



Hochschule für Angewandte Wissenschaften Hamburg  
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Pascal Behrmann

Konzeption und Evaluierung eines agilen  
Entwicklungsprozesses für Einzelentwickler am  
Beispiel eines Unterstützungssystems für Autoren

Pascal Behrmann

Konzeption und Evaluierung eines agilen  
Entwicklungsprozesses für Einzelentwickler am  
Beispiel eines Unterstützungssystems für Autoren

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang Angewandte Informatik  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Bettina Buth  
Zweitgutachter : Prof. Dr. rer. nat. Michael Neitzke

Abgegeben am 22. Juli 2008

**Pascal Behrmann**

**Thema der Diplomarbeit**

Konzeption und Evaluierung eines agilen Entwicklungsprozesses für Einzelentwickler am Beispiel eines Unterstützungssystems für Autoren

**Stichworte**

Agile Entwicklung, Scrum, Extreme Programming, Test First, zyklische Entwicklung, Schriftstellerei

**Kurzzusammenfassung**

SoDa, kurz für Solo Development is agile, ist eine agile Entwicklungsmethode für Einzelentwickler. Es kombiniert Methoden aus Extreme Programming und Scrum und stellt dadurch eine agile und systematische Methode zur Verfügung. Dabei bietet es Möglichkeiten für Einzelentwickler schnell auf Änderungen von Anforderungen zu reagieren und trotzdem Zielgerichtet zu arbeiten.

**Pascal Behrmann**

**Title of the paper**

Conception and evaluation of an agile development process for single developers exemplified by an authoring support system.

**Keywords**

Agile development, Scrum, Extreme Programming, test first, cyclic development, authoring

**Abstract**

SoDa, short for Solo Development is agile, is an agile development method for developers, who work alone. SoDa utilizes practices and methods of Extreme Programming and Scrum and thereby grants it's user a stable and easy-to-use development process. As an agile approach to software development it allows a people-centered, individual and responsive development, thus allowing the developer to fast reactions to change, collaboration with customers and creation of working software.

*Meiner Verlobten Sandra  
und meinen Eltern die mich wäh-  
rend der ganzen Zeit unterstützt ha-  
ben.*

## **Danksagung**

Ich danke meiner Professorin Bettina Buth, für ihre Kommentare und ihre konstruktive Kritik, meinen Eltern, für ihre Unterstützung, meinen Freunden dafür, dass sie es geschafft haben mich von der Arbeit abzulenken, wenn es Zeit dafür war und nicht zuletzt meiner Verlobten Sandra die immer hilfreich an meiner Seite stand.

# Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>Tabellenverzeichnis</b>                                       | <b>9</b>  |
| <b>Abbildungsverzeichnis</b>                                     | <b>10</b> |
| <b>1 Einführung</b>  | <b>11</b> |
| 1.1 Motivation . . . . .   | 11        |
| 1.2 Überblick . . . . .  | 12        |
| 1.3 Ziele . . . . .  | 12        |
| 1.3.1 Agiler Entwicklungsprozess . . . . .                       | 12        |
| 1.3.2 Unterstützungssoftware . . . . .                           | 13        |
| <b>2 Grundlagen</b>  | <b>14</b> |
| 2.1 Agiler Entwicklungsprozess . . . . .                         | 14        |
| 2.1.1 Vorhandene agile Einzelentwicklerprozesse . . . . .        | 15        |
| 2.1.2 Extreme Programming . . . . .                              | 15        |
| 2.1.3 Scrum . . . . .  | 19        |
| 2.1.4 Dokumente . . . . .  | 22        |
| 2.2 Testen . . . . .   | 24        |
| 2.2.1 Testgetriebene Entwicklung . . . . .                       | 24        |
| 2.2.2 Anwendungsfallbasiertes Testen . . . . .                   | 24        |
| <b>3 Analyse</b>   | <b>26</b> |
| 3.1 Extreme Programming . . . . .                                | 26        |
| 3.1.1 Praktiken . . . . .  | 26        |
| 3.2 Scrum . . . . .  | 30        |
| 3.2.1 Sprint und Sprintplanung . . . . .                         | 30        |
| 3.2.2 Dokumente . . . . .  | 31        |
| 3.2.3 Scrum Rollen . . . . .                                     | 31        |
| 3.3 Fazit . . . . .  | 32        |
| 3.3.1 Nutzbare Praktiken des Extreme Programming . . . . .       | 32        |
| 3.3.2 Nutzbare Ideen und Strukturen der Scrum Methodik . . . . . | 33        |
| <b>4 SoDa - Solo Development is agile</b>                        | <b>34</b> |

---

|          |   |           |
|----------|---|-----------|
| 4.1      | Übersicht . . . . .   | 34        |
| 4.1.1    | Ablauf der Entwicklung . . . . .                                      | 34        |
| 4.2      | Geeignete Projekttypen . . . . .                                      | 35        |
| 4.3      | Praktiken . . . . .   | 36        |
| 4.3.1    | Feingranulares Feedback . . . . .                                     | 36        |
| 4.3.2    | Andauernder Prozess . . . . .   | 36        |
| 4.3.3    | Verständlichkeit und Kommunikation . . . . .                          | 37        |
| 4.4      | Dokumente . . . . .   | 38        |
| 4.4.1    | Product backlog . . . . .   | 38        |
| 4.4.2    | Analyse backlog . . . . .   | 41        |
| 4.4.3    | Sprint backlog . . . . .  | 42        |
| 4.4.4    | Burn Down Chart . . . . .   | 42        |
| 4.4.5    | Glossar . . . . .   | 43        |
| 4.4.6    | Alternativen . . . . .  | 43        |
| 4.5      | Sprint . . . . .  | 44        |
| 4.5.1    | Sprinttypen . . . . .   | 44        |
| 4.5.2    | Sprintplanung . . . . .   | 46        |
| 4.5.3    | Tagesplanung . . . . .  | 49        |
| 4.5.4    | Sprintendeplanung . . . . .   | 50        |
| 4.5.5    | Sprintende und unfertige Anforderungen . . . . .                      | 50        |
| 4.6      | CO <sub>2</sub> Communication, Organisation and Orientation . . . . . | 51        |
| <b>5</b> | <b>Unterstützungssystem für Autoren</b>                               | <b>52</b> |
| 5.1      | SoDa in der Entwicklung . . . . .                                     | 52        |
| 5.1.1    | Erheben und Festhalten von Anforderungen . . . . .                    | 52        |
| 5.1.2    | Erfüllen und Testen von Anforderungen . . . . .                       | 53        |
| 5.2      | Die Vision . . . . .  | 55        |
| 5.2.1    | Anforderungen . . . . .   | 55        |
| 5.3      | Benutzer . . . . .  | 62        |
| 5.3.1    | Aufbau der Benutzer und Kontakt . . . . .                             | 62        |
| 5.3.2    | Umfrageergebnisse . . . . .   | 62        |
| 5.3.3    | Benutzerwünsche . . . . .   | 67        |
| 5.4      | Analyse bestehender Software . . . . .                                | 67        |
| 5.4.1    | Vorgestellte Software . . . . .                                       | 68        |
| 5.4.2    | Fazit . . . . .   | 73        |
| 5.5      | Analyse der benötigten Software . . . . .                             | 77        |
| 5.5.1    | Features . . . . .  | 77        |
| 5.5.2    | Funktionale Features . . . . .  | 77        |
| 5.5.3    | Nichtfunktionale Features . . . . .                                   | 82        |
| 5.5.4    | Notwendigkeit eines weiteren Produktes . . . . .                      | 83        |

---

|          |   |            |
|----------|---|------------|
| 5.5.5    | Das Unterstützungssystem für Autoren . . . . .      | 84         |
| <b>6</b> | <b>Evaluierung von SoDa</b>                         | <b>88</b>  |
| 6.1      | Erfahrungen . . . . .                               | 88         |
| 6.1.1    | Versuchsaufbau . . . . .                            | 88         |
| 6.1.2    | Nicht striktes SoDa . . . . .                       | 89         |
| 6.1.3    | Striktes SoDa . . . . .                             | 89         |
| 6.2      | Einsatzfähigkeit . . . . .                          | 89         |
| 6.2.1    | Einsatzfähigkeit für Studenten . . . . .            | 90         |
| 6.2.2    | Einsatzfähigkeit für Entwickler . . . . .           | 91         |
| 6.3      | Veränderungsvorschläge . . . . .                    | 91         |
| 6.3.1    | Notwendige Veränderungen . . . . .                  | 91         |
| 6.3.2    | Mögliche Veränderungen . . . . .                    | 93         |
| <b>7</b> | <b>Fazit</b>  | <b>95</b>  |
| 7.1      | SoDa aus Sicht des Autors . . . . .                 | 95         |
| 7.2      | Ausblick in die Zukunft . . . . .                   | 96         |
| 7.2.1    | Die Zukunft von Solo Development is agile . . . . . | 96         |
| 7.2.2    | Die Zukunft der Autorensoftware . . . . .           | 96         |
|          | <b>Literaturverzeichnis</b>                         | <b>97</b>  |
|          | <b>A Hilfsmittel</b>                                | <b>99</b>  |
|          | <b>Glossar</b>                                      | <b>100</b> |
|          | <b>Index</b>  | <b>102</b> |



# Tabellenverzeichnis

|     |   |    |
|-----|---|----|
| 2.1 | Einfaches Use Case Beispiel . . . . .                                       | 22 |
| 2.2 | Detailliertes Use Case Beispiel . . . . .                                   | 23 |
| 4.1 | Product backlog Beispiel . . . . .  | 40 |
| 4.2 | Analyse backlog Beispiel . . . . .  | 41 |
| 4.3 | Burn Down Chart Beispiel . . . . .  | 43 |
| 5.1 | Ein kleines Beispiel für Analyseaufgaben. . . . .                           | 54 |
| 5.2 | Funktionale Anforderungen der Vision . . . . .                              | 55 |
| 5.3 | Funktionale Anforderungen aus Konkurrenzprodukten . . . . .                 | 74 |
| 5.4 | Funktionale Anforderungen, die aus der Analysephase gefolgert werden können | 78 |
| 5.5 | Vergleich der erfüllten Anforderungen der getesteten und geplanten Produkte | 84 |

# Abbildungsverzeichnis

|     |   |    |
|-----|---|----|
| 4.1 | SoDa Logo . . . . .   | 34 |
| 4.2 | Vereinfachte Darstellung eines SoDa Projektes . . . . .   | 35 |
| 4.3 | Dokumente im SoDa Projekt. PB stellt das Product Backlog da, das aus $CO_2$ Phasen (5) und der Sprintendeplanung (4) mit Anforderungen gefüllt wird. Es wird während der Sprintanfangsplanung genutzt (6) um Anforderungen in das Sprintbacklog oder Analysebacklog, beides dargestellt durch SB, zu übertragen (3). Die Anforderungen werden dann während der Tagesplanung (2) in das Burn Down Chart, das durch BC dargestellt wird, übertragen (1) . . . . . | 39 |
| 4.4 | Vollständige Darstellung eines SoDa Projektes . . . . .   | 45 |
| 4.5 | Darstellung des mit jeder $CO_2$ Phase immer genauer werdenden Anforderungen, die am Ende zum korrekten Programm führen. . . . .  | 51 |
| 5.1 | Ergebnisse der Umfrage, zur Bedeutung der Abkürzungen siehe Kapitel 5.3.2   | 63 |

# 1 Einführung

Diese Arbeit befasst sich mit einer agilen Entwicklungsmethode für Einzelentwickler, die während des Schreibens dieser Arbeit konzipiert und beurteilt wurde. Diese Entwicklungsmethode, mit dem Namen SoDa, das für "Solo Development is agile" steht, basiert auf den beiden etablierten agilen Ansätzen des Extreme Programmings und Scrum. Dabei wurde sie so angepasst, dass sie den Bedürfnissen eines allein arbeitenden Entwicklers gerecht werden soll. Als agile Methode hat SoDa das Ziel einem Entwickler zu ermöglichen Software zu entwickeln, die die Wünsche des Kunden erfüllt und nicht nur die Verträge mit ihm. Sowie einen Prozess zur Verfügung zu stellen, der schnell auf Veränderungen reagiert und sich dem Entwickler und dem Kunden anpasst und nicht verlangt, dass diese sich anpassen müssen.

