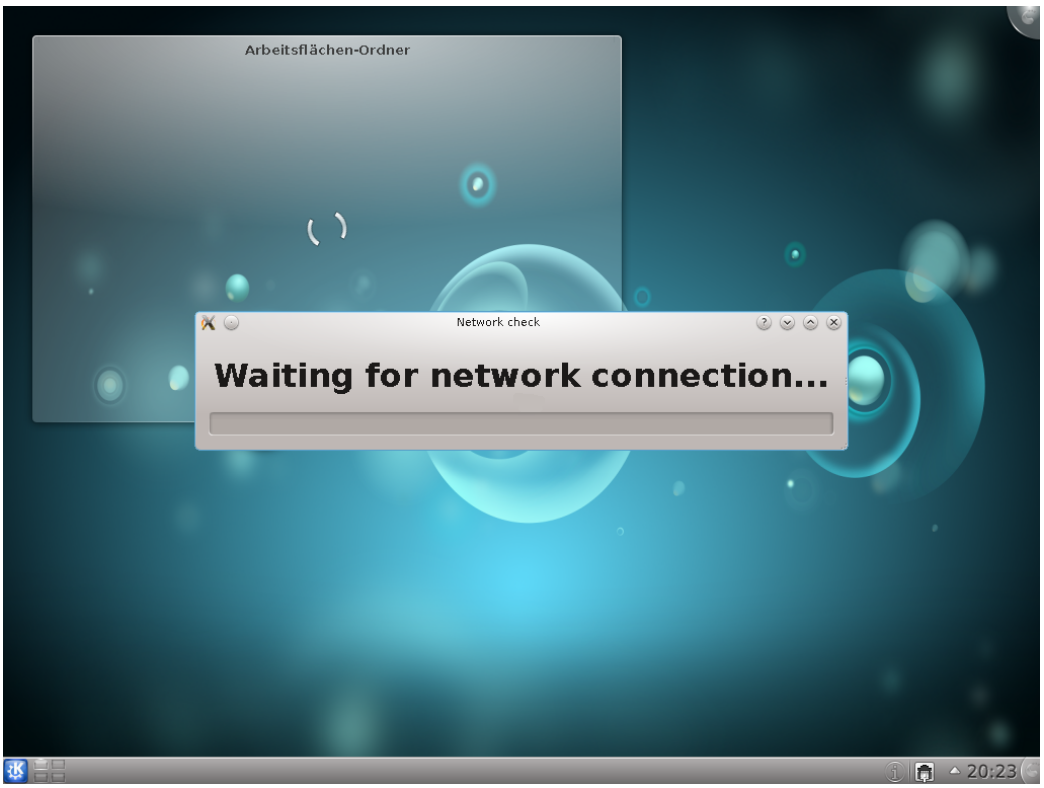

Abbildung 12: Launcher

Anschließend lädt dann bei erfolgreicher Serververbindung die eigentliche GUI.