## 1.1 Motivation

Die Methoden und Konzepte der agilen Softwareentwicklung gehören heutzutage zum Repertoire eines Studenten der Hochschule für Angewandte Wissenschaften Hamburg. Es ist also auch nicht verwunderlich, dass ein solcher Student auch einen solchen Ansatz wählt, wenn sich sein Studium dem Ende nähert, um Software zu entwickeln. Doch für die Abschlussarbeit scheinen diese Entwicklungsprozesse nur bedingt geeignet zu sein, da sie auf die Arbeit von Gruppen ausgelegt sind, die meisten Arbeiten jedoch von Einzelpersonen erstellt werden, oder so bearbeitet werden müssen, dass klar erkennbar ist welcher der Prüflinge für welchen Abschnitt verantwortlich ist. Für jemanden, der die Worte des Agilen Manifests[[Beck u. a. \(2001\)](#)] während des Studiums als eine Grundlage der Softwareentwicklung akzeptiert hat, wird der Schritt zum Wasserfallmodell ein Rückschritt sein. Genau dies ist der Grund für die Konzeption eines Prozesses der es Einzelentwicklern ermöglichen soll gezielt Software zu entwickeln, mit den Prinzipien der agilen Softwareentwicklung. Auch für einen Entwickler, der alleine arbeitet, da er einzige Entwickler seiner Firma ist oder aus anderen Gründen als Einzelperson eingesetzt wird, ist der agile Ansatz oft ein fast unverzichtbares Hilfsmittel, wenn es in Bereiche geht, die ihm unbekannt sind oder sich schnell ändern, und daher eine hohe Flexibilität erfordern. Daher gibt es eine große Anzahl von verschiedenen Feldern in denen ein solcher Entwicklungsprozess verwendet werden kann und verwendet werden sollte, um Software zu erstellen die auf die Wünsche des Kunden zugeschnitten ist.

Die Beschreibung eines solchen agilen Prozesses soll während dieser Arbeit konzipiert und seine Stärken und Schwächen getestet werden.

Die überzeugendste Methode einen Prozess zu testen und eben jene Stärken und Schwächen zu finden, ist es ihn während der Anwendung zu beobachten. Erst während der Anwendung können sich wirkliche Probleme auftun und bewältigt werden. Hierfür soll ein Softwaresystem entwickelt werden, das Autoren bei der Erstellung belletristischer Literatur unterstützt.

## 1.2 Überblick

Im ersten Abschnitt dieser Arbeit werden die Grundlagen der agilen Entwicklung in Kapitel 2 beschrieben und die bereits existierenden Ansätze agiler Entwicklung für Einzelentwickler untersucht. In Kapitel 3 wird die Nutzbarkeit der Ideen und Praktiken agiler Methoden für Einzelentwickler überprüft. Aus den Erkenntnissen dieser Analyse folgt der in Kapitel 4 beschriebene Prozess. Im darauf folgenden Kapitel 5 wird der während der Evaluation betrachtete Softwarebereich dargestellt, um die Veränderungen und Entwicklungsschritte, denen der Prozess unterworfen war zu verdeutlichen. In Kapitel 6 werden die, während der Arbeit mit dem Prozess, festgestellten Vor- und Nachteile beschrieben und Verbesserungsvorschläge gegeben. Abschließend wird in Kapitel 7 ein Fazit über den Prozess, die Software und die Arbeit während der Bachelorarbeit gezogen.

## 1.3 Ziele

Da es bisher noch keinen etablierten Prozess gibt, der für Einzelentwickler Möglichkeiten der agilen Entwicklungen zur Verfügung stellt und es ihm so erlaubt auch ohne Teams in kürzester Zeit Software für ihm nur wenig bekannte Einsatzgebiete zu erstellen, ist es das Ziel dieser Arbeit einen Prozess zu entwickeln der dies unterstützt und sich leicht sowohl von erfahrenen Entwicklern als auch von unerfahrenen Anfängern oder Studenten übernehmen lässt.

### 1.3.1 Agiler Entwicklungsprozess

Der agile Prozess zur Softwareentwicklung für Einzelentwickler basiert auf zwei etablierten agilen Modellen. Er soll sowohl viele der praktischen Methoden des Extreme Programming, als auch die Projektplanungstechniken von Scrum in sich vereinen, um dadurch ein agiles,

einfaches und gleichzeitig sicheres Erstellen von Software zu ermöglichen. Die Anpassungen, die für die Einzelpersonenentwicklung und die Zusammenarbeit dieser beiden Ansätze notwendig waren, werden im Kapitel 3 beschrieben.

### 1.3.2 Unterstützungssoftware

Das Unterstützungssystem für Autoren soll Menschen helfen literarische Texte, wie Romane oder Kurzgeschichten, zu erstellen. Hierzu soll es den Autoren Hilfsmittel zur Verfügung stellen. Diese Hilfsmittel könnten Möglichkeiten zur Hinzufügung von Notizen, Informationen über wichtige Personen oder aber auch ausgefallene Funktionen wie Unterstützung bei der Neuordnung von Kapiteln umfassen. Die eigentliche Funktionalität soll hierbei von Autoren bestimmt werden die in verschiedenen Internetgemeinden befragt werden. Die Vision des Autors für dieses Produkt umfassen Funktionen, die es einem Autoren ermöglichen Informationen über Charaktere, Szenen, Objekte und Orte zu speichern und auszuwerten. Zum Beispiel die Darstellung der zeitlichen Abläufe der Szenen, die in anderer Reihenfolge auftreten können als die Szenen in der Geschichte selbst auftreten. So wird es dem Autor erlaubt schneller eine Übersicht über den Ablauf seiner Geschichte zu erlangen.

## 2 Grundlagen

In diesem Kapitel werden die Grundlagen, die nötig waren um den agilen Prozess für Einzelentwickler zu erstellen, erklärt. Ein kurzer Überblick über die Ideen der agilen Entwicklung und des Testens wird gegeben, um dem geeigneten Leser den Einstieg in das Thema zu erleichtern.

### 2.1 Agiler Entwicklungsprozess

Agile Softwareentwicklung ermöglicht es, sich schnell auf neue oder veränderte Anforderungen eines Projektes anzupassen und dadurch Software so zu bauen wie sie vom Kunden benötigt wird. Um dieses hochgesteckte Ziel zu erfüllen verwenden die Modelle der agilen Entwicklungen eine Vielzahl von Methoden. Allen Modellen gemein ist jedoch, dass sie darauf basieren in kurzen Zyklen zu arbeiten, um so eine schnelle Anpassung des Projektes an neue, oder neu erkannte, Anforderungen zu gewährleisten. Jeder Zyklus in einem Entwicklungsprozess stellt dabei ein für sich stehendes Softwareprojekt da, in dem Anforderungen analysiert, bewertet und erfüllt werden müssen, in dem der entstehende Code getestet wird und in dem eine Dokumentation der erstellten Software entsteht. Jeder dieser Zyklen erzeugt eine Software die ein wenig mehr Funktionalität hat, als das Ergebnis des vorherigen Zyklus, und gemäß den Anforderungen getestet wurde. Die agilen Methoden basieren auf dem, bereits zuvor erwähnten, agilen Manifests, dessen Grundlage es war Entwicklungsprozesse personenzentrierter zu gestalten. In diesen Prozessen stehen Personen über den Prozessen, da sowohl die Entwickler, als auch die Benutzer, Experten auf ihren jeweiligen Gebieten sind und ein Projekt sich anpassen können muss, um zu gewährleisten, dass Software erstellt wird die den Kunden zufrieden stellt und nicht nur Software die den Prozess zufrieden stellt. Da Softwareentwicklung normalerweise in Teams vollführt wird, sind die Prozesse der agilen Softwareentwicklung diese diese Form erdacht worden. Viele der Prinzipien und Methoden sind speziell darauf ausgelegt die Kommunikation eines Teams zu verbessern oder benötigen mehr als eine Person um durchgeführt zu werden. Für einige dieser Techniken müssen Substitute gefunden werden, andere Techniken können angepasst werden und einige von ihnen können weggelassen werden, da die Probleme die sie beheben sollen erst durch Gruppen auftauchen.

### 2.1.1 Vorhandene agile Einzelentwicklerprozesse

Da es Einzelpersonen gibt die Software entwickeln und agile Softwareentwicklung eine anerkannte Methode ist, erscheint es wahrscheinlich, dass es bereits Ansätze gibt die beides zusammenzubringen. Als der Autor vor dem Beginn seiner Bachelorarbeit nach solchen Ansätzen suchte, um sie im Zuge einer solchen Arbeit zu verwenden, fand er jedoch keine Methode die speziell für den Einsatz von Einzelentwicklern konzipiert wurde. Zwar gab es mehrere Diskussionen und Artikel, die besagten, dass es möglich sei Software als Einzelperson mit agilen Entwicklungsmethoden zu entwickeln, jedoch fehlte es den Artikeln an tiefgehenden Analysen und vollständig an Evaluationen. In allen, dem Autor bekannten, Fällen basieren diese Artikel darauf, dass aus einer etablierten Methode die Praktiken und Ideen gestrichen werden, die als Einzelperson nicht, oder nur schwer, durchführbar sind, ohne über die dadurch folgenden Konsequenzen nachzudenken oder ihnen entgegen zu wirken.

#### 2.1.1.1 Solo Scrum

Der einzige weitere Ansatz der öfter Erwähnung findet, ist eine Adaption von Scrum mit dem Namen Solo Scrum. Fast alle Quellen zu diesem Thema scheinen sich jedoch auf einen Wikipedia Artikel[Wikipedia (2008)] zu beziehen, der wenig über die Art der Anpassung erwähnt. Die einzige weitere ernst zu nehmende Quelle ist ein Artikel von Peter Bell[Bell (2007)], in dem er, laut eigener Aussage, ohne Erfahrungen mit Scrum zu haben, einen Vorschlag beschrieb wie es Möglich wäre Scrum als Einzelperson zu nutzen. Somit ist dies ebenfalls kein fertiger Entwicklungsprozess und damit keine echte Alternative zu einem getesteten Entwicklungsprozess. Solo Scrum basiert sehr stark auf Scrum, und wurde für die Bedürfnisse eines Soloentwicklers angepasst. Um schnellere Rückmeldungen über den Fortschritt des Projektes zu bekommen, sollen die Aufgaben 1-3 Stunden umfassen und Schätzungen der Dauer in Mannstunden getätigt werden. Die Zeit eines Sprints ist auf eine Woche festgelegt, da dass Projekt an dem der Autor des Artikels arbeitete ein nur eine Woche dauerndes Projekt hatte. Da er davon ausgeht, dass die Arbeitszeit stark unter eingeschränkten Ressourcen und vielen Ablenkungen leidet macht er tägliche und über den bisherigen Sprintverlauf kumulative Checks der erledigten Arbeit gegen die anfänglichen Schätzungen, um so zu erfahren ob und wie weit er hinter dem Zeitplan liegt, um so schnellstmöglich gegenzulenken.

### 2.1.2 Extreme Programming

Extreme Programming ist eine Sammlung von Praktiken für das agile Entwickeln von Software, und ist während des Chrysler Comprehensive Compensation System (C3) Projektes

durch Kent Beck entstanden. Extreme Programming erlaubt ein hohes Maß an Flexibilität in Bezug auf die Anforderungen und vertritt den Standpunkt, dass die Möglichkeit von sich ständig ändernden Anforderungen eher der Realität entsprechen, als die Möglichkeit, dass man zu Beginn eines Projektes alle Anforderungen verstehen und modellieren könnte.

### 2.1.2.1 Praktiken

Extreme Programming gibt keinen festen Entwicklungsprozess vor, sondern beschreibt eine Reihe von Praktiken, die zusammen einen agilen Prozess ergeben. Das Zusammenspiel der einzelnen Praktiken ist darauf ausgelegt eine kundennahe Entwicklung zu fördern, die es den Entwicklern erlaubt ihr Projekt, und damit auch die entstehende Software, schnell an existierende und neu aufkommende Bedürfnisse anzupassen und zu jedem Zeitpunkt eine funktionsfähige und getestete Software zur Verfügung zu stellen.

**Schnelles Feedback** Da eine der Grundlagen der agilen Entwicklung die Erkenntnis ist, dass viele Softwareprojekte daran scheitern, dass die Anforderungen an das Produkt nicht richtig erkannt wurden, verfügt Extreme Programming über eine Reihe von Praktiken, die darauf abzielen die Software und die Anforderungen besser zu verstehen, überblicken und erfüllen zu können.

**Pair Programming** Pair Programming bedeutet, dass jeder Code von Teams aus zwei Programmierern erzeugt wird, die zusammen an einer Arbeitsstation arbeiten. Dadurch kann einer der beiden detailliert das Problem lösen, während der andere das Gesamtbild im Blick hat, und den erzeugten Code einem beständigen Review unterzieht. Pair Programming folgt zwei Zielen, die Qualität des Codes zu erhöhen und, durch regelmäßiges tauschen der Paa-re, eine Übersicht über die Gesamtaufgabe an alle Programmierer zu geben.

**Planning Game** Der grundlegende Planungsprozess findet zum Beginn jeder Iteration statt und basiert auf einer engen Zusammenarbeit mit den Benutzern und User Storys. Innerhalb dieses Prozesses wird sowohl festgelegt, welche Anforderungen an das Produkt gestellt wird, als auch welche Aufgaben erledigt werden müssen.

**Test Driven Development** Innerhalb der testgetriebene Entwicklung werden Unit Tests geschrieben bevor der Code des eigentlichen Programms geschrieben wird, hierdurch soll der Programmierer alle Möglichkeiten bedenken bei denen der Code einen Fehler erzeugen könnte und ein Produkt entstehen, das die Anforderungen erfüllt und nicht ein Test, der das Programm erfüllt, nicht jedoch die Anforderungen.



**Whole Team** Beim Extreme Programming wird eine große Nähe zum Kunden gefordert, dies soll dadurch gewährleistet werden, dass der Kunde immer Mitglied des Teams ist. Wenn, zum Beispiel, eine Software für die Steuererklärungen geschrieben wird, sollte ein Experte für Steuerrecht Mitglied des Teams sein, um Fragen beantworten zu können.

**Kontinuierlicher Prozess** Die Entwicklung von Software wird innerhalb des Extreme Programmings als fortlaufender Prozess angesehen. Dadurch, dass die Anforderungen an eine Software niemals vollständig verstanden werden können und sich im Laufe der Zeit immer wieder ändern, ist auch die Entwicklung einer Software niemals vollständig abgeschlossen. Um den Prozess der Software Entwicklung diesem Weltbild anzupassen, schlägt Extreme Programming mehrere Praktiken vor.

**Continuous Integration** Extreme Programming fordert, dass die Entwickler immer mit den aktuellen Versionen des Quellcodes arbeiten sollten, so dass bei der Integration Probleme vermieden werden können. Ein Entwickler sollte deshalb seinen Code den anderen Entwicklern zur Verfügung stellen, sobald er eine Teilaufgabe erledigt hat. Weiterhin soll jeder Quellcode bei der Integration automatisch getestet werden und das Produkt mit den Änderungen neu erstellt werden.

**Refactoring oder Design Improvement** Da Extreme Programming fordert, dass der Code nur mit den jetzigen Anforderungen im Blick erstellt werden soll, nicht mit den möglichen Anforderungen der Zukunft, da sich die Anforderungen beständig ändern können und es dadurch notwendig werden kann das Design der Software zu ändern. Extreme Programming sieht diesen Prozess als ein kontinuierliches Vorgehen an, der ein notwendiger Teilschritt zur Erfüllung anderer Ziele ist, und somit immer durchgeführt werden soll, wenn Probleme oder Symptome schlechten Designs auftauchen oder Code generalisiert werden kann.

**Small Releases** Wie bereits angedeutet versucht Extreme Programming immer ein lauffähiges Produkt zu liefern, und fordert deshalb, dass die Software in kleine Releases aufgeteilt wird, die alle lauffähig sein sollen ohne von erst in der Zukunft zu erstellende Releases abhängig zu sein. Durch diese Release soll der Kunde Vertrauen in das Produkt gewinnen und möglichst schnell erkennen welche Änderungen an ihm Vorgenommen werden müssen um die Software nutzen zu können.

**Shared Understanding** In der agilen Softwareentwicklung wird fehlende Kommunikation zwischen Teammitgliedern und das Fehlen der Übersicht über das komplette Projekt oft als ein weiterer Grund für das Scheitern von Softwareprojekten angesehen. Um dieses Problem zu beheben wurden mehrere Praktiken eingeführt, die es erlauben sollen dem Team eine Übersicht über das komplette Projekt zu gewinnen.

**Coding Standards** Coding Standards sind eine Menge von Regeln über den Stil und die Formatierung des Quellcodes auf die sich die Entwickler geeinigt haben, dieser Standard kann entweder vom Hersteller der genutzten Programmiersprache stammen oder Speziell für das Produkt oder die Firma gelten.

**Collective Code Ownership** Collective Code Ownership ist die Forderung, dass alle Entwickler für den vollständige Quellcode des Produktes verantwortlich sind, und Fehler berichtigen sollten, wenn sie welche finden.

**Simple Design** Ähnlich wie in den Naturwissenschaften gilt auch beim Extreme Programming der Ansatz von Ockhams Rasiermesser, grob gesprochen: "Von zwei Möglichkeiten, wähle immer die Einfachere". Es sollte immer der einfachste Code gewählt werden, der ein Problem erfüllt, dadurch ist es einfacher einen bestehenden Quellcode zu verstehen und neue Funktionalität hinzuzufügen.

**System Metaphor** Die Systemmetapher sagt aus, dass die Namen der Klassen und Methoden so gewählt sein sollen, dass es für alle Mitglieder des Teams ersichtlich sein muss welche Aufgabe sie hat. Zum Beispiel eine Klasse Rechtschreibprüfung sollte also die Rechtschreibprüfung übernehmen, und mit einer Methode prüfeWort(Wort) ein einzelnes Wort auf korrekte Rechtschreibung prüfen.

**Wohlergehen der Programmierer** Als einen letzten Punkt für das scheitern von Softwareprojekten sieht Extreme Programming, dass Entwickler nach zu langer Arbeitszeit unkonzentriert werden und dies die Fehlerrate so stark erhöht, dass die Berichtigung der gemachten Fehler mehr Zeit verschlingt als durch die zusätzliche Arbeitszeit gewonnen wird.

**Sustainable Pace** Programmierer sollten nicht mehr als 40 Stunden Wöchentlich arbeiten, falls in einer Woche Überstunden gemacht werden, sollte dies in der darauf folgenden Woche nicht wiederholt werden. Extreme Programming geht davon aus, dass Entwickler am Produktivsten sind wenn sie ausgeruht sind und dass Überstunden dazu führen dass ihre Produktivität und die Qualität des Produktes sinken. Weiterhin sagt Extreme Programming, dass häufige Überstunden auf ein tiefergreifendes Problem im Projekt hindeuten, das nicht durch Überstunden gelöst werden kann.

### 2.1.2.2 Fazit

Die meisten Praktiken des Extreme Programming lassen sich ohne Probleme umsetzen oder anpassen. Nur die Praktik des Pair Programming lässt sich nicht umsetzen. Dadurch sollte man versuchen andere Praktiken einzuführen die das Ziel des Pair Programming unterstützen, also die Qualität des Codes erhöhen. Außerdem ist es für einen Einzelentwickler noch wichtiger als für ein Team sehr Diszipliniert im Umgang mit den Praktiken zu sein, da sie nur durch ihn Aufrecht erhalten werden und es keine weitere Kontrollinstanz gibt.

## 2.1.3 Scrum

Scrum ist ein Projektmanagementform für agile Softwareprojekte, es gibt keine Praktiken vor wie Software entwickelt werden soll, sondern beschreibt wie die Entwicklung geplant werden sollte und überlässt die Regeln der Entwicklung den Entwicklern selbst. Deshalb lässt es sich mit wenigen Veränderungen zusammen mit Extreme Programming vereinen, die über die Aufgabe des Planens nur wenig vorgeben.

### 2.1.3.1 Sprint und Scrumtreffen

Da Scrum ein agiler Prozess ist, geht auch dieser Ansatz von der Annahme aus, dass es nicht möglich ist die Anforderungen eine Software zu einem beliebigen Zeitpunkt zu überblicken. Um den Entwicklern nicht beständig wechselnden Anforderungen auszusetzen gibt es jedoch innerhalb des Scrum Prozesses Sprints. In ihnen wird an, zu Beginn des Sprints definierten, Anforderungen gearbeitet. Um die Probleme der Kommunikation und der fehlenden Übersicht zu beheben sieht Scrum tägliche Treffen vor, in denen das Projekt und der Fortschritt besprochen wird.

**Scrumtreffen** Ein Scrum oder Scrumtreffen ist ein tägliches Meeting des Teams, in dem besprochen wird was seit dem letzten Treffen erreicht wurde, was am heutigen Tag voraussichtlich erreicht wird und ob und welche Probleme, die das Erreichen eines Zieles verhindern könnten, es gibt. Ein solches Treffen ist normalerweise mit einer Zeitbeschränkung von 15 Minuten versehen.

**Sprintanfangstreffen** Zu Beginn jedes Sprints wird ein Treffen angesetzt, bei dem die Ziele des Sprints geklärt werden, und die einzelnen Aufgaben für diesen Sprint aufgeteilt werden. Hierbei erklärt der Product Owner welche Ziele er erfüllen möchte, und die Entwickler entscheiden wieviele dieser Ziele sie innerhalb des Sprints erreichen können.

**Sprint** Während eines Sprints konzentrieren sich das Team darauf die Ziele des so genannten Sprint Backlogs, einer Liste von Anforderungen, zu erfüllen. Während der Zeit des Sprints ist es niemandem möglich die Ziele zu ändern, dadurch sollen sich die Entwickler mit einem bestimmten Problem beschäftigen können, ohne durch Veränderungen oder Diskussionen abgelenkt zu sein. Normalerweise dauert ein Sprint 15-30 Tage.

**Sprintende Treffen** Auch Retrospektive genannt, soll dieses Treffen es dem Team ermöglichen Probleme und Verbesserungen anzusprechen um den Entwicklungsprozess einem beständigen Refactoring zu unterziehen.

### 2.1.3.2 Dokumente

Scrum verfügt über drei wichtige Dokumente die zum strukturierten Festhalten von wichtigen Informationen, wie Anforderungen und dem Projektfortschritt und ihrer Veröffentlichung dient.

**Product backlog** Das Product Backlog enthält vage Beschreibungen aller Anforderungen und gewünschten Features des Produktes in Form von User Storys. Es beschreibt was gebaut werden soll, und ist von jedem veränderbar. Es enthält grobe Schätzungen der Bearbeitungsdauer, um es dem Product Owner zu erleichtern Prioritäten zu setzen.

**Sprint backlog** Das Sprint Backlog besitzt einen höheren Detailgrad als das Product Backlog, es enthält Informationen darüber wie die Anforderungen sind, wie sie umgesetzt werden sollten und wie sie zu testen sind. Aufgaben sollten so aufgeteilt werden, dass für keine Aufgabe mehr als 16 Stunden Arbeitszeit geschätzt wird. Diese Aufgaben werden nicht verteilt, sondern Teammitglieder suchen sich Aufgaben aus, die sie erledigen wollen.

**Burn Down** Das Burn Down Chart zeigt die bisher erledigten und noch zu erledigenden Aufgaben dieses Sprints, und visualisiert so dem Team den bisherigen Fortschritt.

### 2.1.3.3 Scrum Rollen

Scrum kennt mehrere Rollen die in zwei Gruppen eingeteilt werden können: Pigs und Chicksen, die Namen stammen aus einem Witz, der den Unterschied zwischen den Beiden klarstellt.