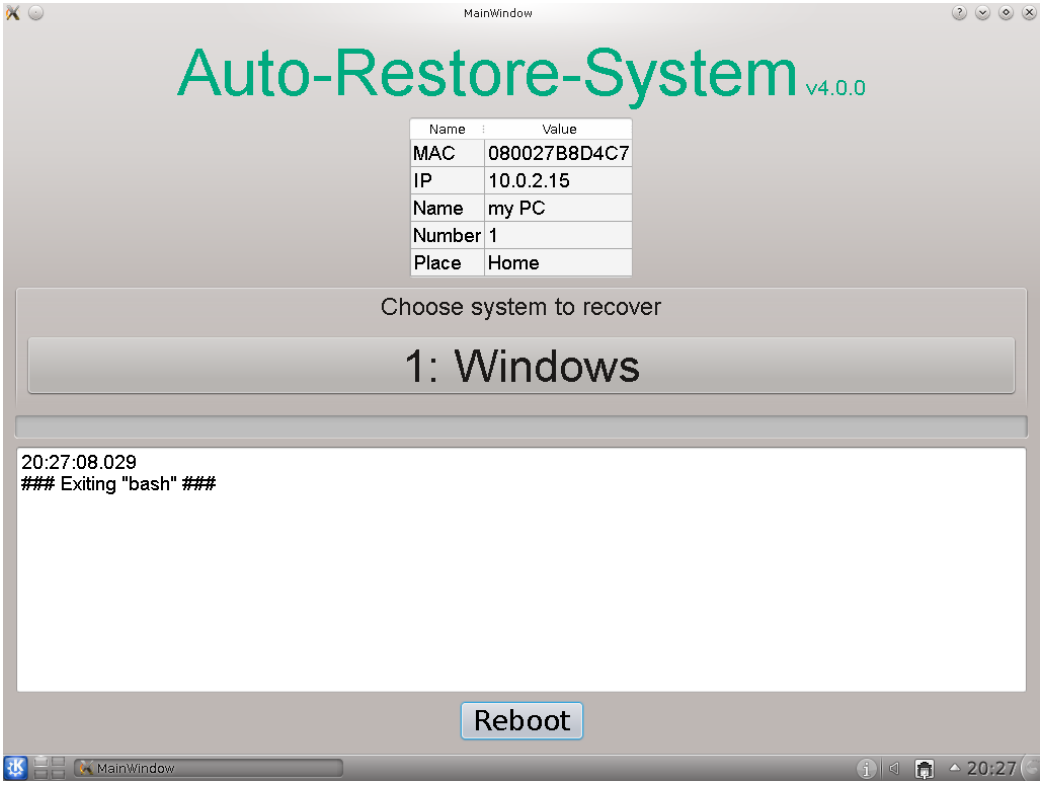

Abbildung 13: Haupt-GUI

Nun hat der Nutzer die Möglichkeit, aus der gegebenen Liste an Partitionen eine auszuwählen. Ist dies erfolgt, wird das entsprechende Image von Server oder Cache geladen und in die jeweilige Partition eingespielt. Abschließend werden noch die in der Datenbank eingetragenen Softwarepakete extrahiert.

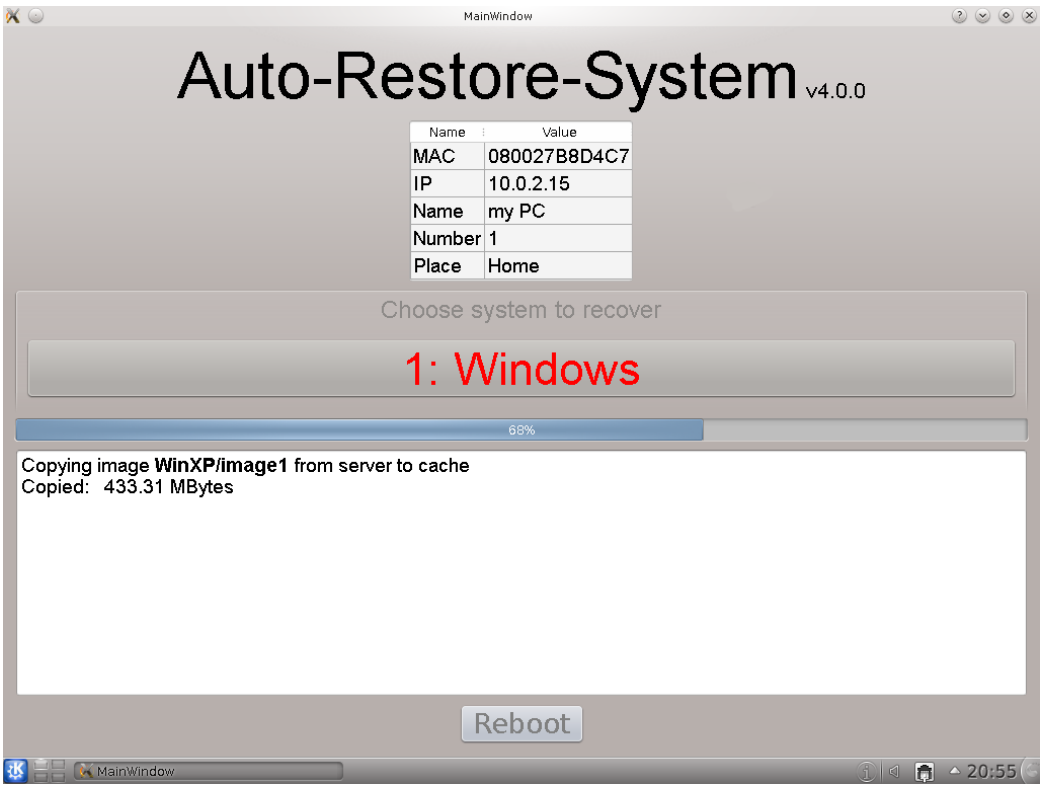

Abbildung 14: Rücksicherung

Anschließend werden vom System die benötigten Partitionsinformationen, sowie Bootmanager und eventuell boot.ini geschrieben. Ohne, dass diese korrekt sind, wäre ein Starten des Betriebssystems unmöglich.

8 Ausblick

Extreme Programming bei Einzelentwicklung hat sich als durchaus praktikabel erwiesen. Der "Overhead" durch die vorgegebenen Konzepte konnte relativ gering gehalten werden. Der Zusatzaufwand für das Test-Driven-Development ist zwar nicht unerheblich, jedoch bringt es auch einen wesentlichen Vorteil bei Codeänderungen mit sich.

XP hat dazu beigetragen, Ordnung in den oft relativ planlosen Prozess der Einzelentwicklung zu bringen. Die klare Aufteilung in Iterationen und die Vorgaben bezüglich der Planungsphase tragen wesentlich zur Verbesserung der Implementierung bei.