A pig and a chicken are walking down a road. The Chicken looks at the pig and says "Hey, why don't we open a restaurant?" The pig looks back at the chicken and says "Good idea, what do you want to call it?" The chicken thinks about it and says "Why don't we call it 'Ham and Eggs'?" "I don't think so" says the pig, "I'd be committed but you'd only be involved"

Pigs sind also ein Teil des Projektes, sie tragen häufig und regelmäßig dazu bei, während Chickens Personen sind, die an der Software interessiert sind, aber nicht ein Teil des Teams, und daher, falls das Projekt fehlschlägt, nicht in der *Schusslinie*.

**"Pig"Roles** Pigs sind die Personen die ein Teil des Entwicklungsteams sind. Der **Product Owner** repräsentiert den Kunden im Team, er schreibt die User Storys, priorisiert sie und fügt sie in das Product Backlog ein. Der **ScrumMaster** hat die Aufgabe Hindernisse für das Team aus dem Weg zu räumen, um es dem Team zu ermöglichen das Ziel des Sprints zu erreichen. Er wacht auch über die Einhaltung der Sprint Regeln und Scrum Praktiken. Das **Team** sind die Leute die die eigentliche Entwicklung der Software übernehmen.

**"Chicken"Roles** Chicksen sind kein Teil des Entwicklungsteams, müssen jedoch auch beachtet werden. Einer der wichtigsten Grundsätze des agilen Ansatzes ist es schnell auf Anpassungen der Anforderungen zu reagieren und den Benutzer der Software mit in den Entwicklungsprozess einzubinden. Um schnell zu Reagieren kann es notwendig sein Benutzer, Experten für bestimmte Technologien oder Anforderungen oder andere Personen kurzzeitig in das Team zu holen. Auch hier werden wieder drei Gruppen unterschieden. Die **Benutzer**

sind die Personen, die das Produkt nutzen werden und können daher eine andere Gruppe sein, als der Kunde, der das Produkt in Auftrag gegeben hat. **Experten** die das Team beraten oder eine spezielle Aufgabe erledigen, aber nicht in jedem Sprint gebraucht werden. Die letzte Gruppe sind die **Stakeholder**, die jede andere Person darstellt die Interesse an der Fertigstellung des Produktes haben, z.B. die Geschäftsführung.

#### 2.1.3.4 Fazit

Scrum ist ein Planungsprozess für Teams, und daher sehr stark darauf ausgelegt, dass Teams zusammenarbeiten. Viele Maßnahmen von Scrum müssen daher angepasst werden, so müssen Wege gefunden werden, um die Treffen alleine effektiv durchzuführen. Außerdem muss die Trennung, der verschiedenen Rollen die ein Entwickler annehmen müssen, sehr streng sein.

### 2.1.4 Dokumente

#### 2.1.4.1 Use Case

Ein Use Case, oder Anwendungsfall, ist eine Methode das erwartete äußere Verhalten von Softwaresystemen zu beschreiben. Damit sind Use Cases dazu geeignet die funktionalen Anforderungen eines System zu definieren und die grobe Beschreibung eines Black Box Tests liefert. Im Zuge dieser Bachelorarbeit wird zwischen verschiedenen Detaillierungsgraden unterschieden. Einfache Use Cases beschreiben das grobe Verhalten einer ganzen Funktionsklasse mit mehreren logisch zusammenhängenden Einzelfunktionen, aus ihnen werden detailliertere Use Cases erstellt um einzelne Funktionen zu beschreiben.

Tabelle 2.1: Ein Beispiel eines kurzen Use Cases

|                     |  |
|---------------------|--|
| <b>Use Case</b>     | Use Case 4.0 Notizfunktion   |
| <b>Beschreibung</b> | Eine einfache Notizfunktion.   |
| <b>Akteure</b>      | Benutzer   |
| <b>Annahmen</b>     | Die Software ist aktiv und kein Einstellungsdialog ist geöffnet.   |
| <b>Schritte</b>     | 1. Benutzer will eine Notiz erstellen und aktiviert die Funktion über eine einstellbare Tastenkombination. Ein Schreibfenster öffnet sich und der Fokus wird auf das Fenster gelegt. |

Fortsetzung auf nächster Seite...

Tabelle 2.1 – Fortgesetzt

|  |   |
|--|---|
|  | 2. Der Benutzer schreibt die Notiz und schließt das Fenster die Notiz wird gespeichert.   |
| <b>Variation</b>                       | 1a. Benutzer will eine geschriebene Notiz öffnen und öffnet ein Kontextmenü mit den ersten Worten jeder geschriebenen Notiz über eine einstellbare Tastenkombination. |
|  | 2a. Der Benutzer wählt eine Notiz aus und die Notiz öffnet sich in einem eigenen Fenster.   |
|  | 3a. Der Benutzer ließt die Notiz und schließt das Fenster wieder  |
|  | 3a/b. Der Benutzer verändert die Notiz und schließt das Fenster die Notiz wird gespeichert  |
| <b>Nicht Funktionale Anforderungen</b> |   |
| <b>Offene Punkte</b>                   | Wie löscht man eine Notiz?  |

Tabelle 2.2: Ein Beispiel eines detaillierten Use Cases

|                     |   |
|---------------------|---|
| <b>Use Case</b>     | Use Case 4.0a Notiz schreiben   |
| <b>Beschreibung</b> | Schreiben einer Notiz   |
| <b>Akteure</b>      | Benutzer  |
| <b>Annahmen</b>     | Die Software ist aktiv und kein Einstellungsdialog ist geöffnet.  |
| <b>Schritte</b>     | 1. Benutzer will eine Notiz erstellen und aktiviert die Funktion über eine einstellbare Tastenkombination, Standard ist Steuerung + Alt + N. Ein Fenster mit einem Textfeld wird geöffnet und in den Fokus geholt |

Fortsetzung auf nächster Seite. . .

Tabelle 2.2 – Fortgesetzt

|  |  |
|--|--|
|  | 2. Der Benutzer schreibt die Notiz und schließt das Fenster die Notiz wird gespeichert.                                |
| <b>Variation</b>                       | 2a. Der Benutzer schreibt keine Notiz, lässt das Textfeld leer und schließt das Fenster, keine Notiz wird gespeichert. |
| <b>Nicht Funktionale Anforderungen</b> |  |
| <b>Offene Punkte</b>                   |  |

## 2.2 Testen

Das Testen von Software ist ein essentieller Bestandteil der Softwareentwicklung, durch den die Qualität der Software sichergestellt werden soll. Innerhalb dieses Prozesses wird die Software ausgeführt und ihr reales Verhalten mit dem erwarteten Verhalten abgeglichen.

### 2.2.1 Testgetriebene Entwicklung

Die agile Entwicklung fordert einen frühen Beginn des Testprozesses, deshalb nutzen viele agile Entwicklungsmethoden einen Ansatz in dem zuerst die Tests geschrieben werden, die das Erfüllen der definierten Anforderungen überprüfen sollen. Darauf folgend wird die Funktion geschrieben, die diese Tests erfüllt. Oft wird hier, wie in der ganzen agilen Entwicklung, iterativ gearbeitet, zuerst wird ein Test geschrieben, der nur die minimale Funktionalität überprüft und erfüllt, dann wird etwas mehr Funktionalität geprüft und so fort. Eine Funktion gilt so als erfüllt, sobald die zu ihr gehörenden Tests erfüllt sind.

### 2.2.2 Anwendungsfallbasiertes Testen

Da Softwaretesten sicherstellen soll, dass Software gewisse Anforderungen erfüllt und Anforderungen in Anwendungsfällen niedergeschrieben werden ist es eine logische Folgerung



auf ihnen basierend Tests zu erzeugen. Anwendungsfallbasierte Test prüfen die Erfüllung eines erwarteten äußeren Verhaltens und trifft keine Aussagen über den inneren Aufbau einer Software

# 3 Analyse

Im folgenden Kapitel werden die beiden Entwicklungsprozesse, die als Grundlage für den Einzelentwicklungsprozess dienen sollen, unter den Gesichtspunkten einer Entwicklung durch Einzelpersonen betrachtet. Gleichzeitig wird der Entscheidungsprozess zur Übernahme und Veränderung von Teilen dieser Prozesse in SoDa beschrieben.

## 3.1 Extreme Programming

Extreme Programming wurde für kleine Gruppen entworfen, die in wechselnden Zweierpaarungen arbeiten, dadurch erscheint diese Form der Softwareentwicklung auf den ersten Blick als unvereinbar mit der Soloentwicklung. Jedoch sind viele der Praktiken auch für Einzelentwickler einsetzbar und hilfreich.

### 3.1.1 Praktiken

Extreme Programming empfiehlt eine Reihe von Praktiken, die es einem kleinen Team aus Entwicklern ermöglichen soll effektiv und agil Software zu entwickeln und dabei die größten Probleme zu umgehen.

#### 3.1.1.1 Pair Programmings

Als eine Einzelperson ist es natürlich nicht möglich eine Praktik durchzuführen, die es erfordert zwei Personen an einem PC arbeiten zu lassen. Dadurch, dass diese Praktik nicht erfüllt werden kann, sollte ein besonderes Augenmerk auf die Erfüllung der Ziele gelegt werden, die durch diese Praktik erreicht werden sollten. Die Ziele sind es Qualität des Codes und eine projektweite Übersicht des Entwicklers sicherzustellen. Ein Einzelentwickler kann schnell die Übersicht über das Projekt verlieren, wenn er an einer spezifischen Funktion schreibt, ohne einen Partner kann hierdurch die Qualität des Codes leiden, da der Programmierende zu sehr auf seine Aufgabe konzentriert ist um Refaktorisierungsmöglichkeiten zu sehen und

Programmierfehler können sich ohne ein zweites wachsames Augenpaar leichter einschleichen.

### 3.1.1.2 Planning Game

Das Planning Game basiert im Extreme Programming auf dem Konsens erfahrener Entwickler, um so extremwertigen Einzelmeinungen auszugleichen. Eine solche Planung ist natürlich auch für Einzelentwickler unerlässlich, es muss jedoch besonders darauf geachtet werden, dass nun nur noch eine Person Entscheidungen trifft, und nicht, wie üblich, eine Gruppe von Experten. Es muss also ein Weg gefunden werden Fehlschätzungen zu kompensieren, die durch Unkenntnis der Anforderung, der benötigten Technologie oder Überschätzung der eigenen Fähigkeiten entstehen können. Die Praktik des Planning Games ist in den Prozess der Sprintplanung in Kapitel [4.5.2](#) eingeflossen, wurde jedoch leicht angepasst, um für Einzelentwickler angemessener zu sein.

### 3.1.1.3 Test Driven Development

Testgetriebene Entwicklung kann auch von Einzelentwicklern durchgeführt werden und wird die selben Zwecke erfüllen können wie in einer teambasierenden Entwicklung. Daher wurde die Praktik der testgetriebenen Entwicklung ohne Veränderungen übernommen. Jedoch werden hier die Anforderungen nur von einer Person überprüft, da Anforderungserhebender, Tester und Programmierer ein und dieselbe Person sind, bei der Entwicklung in Gruppen werden diese Rollen von mehreren Personen ausgefüllt und durch diese Personen zumindest mit ihrem gesunden Menschenverstand überprüft.

### 3.1.1.4 Whole Team

Da nur eine Person das Team bildet, kann kein Kunde zum Team "hinzugezogen" werden. Hier hängt es also von der Art des Projektes ab, ob dieses Ziel erfüllbar ist, oder nicht. In jedem Fall sollte versucht werden eine große Nähe zum Endbenutzer und zum Kunden zu halten um Anforderungen korrekt erheben zu können und um die gesetzten Ziele zu überprüfen. Jedoch ist die Idee eines vom Entwickler ausgehenden permanenten Kontakts zur Konkretisierung von Anforderungen und ein zeitlich beschränkter Kontakt zur Erhebung neuer Anforderungen über die ganze Entwicklungsdauer hinweg in den Prozess eingeflossen. Die Konkretisierung dieser Idee ist in in Kapitel [4.6](#) genauer beschrieben.

### 3.1.1.5 Continuous Integration

Der Teil, der Continuous Integration, der fordert, dass alle Entwickler mit der neuesten Version des Quellcodes arbeiten müssen, ist für einen Einzelentwickler unnötig, da er immer über die neuste Version verfügen sollte. Er führt alle Änderungen selbst durch und könnte nur durch Wechseln von Arbeitsplätzen oder gewollten Handlungen mit veraltetem Quellcode arbeiten können. Die eingesetzten Hilfsmittel um das Ziel der Continuous Integration zu erreichen und das Ziel immer ein möglichst lauffähiges System zu haben, sind auch für Einzelentwickler von Vorteil. Diese Hilfsmittel und ihre Funktion werden im Abschnitt über die Collective Code Ownership besser beleuchtet. Die Forderung, dass jeder Quellcode sofort getestet und integriert werden sollte, hat natürlich weiterhin Bestand. Integrationsprobleme an das Ende eines Entwicklungszyklus zu verschieben bietet weder für Gruppen noch für Einzelentwickler Vorteile. Die Forderung der Continuous Integration wurden daher nur in sofern übernommen, dass der Autor davon ausgeht, dass die Entwickler durch den Test First Ansatz ihre Tests sofort ausführen und den Quellcode direkt integrieren.

### 3.1.1.6 Refactoring or Design Improvement

Die Qualität des Designs zu erhöhen ist für Gruppen und Einzelpersonen gleich wichtig und sollte von Beiden sehr ernst genommen werden. Refaktorisierung ist daher von einem Einzelentwickler genauso durchzuführen wie in herkömmlichen agilen Projekten. Im Licht der nicht Durchführbarkeit des Pair Programmings erscheint dieses Ziel sogar als noch wichtiger. Deshalb wurde diese Praktik übernommen und Hilfestellungen für eine Verbesserung des Refaktorisierungsprozesses gegeben.

### 3.1.1.7 Small Releases

Das Ziel, Anforderungen schnellstmöglich zu erkennen und das Vertrauen des Kunden zu gewinnen, trifft natürlich auch für Einzelentwickler zu. Die Ausführbarkeit unterscheidet sich, von der in traditionellen agilen Projekten, nur in sofern, dass es anzunehmen ist, dass die Menge des erzeugten Codes und damit indirekt auch der erfüllten Anforderungen in einem gleichlangen Zeitraum kleiner sein wird, wenn nur eine Person arbeitet. Dieses Ziel wurde innerhalb der lauffähigen Sprint Release übernommen und auf den speziellen Prozess von SoDa konkretisiert.

### 3.1.1.8 Coding Standards

Coding Standards sollten auch für Einzelentwickler gelten und ihre Einhaltung überprüft werden, da Wartbarkeit auch für Einzelpersonen ein Ziel darstellt. Daher konnte diese Praktik ohne weitere Veränderungen übernommen werden.

### 3.1.1.9 Collective Code Ownership

Bei einem einzigen Entwickler ist diese Forderung naturgemäß erfüllt und bedarf keiner weiteren Beachtung, doch die Software, die dieses Ziel in einem Team erfüllen soll kann auch einem Einzelentwickler helfen. Versionskontrollsoftware ermöglicht es ihm nicht nur von verschiedenen Orten auf sein Projekt zugreifen zu können, sondern ermöglicht das Zurücksetzen von Änderungen, falls dies Notwendig werden sollte. Die Forderung nach dieser Praktik wird durch die nichtexistenz einer Gruppe obsolet.

### 3.1.1.10 Simple Design

Simple Design ist für Gruppen und Einzelpersonen gleichermaßen erfüllbar und das mit ihm verbundene Ziel ist für einen Soloentwickler ebenso wichtig wie für eine Projektgruppe. Da für Beide die zukünftigen Anforderungen an die Software und damit die Richtung von etwagem vorausschauendem Design unklar ist, damit Schadhaf wirken kann, falls die falsche Richtung eingeschlagen wird und zusätzliche Arbeitszeit benötigt, ist ein solches Design nicht empfehlenswert. Diese Forderung konnte somit unverändert übernommen werden.

### 3.1.1.11 System Metaphor

Entwicklung durch eine Einzelperson kann den Zielen dieser Praktik sowohl Schaden als auch Nutzen. Eine Person hat es einfacher ihre eigenen Bezeichnungen zu verstehen, und kann dadurch einer Systemmetapher besser folgen. Jedoch könnte genau dieser Sachverhalt dazu führen, dass der Entwickler nicht mehr darauf achtet verständliche Bezeichnungen zu wählen und sich dadurch die gesamte Qualität des Codes langsam verschlechtert. Diese Forderung wurde mit einer Ermahnung, jedoch ohne weitere Veränderungen übernommen, da dem Autoren keine Möglichkeit bekannt ist, mit der automatisiert oder für einen Einzelentwickler mit angemessenem Aufwand verbunden, die Nutzung einer verständlichen Metapher sichergestellt werden kann.

### **3.1.1.12 Sustainable Pace**

Das Ziel nur vierzig Stunden zu Arbeiten sollte für Einzelpersonen ähnlich einfach, oder schwer, zu erreichen sein, wie für ein Team. Ob sich ein Student während seiner Bachelorarbeit daran halten kann, bleibt jedoch abzuwarten. Aufgrund dessen wird die Forderung nach einer Maximalbeschränkung der Arbeitszeit nicht übernommen und es wird darauf vertraut, dass der Entwickler seine eigenen Fähigkeiten in diesem Bereich nicht überschätzt.

## **3.2 Scrum**

Scrums grundlegender Prozess der Planung und Ausführung ist nicht an Gruppen geknüpft, aber da die Entscheidungen oft auf dem Konsens von Experten beruhen, ist es anzunehmen, dass gerade in diesen Teilen Änderungen vorgenommen werden müssen.

### **3.2.1 Sprint und Sprintplanung**

Die Struktur von kurzen Zyklen, ist ein wiederkehrendes Motiv agiler Entwicklung und ist notwendig für diese Form der Entwicklung: Die Treffen müssen ersetzt werden, um es einer Einzelperson möglich zu machen, die Aufgaben der Treffen auszuführen.

#### **3.2.1.1 Scrumtreffen**

Einem Team aus einer einzelnen Person fällt es schwerer ein Treffen abzuhalten, da dies als Nebentätigkeit abgetan und übersprungen werden könnte. Doch sind die Fragen die in diesem Treffen gestellt und beantwortet werden auch für sein Projekt äußerst wichtig, daher muss er diese Fragen für sich selbst beantworten. Dieses Treffen wurde als Tagesplanung übernommen, in der der Entwickler sich die Zeit nimmt eben diese Fragen zu beantworten und Dokumente zu aktualisierte.

#### **3.2.1.2 Sprintanfangs Treffen**

Wie schon zuvor beschrieben ist ein Treffen aus einer Person schwierig. Da dieses Treffen einen Konflikt zwischen den Standpunkten des Product Owners und der Entwickler darstellt, wird es zu Problemen kommen, wenn der Entwickler eine der beiden Seiten übervorteilt. Dies wurde im Aufbau der Sprintplanung beachtet, die versucht durch eine klare Struktur und einem Schätzverfahren, diese Nachteile zu beheben.

### 3.2.1.3 Sprint

Für einen Einzelentwickler besteht bei den Sprints das Problem, dass er wahrscheinlich auch innerhalb des Sprints Kontakt zu den Kunden halten muss, und dadurch von seinem momentanen Problemen abgelenkt wird und zukünftige Anforderungen, die erst während des Sprints festgelegt werden, bereits im Vorherein einbaut. In diesem Fall hilft nur Selbstdisziplin um nicht die Ziele die mit der Idee des Sprints verbunden sind zu gefährden. Der Sprint wurde fast unverändert übernommen, nur die Länge wurde auf eine Woche festgelegt. Zusätzlich zu dem bekannten Sprint wurde jedoch ein Analysesprint erschaffen, um Anforderungen explizit zu erheben und die Schätzung der Dauer, die für das erfüllen einer Anforderung benötigt wird, zu verbessern.

### 3.2.1.4 Sprintende Treffen

Wie bereits bei Scrumtreffen besteht hier die Gefahr dieses Treffen als Einzelperson nicht durchzuführen oder dies nur lapidar zu tun. Außerdem besteht hier der Nachteil, dass der gesamte Prozess nur aus einer Position gesehen wird und nicht von mehreren Experten die jeweils ihre eigenen Ideen und Vorstellungen haben und dadurch eine Diskussion hervorrufen können. Ein Einzelentwickler sollte also mehr Energie für dieses Treffen verwenden als ein Team, um den Prozess aus verschiedenen Blickwinkeln zu sehen. Der Prozess der Verbesserung wurde in der Sprintendeplanung konkretisiert und mit Verfahren versehen, die es ermöglichen sollen die Probleme eines Sprints besser zu Visualisieren.

## 3.2.2 Dokumente

Die Wichtigkeit von Dokumenten und ihre Funktion unterscheiden sich nur gering bei einem Einzelentwicklerprozess zu einem traditionellen Entwicklungsprozess. Zwar ist es für eine Einzelperson nicht notwendig Informationen an andere Teammitglieder zu veröffentlichen, aber durch die niedergeschriebenen Dokumente besitzt er eine materielle Repräsentation seines Projektes und seines Fortschritts, die ihm auch helfen keine essentiellen Informationen zu übersehen. Alle Dokumente wurden deshalb in den SoDa Prozess übernommen, und zwei weitere Dokumenttypen eingefügt.

## 3.2.3 Scrum Rollen

Das Problem dieser Aufteilung für einen Einzelentwickler liegt natürlich darin, dass er nur eine Person ist, und damit in jedem Fall ein "Pig", jedoch ist es trotzdem wichtig zu sehen welche Rollen es gibt, da er natürlich immernoch diese Rollen verkörpern muss.

### 3.2.3.1 "Pig"Roles

Diese drei Rollen wird der Entwickler nun selbst übernehmen müssen, und er sollte versuchen diese Rollen streng voneinander zu trennen, um alle Seiten gleichwertig darzustellen.

### 3.2.3.2 "Chicken"Roles

Da das Team nur aus einer Person besteht, können keine Personen existieren die nur Halb zum Team gehören, ähnliche Rollen wird es jedoch auch bei einem Einzelentwicklerprozess geben. Natürlich sollte es auch für einen Soloentwickler feste Ansprechpartner geben die für Fragen zur Verfügung stehen.

## 3.3 Fazit

Der überwiegende Teil der Praktiken und Methoden der beiden agilen Entwicklungsmethoden kann weiterhin genutzt werden. Dies erscheint überraschend, da eine der grundlegenden Aussagen der agilen Entwicklung ist, dass die Kommunikation im Team ein großes Problem darstellt. Jedoch verfolgt die agile Entwicklung auch weitere Ziele, die alle hohe Relevanz für die Softwareentwicklung haben, ob diese nun in Gruppen oder als Einzelperson geschieht.

Eines der größten Probleme die aus der Soloentwicklung hervorgehen ist die Abhängigkeit von einer Einzelmeinung, die durch Fehleinschätzungen und schlechte Übersicht über die Anforderungen sehr stark von der Realität abweichen kann, sehr viel stärker als in einer Gruppenentwicklung in der Schätzung und Meinungen in einem Konsens entstehen und dadurch weniger starke Extremwerte erzeugt.