Alle wichtigen Aspekte von XP konnten beibehalten werden. Dies erwies sich als ein Effektiver Weg für die Einzelprogrammierung. Es ist nötig, die richtige Balance zwischen zu viel Overhead und zu wenig restriktiv zu finden. Weiters ist auch einiges an Sorgfalt nötig, um die einzelnen Personas wirklich getrennt betrachten zu können.

Im Rahmen dieser Arbeit konnte der Ansatz an einem konkreten Projekt evaluiert werden. Da durch den aktuellen Trend zu kleineren Tools bzw. "Apps" viele Ein-Mann-Projekte abgewickelt werden, wäre es interessannt, wenn diese XP-Methode aufgegriffen würde. Eine Auswertung mehrerer Projekte würde einen besseren Überblick über die Praxistauglichkeit für verschiedenste Einsatzgebiete liefern.

Der Autor wird auch bei zukünftigen Einzelprojekten wieder auf XP-Konzepte setzen.

Abkürzungsverzeichnis

CRC ..... Class, Responsibilities, Collaborators

EST ..... estimated

HDD ..... Hard disk-drive

KISS ..... Keep it simple, stupid

LOC ..... Lines of code

PRI ..... priority

PSP ..... Personal Software Process

PXP ..... Personal Extreme Programming

StP ..... Story points

TDD ..... Test-Driven-Development

US ..... User story

VM ..... virtual machine

WiP ..... Work in Progress

XP ..... Extreme Programming

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

[AU08] Ravikant Agarwal and David Umphress. Extreme programming for a single person team. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 82–87, New York, NY, USA, 2008. ACM.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, Second Edition.* Addison Wesley Professional, 2004.

[BB11] Jim Benson and Tonianne DeMaria Barry. *Personal Kanban.* Createspace, 2011.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile manifesto. `http://http://www.agilemanifesto.org`, 2001.

[BC89] Kent Beck and Ward Cunningham. A Laboratory for Teaching Object–Oriented Thinking. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 1–6, October 1989. `http://c2.com/doc/oopsla89/paper.html`.

[Beh08] Pascal Behrmann. Konzeption und evaluierung eines agilen entwicklungsprozesses für einzelentwickler am beispiel eines unterstüzungssystems für autoren. Bachelorarbeit, Hochschule für Angewandte Wissenschaften Hamburg, 2008.

[Ben11] Jim Benson. Personal kanban. `http://www.personalkanban.com`, 2011.

[BF00] Kent Beck and Martin Fowler. *Planning Extreme Programming.* Addison Wesley, 2000.

[CC11] Inc. Cunningham & Cunningham. Pair programming ping pong pattern. http://www.c2.com/cgi/wiki?PairProgrammingPingPongPattern, 2011.

[DKI09] Yani Dzhurov, Iva Krasteva, and Sylvia Ilieva. Personal extreme programming – an agile process for autonomous developers, 2009.

[EMT05] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:226–237, 2005.

[Gre11] James W. Grenning. *Test-Driven Development for Embedded C*. The Pragmatic Programmers, 2011.

[Hum96] Watts S. Humphrey. *Introduction to the Personal Software Process*. Addison Wesley Longman, 1996.

[Jef05] Ron Jeffries. Extreme programming for one. http://c2.com/cgi/wiki?ExtremeProgrammingForOne, 2005.

[Jef11] Ronald E. Jeffries. Xprogramming. http://xprogramming.com, 2011.

[Mey10] Benjamin Meyer. Qautotestgenerator. https://github.com/icefox/qautotestgenerator, 2010.

[Obj10] Blue Collar Objects. Testdrivendevelopment. http://www.bluecollarobjects.com/bin/view/Main/TestDrivenDevelopment, 2010.

[Pru11] John Pruitt. Personal scrum. http://blog.jgpruitt.com/tag/personal-scrum, 2011.

[Sri09] Ananth Srirangarajan. The scrum process for independent programmers. Master's thesis, Auburn University, 2009.

[SW08] James Shore and Shane Warden. *The Art of Agile Development*. O'Reilly Media Inc., 2008.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Harvard Business Review*, 1986.

[Wel09] Don Wells. Extreme programming: A gentle introducction. `http://www.extremeprogramming.org`, 2009.

[Wes05] Frank Westphal. *Testgetriebene Entwicklung mit JUnit & FIT*. Dpunkt Verlag, 2005.

[Wik11a] Wikipedia. Kanban in der it. `http://de.wikipedia.org/wiki/Kanban_in_der_IT`, 2011.

[Wik11b] Wikipedia. Kiss-prinzip. `http://de.wikipedia.org/wiki/KISS-Prinzip`, 2011.

[Wik11c] Wikipedia. User story. `http://de.wikipedia.org/wiki/User_Story`, 2011.

[Wik11d] Wikipedia. Value stream mapping. `http://en.wikipedia.org/wiki/Value_stream_mapping`, 2011.

[Woo09] Andrew Woodward. Doing agile in a team of one. `http://www.21apps.com/agile/doing-agile-in-a-team-of-one`, 2009.