### 3.3.1 Nutzbare Praktiken des Extreme Programming

Die meisten Praktiken des Extreme Programming lassen sich auf einfachste Weise übernehmen und passen sowohl mit der Soloentwicklung als auch mit den Methoden von Scrum zusammen. **Planning Game** funktioniert analog zu der Planungsmethode von Scrum, auch wenn beide normalerweise die Anwesenheit mehrere Personen erfordern. **Test Driven Development** ist eine erprobte Technik, die sich auch als Einzelperson durchführen lässt. Die angesprochenen Ziele der **Continuous Integration**, die fordern immer die neuste Version



eines Quellcodes sofort in das Projekt zu integrieren und jeden Quellcode vor der Integration zu testen haben auch in jeder Projektform ihre Gültigkeit. Dazu gehörend ist auch, dass die Software in kleinen Schritten entwickelt wird, die jeweils funktionsfähige Programme ergeben, die Forderung der **Small Releases**. Die Notwendigkeit die Struktur und den Code einer kontinuierlichen Verbesserung durch **Refactoring** zu unterziehen, ist eine der wichtigsten Folgerungen aus dem Problem, dass Anforderungen zu keinem Zeitpunkt vollständig verstanden werden können. Die Forderungen über die Art und Weise des Designs und der Struktur des Quellcodes, namhaft also **System Metaphor**, **Simple Design** und das Nutzen von **Coding Standards** sind in gleicher Weise ohne Probleme auch für Einzelentwickler ausführbar, ohne angepasst werden zu müssen.

### 3.3.2 Nutzbare Ideen und Strukturen der Scrum Methodik

Die Entscheidungen, die in den **Treffen** getroffen werden, sind gleichermaßen Relevant für alle Formen der agilen Entwicklung. Sie müssen sowohl von Teams als auch Einzelpersonen zum selben Zeitpunkt und mit der selben Sorgfalt entschieden werden. "Treffen" die eine Person alleine abhält sind nicht vergleichbar mit Gruppentreffen, da ihnen Gruppendynamik und Meinungsvielfalt fehlt. Es müssen Lösungen gefunden werden um die Treffen effektiv zu gestalten und der Gefahr zu entgehen, dass sie als nebensächlich angesehen werden. Der **Sprint** ist Analog zur Forderung von Zyklen und Small Releases im Extreme Programming und ist ein grundsätzliches Muster der agilen Entwicklung und der Erkenntnis, dass Anforderungen sich verändern und nicht Vollständig erfasst sein können. Die **Dokumente** die während des Scrum Prozesses erzeugt werden, können auf die selbe Art und Weise, mit der selben Struktur weiterbenutzt werden, da die selben Informationen auch für Einzelentwickler von Bedeutung sind **Rollen** sind für die Einzelentwicklung nicht so wichtig, wie für Gruppen. Zwar existieren die meisten dieser Rollen in gewisser Weise immer noch, jedoch können ihre Funktionen nicht mehr so leicht getrennt werden.

## 4 SoDa - Solo Development is agile

SoDa ist das Hauptergebnis dieser Arbeit und wurde im Zuge dieser Arbeit sowohl konzipiert als auch evaluiert. Es ist ein agiler Entwicklungsprozess für Einzelentwickler, der auf Extreme Programming und Scrum basiert. Er ist im Zuge dieser Bachelorarbeit entstanden, um das beschriebene Problem, dass agile Entwicklung nur für Teams ausgelegt ist, zu lösen. Dabei ist es, wie in der agilen Entwicklung üblich, kein festgeschriebener unveränderlicher Prozess, sondern ein Rahmen, der es einzelnen Personen erlauben soll agile Entwicklung zu praktizieren und den Prozess ihren Bedürfnissen anzupassen.

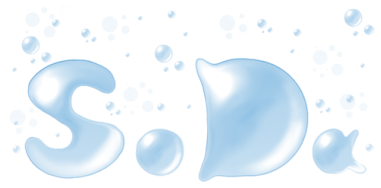


Abbildung 4.1: SoDa Logo

### 4.1 Übersicht

SoDa ist ein agiler Entwicklungsprozess der Entwicklern, die ein Softwareprojekt als Einzelperson durchführen, einen Rahmen bietet. Dabei bietet SoDa einen testgetriebene zyklische Prozess an, der kundennahes dynamisches Entwickeln ermöglicht, jedoch genug Stabilität bietet um eine ausgereifte Software zu entwickeln. Der Prozess ist rund um kurze Zeiträume, so genannte Sprints, aufgebaut in denen am Anfang dieser Zeiträume definierten Ziele verfolgt werden. Sprints haben eine festgelegte Länge von einer Woche, um so schnellstmöglich auf Veränderungen zu reagieren. SoDa stellt Dokumenttypen und Vorgehensweisen zur Verfügung die es dem Entwickler ermöglichen seine Arbeitszeit effektiv zu planen und zu überprüfen ob sich das Projekt noch innerhalb der geplanten Zeit befindet. Schlussendlich verfolgt es das Ziel eine einfach zu wartende und weiterzuentwickelnde Software zu erzeugen, die sowohl der Kunde als auch der Benutzer benötigen.

#### 4.1.1 Ablauf der Entwicklung

Der grundlegende Ablauf der Entwicklung innerhalb von SoDa ist stark an Scrum angelehnt und ist um die einwöchigen Sprints kreiert. Nach jedem Sprint findet eine ein- bis zweitägige



Abbildung 4.2: Vereinfachte Darstellung eines SoDa Projektes

Orientierungsphase statt. Hier wird festgestellt ob die Entwicklung den richtigen Weg geht und ob stärkere Veränderungen vorgenommen werden. Zu Beginn eines Sprints werden Ziele festgelegt, die innerhalb des Sprints nicht mehr verändert werden, so hat jeder Sprint festgelegte Aufgaben, die darin liegen können bestimmte Anforderungen zu erfüllen oder neue Anforderungen zu erheben, während in der Orientierungsphase festgestellt wird, ob es notwendig ist getroffene Entscheidungen zu verändern und ob der Kunde mit den bisherigen Fortschritten zufrieden ist. Für eine Übersicht des Ablaufs siehe auch die veranschaulichende Grafik [4.2](#)

## 4.2 Geeignete Projekttypen

Als agile Methode zur Entwicklung von Software eignet sich SoDa nicht für jede Art von Projekt, und unterliegt den selben Beschränkungen wie andere agile Methoden. Als solche ist SoDa passend für Projekte in denen sich die Anforderungen schnell ändern können, oder zu Beginn des Projektes unklar sind, eignet sich jedoch weniger für die Entwicklung von Echtzeitsystemen oder andere kritische Systemen, deren Entwicklung, Aufgrund von Versicherungs- oder anders gearteter Gründen, sehr Formal sein müssen. Eine weitere Einschränkung entstammt der Größe des Teams, ein einzelner Entwickler wird nur kleinere Pro-

jekte, in einer für den Kunden annehmbaren Zeit, durchführen können. Solche Projekte können Forschungsprojekten oder wissenschaftlichen Arbeiten, wie sie häufig in Hochschulen und Universitäten als Abschlussarbeiten zu finden sind, sein oder Softwareprojekte, deren Größe oder Zeitplanung es erlaubt nur einen einzigen Entwickler an ihnen arbeiten zu lassen. Ein weiterer Verwendungszweck ist die persönliche Zeit- und Entwicklungsplanung für Projekte die nicht agil sind, jedoch den einzelnen Entwickler keine Arbeitsplan vorschreiben. Auch Open Source Projekte, bei denen jeder Entwickler seine eigenes Arbeitspensum plant, sind für eine Verwendung von SoDa geeignet.

## **4.3 Praktiken**

SoDa stellt Verhaltensregeln zur Verfügung, die es einem Entwickler vereinfachen sollen erfolgreich Software zu entwickeln.

### **4.3.1 Feingranulares Feedback**

Die hier aufgeführten Praktik soll dem Entwickler helfen schneller über den Erfüllungsgrad von Anforderungen.klar zu werden.

#### **4.3.1.1 Testgetriebene Entwicklung**

Anwendungsfallbasierte Tests als Abschlusskriterium für Anforderungen, nur wenn der, bei Sprintanfangsplanung festgelegte, Test keinen Fehler meldet, gilt das Kriterium als erfüllt. Bei kritischen Anforderungen können auch andere Black Box Tests als Kriterium für das Abschließen einer Anforderung gefordert werden.

### **4.3.2 Andauernder Prozess**

Hier finden sich Regeln, die die Entwicklung zu einem andauernden zielgerichteten Prozess werden lassen, die es ermöglichen Software zyklisch zu Entwickeln und mit jedem Zyklus besser an die Wünsche des Kunden anzupassen.

### 4.3.2.1 Refactoring

Refactoring ist kein vorgeplanter Prozess in SoDa, der eine festgelegte Anzahl von Stunden durchgeführt wird, sondern eine Notwendigkeit. Sobald Schwächen im Design, nicht generalisierter Code, oder ähnliche Probleme auftauchen müssen sie vom Entwickler behoben werden, bevor Erweiterungen stattfinden. Zusätzlich zur Expertise und dem gesunden Menschenverstand des Entwicklers können auch automatische Analysewerkzeuge Dopplung von Code oder Zyklen in den Objektrelationen erkennen und wichtige Kennzahlen liefern, um dem Entwickler zu indizieren ab welchem Punkt es unumgänglich ist sich der Refaktorisierung zuzuwenden.

### 4.3.2.2 Lauffähiges Sprint Release

Mit jedem Sprint sollte, wenn möglich, ein lauffähiges Programm entstehen, dessen Funktionalität von Sprint zu Sprint zunimmt. Dadurch ist früher möglich Feedback zu erhalten und so die Software zum frühest möglichen Zeitpunkt in die richtige Richtung zu lenken. Durch eine frühe Visualisierung eines lauffähigen Programms wird auch das Vertrauen des Kunden in das Projekt gestärkt, auch wenn der Kunde der Entwickler selbst ist, wie zum Beispiel in einigen akademischen Arbeiten.

## 4.3.3 Verständlichkeit und Kommunikation

Die folgenden Praktiken sollen es einem Entwickler erleichtern seinen eigenen Code zu verstehen und die Kommunikation mit eventuellen anderen Entwicklern zu erleichtern, falls das Projekt zu einem Mehrentwicklerprojekt umgewandelt oder die Entwickler wechseln.

### 4.3.3.1 Coding Standards

Coding Standards sollten unbedingt durch ein automatisches Tool überprüft und erzwungen werden, um so auch Lesbarkeit, und damit die Verständlichkeit und Refaktorisierungsmöglichkeiten hoch zu halten. Für einen Einzelentwickler lohnt es sich in fast allen Fällen sich offenen Standards oder den Standards des Programmiersprachenanbieters zu halten, so lange er nicht einen spezifischen Standard aus anderen Quellen vorgeschrieben bekommt. Die Entwicklung eines eigenen Standards ist oft zu aufwendig und fehlerträchtig.

### 4.3.3.2 Simple Design

Bei jedem Sprint sollte nur das Ziel des momentanen Sprints gesehen werden und nicht darüber hinaus. Der Code sollte so geschrieben werden, dass er die Abschlussbedingungen erfüllt und nicht bereits für zukünftige Anforderungen. Anforderungen können sich, gerade in Forschungsarbeiten, sehr schnell ändern, sobald das Feld indem sich der Entwickler bewegt klarer wird.

### 4.3.3.3 System Metapher

Ein Entwickler sollte so weit wie möglich die Worte aus der Begriffswelt des Anwenders benutzen, um so mit ihm auf einem Stand zu sein und eine funktionierende Kommunikation zu ermöglichen. Auch hier ist dies besonders bei wissenschaftlichen Arbeiten ein wichtiger Punkt, da sich der Entwickler hier tief in ein ihm womöglich nur schlecht bekanntes Wissensgebiet begibt und klare Mittel benötigt um Wissen mit Personen aus diesem Gebiet auszutauschen.

## 4.4 Dokumente

SoDa nutzt eine Anzahl von Dokumenten, die dem Entwickler helfen Arbeit zu organisieren und die Situation des Projektes zu visualisieren. Während die Backlogs eher die Aufgaben einer unverzichtbaren Organisationshilfe und eines Wissensspeichers übernehmen, stellt das Burn Down Chart den Fortschritt während eines Sprints darstellt. Der Inhalt des obligatorischen Glossars wird hier ebenfalls, der Vollständigkeit halber, beschrieben.

### 4.4.1 Product backlog

Das Product Backlog wird während der Kommunikationsphasen und während Analysesprints erstellt und beinhaltet die Anforderungen an das fertige Produkt. Zumindest die wichtigsten Anforderungen, die als nächstes entwickelt werden sollen, sollten bereits durch User Storys repräsentiert sein, um sie in der nächsten Sprintanfangsplanung zu nutzen. Beim Einfügen in das Product Backlog sollte die Bearbeitungsdauer jeder Anforderung schon einmal grob geschätzt werden, um so eine ungefähre Projektdauer angeben zu können.

Das Product Backlog sollte die Anforderungen aus Kundensicht beinhaltet, nicht die Anforderungen aus Sicht des Entwicklers. Anforderungen die nur den Entwickler betreffen oder Festlegungen auf bestimmte Technologien, soweit ihre Nutzung nicht eine Anforderung ist,

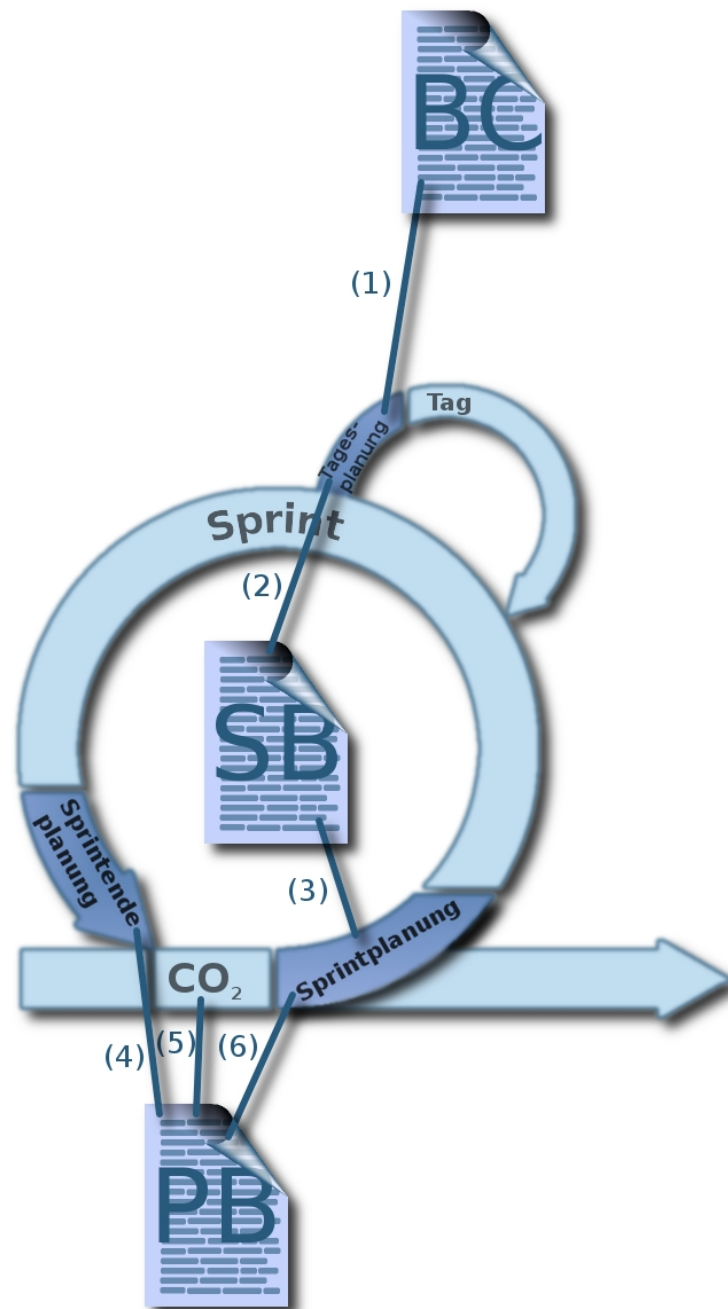


Abbildung 4.3: Dokumente im SoDa Projekt. PB stellt das Product Backlog da, das aus  $CO_2$  Phasen (5) und der Sprintendeplanung (4) mit Anforderungen gefüllt wird. Es wird während der Sprintanfangsplanung genutzt (6) um Anforderungen in das Sprintbacklog oder Analysebacklog, beides dargestellt durch SB, zu übertragen (3). Die Anforderungen werden dann während der Tagesplanung (2) in das Burn Down Chart, das durch BC dargestellt wird, übertragen (1)

| ID | Name                                | Priorität | Schätzung | Use Case  | Notizen   |
|----|-------------------------------------|-----------|-----------|---|---|
| 1  | Rechtschreibprüfung auf Anforderung | 100       | 8 Stunden | Der Benutzer schreibt einen Text und fordert dann die Rechtschreibprüfung an. Alle Worte der momentan geöffneten Szene werden dann mit den Worten in der Rechtschreibungsdatenbank abgeglichen und unbekannte Worte werden gemeldet | Vorschläge für richtige Rechtschreibung sind noch nicht nötig |

Tabelle 4.1: Ein einfaches Beispiel eines Product Backlogs.

sollten noch nicht im Product Backlog getroffen werden, so dass es für den Kunden verständlich ist. Hierdurch kann der Kunde das Product Backlog einsehen und Änderungen vorschlagen und der Zeitpunkt in dem die Richtung in die das Projekt geht wird auf einen späten Zeitpunkt verschoben, in dem die Anforderungen wahrscheinlich besser Verstanden sind. Definitive Festlegungen auf bestimmte Verhaltensweisen oder Technologien sollte erst im Sprint Backlog oder durch die Coding Standards vorgegeben sein.

Da das Product Backlog eine große Tabelle ist, ist die Nutzung eines Tabellenkalkulationsprogramms zu empfehlen um diese festzuhalten.

**ID** Eine Aufsteigende Nummer zur eindeutigen Identifikation

**Name** Ein Name der die Anforderung beschreibt.

**Priorität** Eine Zahl die die Priorität angibt. Die Priorität soll nur zur Ordnung der Anforderungen dienen, eine Anforderung mit der Priorität 20 ist wichtiger als eine mit der Priorität 10, aber nicht unbedingt "doppelt so wichtig". Prioritäten sind nicht festgeschrieben und können jederzeit vom Entwickler geändert werden, sollten sich die Anforderungen ändern. Die Prioritäten die hier festgelegt werden, sollten immer aus Sicht des Kunden sein.

**Schätzung** Eine grobe Schätzung in Story Points, ungestörte Mann-Stunden Arbeit, die es dauern wird um diese Anforderung abzuschließen.



| ID | Name                | Priorität | Schätzung | Anforderungen   | Notizen  |
|----|---------------------|-----------|-----------|---|--|
| 1  | Rechtschreibprüfung | 100       | 3 Stunden | Überprüfung von Worten auf korrekte Rechtschreibung im Hintergrund und auf Anforderung. | Wie groß muss die Anfängliche Wortdatenbank sein? Überprüfen ob freie Datenbanken zur Verfügung stehen |

Tabelle 4.2: Ein einfaches Beispiel eines Analyse Backlogs.

**User Case** Ein kurze Beschreibung des Anwendungsfalls und eine Verknüpfung zum vollständigen Use Case

**Notizen** Notizen über die Anforderung.

#### 4.4.2 Analyse backlog

Das Analyse backlog enthält Aufgaben für die Feststellung von Anforderungen, zu Beginn eines Projektes wird es also mit Punkten gefüllt sein, die die grundlegenden Anforderungen feststellen. Die Punkte dieses Backlogs führen zu Anforderungen im Product Backlog, wenn sie in Analyse Sprints oder während der  $CO_2$  Phase abgearbeitet werden.

**ID** Eine Aufsteigende Nummer zur eindeutigen Identifikation

**Name** Ein Name der die mögliche Anforderung beschreibt.

**Priorität** Eine Zahl die die Priorität angibt, wie im Product Backlog.

**Schätzung** Eine grobe Schätzung in Story Points, ungestörte Mann-Stunden Arbeit, die es dauern wird um diesen Punkt abzuschließen.

**Anforderungen** Eine grobe Übersicht über den Verlauf und die Anforderungen die aus diesem Punkt entstehen könnten.

**Notizen** Notizen über diesen Punkt.

### 4.4.3 Sprint backlog

Das Sprint Backlog wird im Sprintanfangstreffen erstellt, in ihm werden die Anforderungen festgehalten die in diesem Sprint erfüllt werden sollen. Hierzu werden Anforderungen aus dem Product Backlog oder des Analyse Backlogs während des Sprintanfangstreffens übernommen. Das Sprint Backlog ist nach dem Sprintanfangstreffen nicht mehr veränderbar, und sorgt dadurch, dafür das die Anforderungen für die Dauer eines Sprints festgefroren sind und der Entwickler Zeit hat sich auf genau diese Anforderungen zu konzentrieren. Dieses Vorgehen unterstützt den Ansatz des simplen Designs, da der Entwickler nun nur die Anforderungen der Gegenwart sieht.

Anders als im Product Backlog können hier schon im Notizfeld spezifische Ideen oder Technologien zum erledigen einer bestimmten Aufgabe niedergeschrieben werden. Zu diesem Zeitpunkt sollten die Anforderungen so weit verstanden worden sein, dass es möglich ist diese Entscheidungen zu treffen.

### 4.4.4 Burn Down Chart

Das Burn Down Chart dient zur Visualisierung des Fortschritts während des Sprints und erleichtert es dem Entwickler zu sehen ob er sich noch immer im Zeitplan befindet, welche Anforderung er momentan bearbeitet und welche Anforderungen noch bearbeitet werden müssen. Über dem Burn Down Chart steht das Ziel des Sprints, das im Anfangstreffen gesetzt wurde. Darunter Zeilen für noch nicht bearbeitete, bearbeitete und fertige Anforderungen. Außerdem beinhaltet das Chart ein Diagramm auf dem man die Kurve der noch zu erledigenden geschätzten Mannstunden zum Tag des Sprint aufgetragen sind.

Die beiden besten dem Autor bekannte Möglichkeiten für den Aufbau eines Burn Down Charts sind die Benutzung einer Pinwand oder Magnettafel mit Papierkarten oder, falls dies nicht vorhanden ist, die Benutzung eines Tabellenkalkulationsprogramms, da hier ein leichtes editieren und verschieben von Einträgen möglich ist.

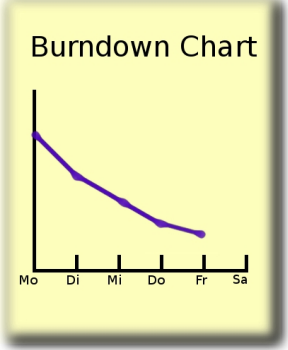
| Sprintziel: Rechtschreibprüfung funktionsfähig machen                             |   |  |   |
|---|---|--|---|
| Nicht bearbeitet  | Wird bearbeitet                                     | Fertig                                     | Burn Down Chart   |
| Vorschläge für korrekte Rechtschreibung (3)<br>Rechtschreibung im Hintergrund (4) | Rechtschreibprüfungs-<br>bibliothek aufbauen<br>(2) | Rechtschreibprüfung<br>auf Anforderung (1) |  <p>Burndown Chart</p> |

Tabelle 4.3: Ein einfaches Beispiel eines Burn Down Charts.

#### 4.4.5 Glossar

Eines der wichtigsten Dokumente der anwenderorientierten Softwareentwicklung ist ein Glossar in dem Bedeutung von Begriffen des Anwenders aufgeschrieben sind. Das Anlegen eines Glossars wird von vielen Entwicklern als eine so Grundlegende Aktion angesehen, dass sie in vielen Beschreibungen von Softwareentwicklungsprozessen nicht mehr erwähnt wird. Der Vollständigkeit halber wird sie hier erwähnt. Das Glossar sollte kontinuierlich gepflegt werden.

#### 4.4.6 Alternativen

Alternativ zu der Benutzung von Magnettafeln, Pinwänden und Tabellenkalkulationsprogrammen wäre es auch möglich generische Software zur agilen Planung von Projekten, wie Double Choco Latte[Dean] des GNU Projektes, oder dem XPlanner[XPlanner] zu nutzen. Diese Tools benötigen jedoch einen hohen Aufwand und haben einen sehr viel größeren Funktionsumfang als der Entwickler benötigt und erschweren ihm so seine tägliche Arbeit. Falls das Nutzen eines solchen Managementprogramms trotzdem gewünscht ist, empfiehlt der Autor den XPlanner oder ein anderes einfaches System zu nutzen, er hat ein einfaches Interface und hat alle Funktionen die benötigt werden. Solche Software kann vor allem großen Nutzen haben, wenn es notwendig ist, das der Entwickler von vielen Orten auf seine Daten zugreifen muss, und sich damit Lösungen wie Pinwände oder Tabellenkalkulationsprogramme als unmöglich oder äußerst unpraktisch erweisen. Eine Web basierende Lösung wie, das bereits

erwähnte, Double Choco Latte kann in diesem Fall im Zusammenhang mit Versionskontrollsoftware sehr hilfreich sein.

## 4.5 Sprint

Ein Sprint ist ein kurzer Zeitraum in dem sich der Entwickler auf das Erreichen eines bestimmten Zieles konzentriert, da er in diesem Zeitraum seinen Fokus auf, zu Beginn des Sprints festgelegte, Anforderungen legen kann. Die typische Sprintlänge von SoDa liegt bei fünf Tagen dies erlaubt es dem Entwickler sich so für eine kurze Periode einer spezifischen Aufgabe zuzuwenden, ohne durch andere abgelenkt zu sein, und sich trotzdem den vielen Anforderungen die an einen Einzelentwickler gestellt werden schnell zuwenden zu können. Die im Vergleich zu Scrum kürzere Zeit des Sprints ermöglicht es auch in den kurzen Bearbeitungszeiten wissenschaftlicher Arbeiten schneller auf Veränderungen reagieren zu können, ohne den jungen Entwickler durch unklare Ziele "in der Luft hängen zu lassen" oder ihm der Gefahr der "Cowboy-Programmierung" auszusetzen.

### 4.5.1 Sprinttypen

Da ein einzelner Entwickler sich auf viele unterschiedliche Aufgaben konzentrieren muss, ist es notwendig neue Formen des Sprints einzuführen, die spezifische Aufgaben haben. Soweit möglich sollten Aufgaben aus einem Sprinttyp nicht mit anderen Sprinttypen vermischt werden, um dem Entwickler nicht aus einer Form des Denkens herauszubringen. Dies scheint im ersten Moment dem Ansatz der agilen Entwicklung zu widersprechen, soll aber diese Prinzipien nur in eine Form gießen, die dem Entwickler die nötige Sicherheit geben soll in einem unbekanntem Gebiet eine Forschungsarbeit durchzuführen oder einem Studenten helfen soll seine Abschlussarbeit zu schreiben.

#### 4.5.1.1 Analyse

Die Aufgabe des Analysesprints ist es neue Anforderungen zu erkennen und ein grundlegendes Verständnis über sie zu gewinnen. Diesen Sprint wird man deshalb häufig am Anfang des Entwicklungsprozesses oder wenn während eines anderen Sprints oder der  $CO_2$  Phase neue Anforderungen auftauchen. Innerhalb des Analysesprints werden die meisten Informationen des Product Backlogs gefüllt, die den Entwicklungssprints ihre Aufgaben geben. Die typischen Aufgaben dieses Sprints lauten "User Story für XY erstellen" und "Benutzung von XY besser verstehen".

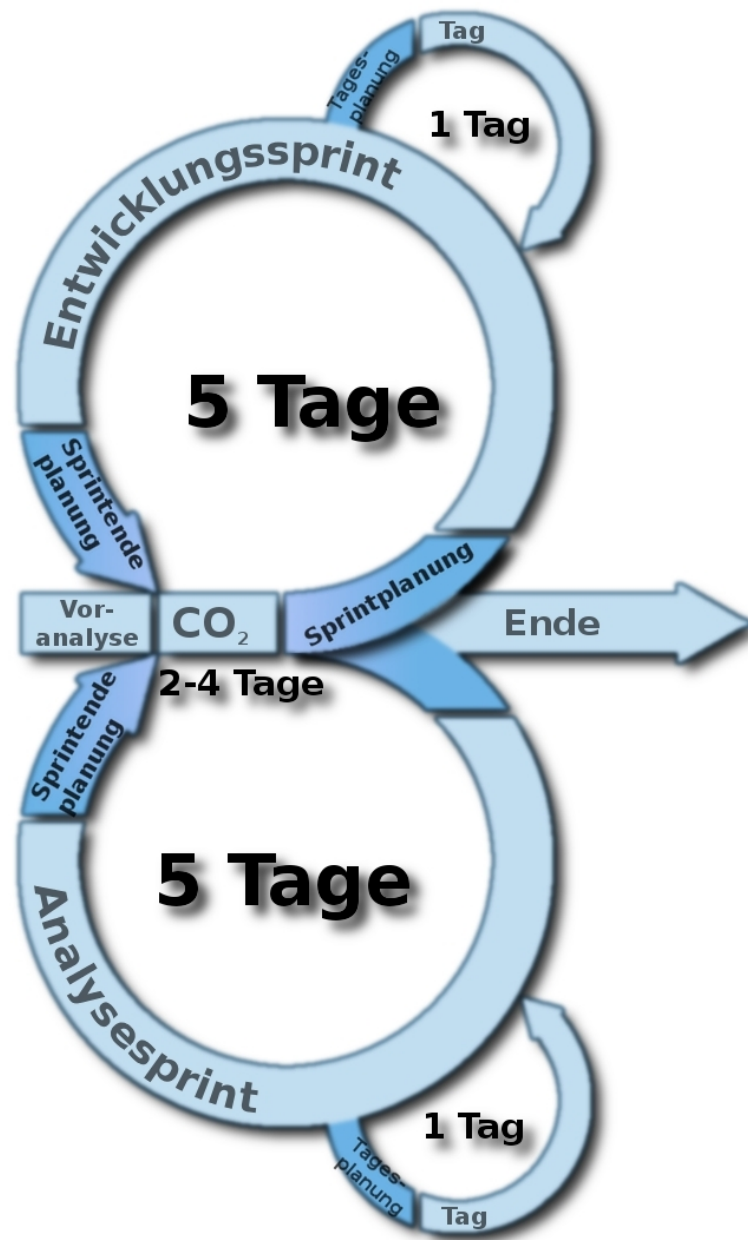


Abbildung 4.4: Vollständige Darstellung eines SoDa Projektes

#### 4.5.1.2 Entwicklung

Während des Entwicklungssprints findet der Hauptteil der eigentlichen Programmierung statt. Die in der Analysephase festgelegten und im Sprintanfangstreffen aus dem Product Backlog entnommenen Anforderungen werden hier erfüllt. Dies bedeutet natürlich ganz und garnicht, dass die Kommunikation mit dem Anwender eingestellt werden soll. Der Kontakt zum Anwender ist ein für das Projekt überlebenswichtiger Prozess und darf deshalb in keiner Phase ganz versiegen. Die Kommunikation soll hier nur einem anderen Zweck dienen, jedoch ähnlich intensiv stattfinden. Innerhalb der Entwicklungsphase geht es darum die letzten Fragen zu den Anforderungen die gefunden wurden, mit dem Kunden zu klären, wenn sie während der Entwicklungsphase auftauchen.

#### 4.5.1.3 Theoretische weitere Sprinttypen

Theoretisch wäre es möglich noch viele weitere Sprinttypen einzuführen, jedoch rät der Autor davon ab den Entwicklungsprozess zu überladen und mit zu komplizierten Regeln zu versehen. *Testsprints* wären zwar möglich, jedoch ist in SoDa Testen ein inhärenter Prozess der Entwicklung, und sollte nicht nur zu bestimmten Zeitpunkten stattfinden. Wenn die Notwendigkeit für *Testsprints* gesehen wird, sollte man den bisherigen Entwicklungsprozess überdenken und ihn anpassen. Für wissenschaftliche Arbeiten sind *Dokumentationssprints* vielleicht vonnöten um die akademische Arbeit selbst zu erstellen.

#### 4.5.2 Sprintplanung

Zu Beginn eines Sprints findet eine Sprintplanung statt, in der der Entwickler bestimmt welche Anforderungen er während des nächsten Sprints erfüllen will und welches Ziel der Sprint verfolgen soll. Dazu ist es wichtig, dass die wichtigsten Anforderungen bereits mit genauen User Storys versehen wurden und der Entwickler sich bereits intensiv, während eines *Analysesprints*, mit ihnen beschäftigt hat. Eine solche Planung sollte zeitlich auf eine Stunde begrenzt sein, um zu vermeiden, dass unnötig Zeit mit Überlegungen und Abwägungen verbracht wird, die nicht zu einem besseren Ergebnis führen. Die Planung sollte damit beginnen, dass der Entwickler sich die höchstpriorisierten Anforderungen durchliest und ihre Bearbeitungszeit erneut schätzt. Danach wird anhand der geschätzten Zeiten und der Prioritäten der Kunden entschieden welche Punkte in das Sprint Backlog übernommen werden.

#### 4.5.2.1 Sprintplanungsübersicht

- Durchlesen der höchstpriorisierten Anforderungen
- Schätzung der Dauer dieser Anforderungen durch das im folgenden Beschriebene Schätzverfahren
- Einpassen der Anforderungen in den Sprint. Anforderungen die nicht vollständig passen müssen Aufgeteilt werden, oder durch niedrig priorisierte Anforderungen ersetzt werden, so dass die vorhandene Zeit nicht von der geschätzten Zeit überschritten wird.
- Eintragen der Anforderungen in das Sprint Backlog oder Analyse Backlog.
- Vorbereiten des Burn Down Charts
- Sprintende nach spätestens 60 Minuten

#### 4.5.2.2 Schätzung

Um den Aufwand für die Erfüllung einer Anforderung abzuschätzen, wird nur die Überlegung des Entwicklers genutzt, da es keine perfekte Möglichkeit gibt. Der Entwickler soll dabei den Aufwand in Story Points angeben, einer Einheit die etwa mit ungestörten Mannstunden Arbeit übereinstimmt. Wichtig bei der Schätzung ist es nicht auf die richtige Anzahl der Stunden zu kommen, sondern, dass die Schätzungen untereinander das richtige Verhältnis haben. Eine Aufgabe mit 20 Story Points also etwa halb so lange braucht wie eine Aufgabe mit 40 Storypoints. Um von Story Points zu Arbeitszeit zu kommen gibt es eine einfache Formel, die auf der Yesterdays Weather Methode beruht.

$$\text{Arbeitsstunden} = \frac{gAIS}{SPIS} * SP$$

Wobei gAIS die gemessenen Arbeitsstunden des letzten Sprints, SPIS die Storypoints des letzten Sprints und SP die Storypoints dieses Sprints sind. Da es häufig zu Verspätungen kommen kann, die in dieser Formel nicht berechnet wurden, zum Beispiel durch unerwartete Ablenkungen, Krankheit, oder durch andere Schwierigkeiten die behoben werden konnten erscheint es manchmal notwendig die Zahlen des letzten Sprints für die Berechnung anzupassen. Bevor ein solcher Schritt gemacht wird, sollte sich der Entwickler Fragen ob das Problem wirklich behoben ist und ob es nicht durch die Probleme des Vorsprints wieder Probleme in diesem Sprint geben könnte, zum Beispiel durch notwendige Refactoringarbeiten die nun getätigt werden müssen, um Code zu verbessern, die durch Zeitknappheit entstanden sind.

**Erster Sprint** Während des ersten Sprints den man durchführt, kann es natürlich keine Vergangenen Werte geben. Hier ist es notwendig einen Richtwert zu nehmen, falls der Entwickler nicht bereits Zahlen aus anderen Projekten, die er durchgeführt hat, besitzt. Ein unerfahrener Entwickler sollte den Bruch durch einen Wert von 2 ersetzen, und somit die Arbeitszeit gegenüber der Schätzung etwa verdoppeln. Dies verkleinert zwar die Anzahl der erreichten Ziele, am ende des ersten Sprints, verhindert jedoch dass die Qualität des Codes leiden muss.

**Alternativen** Alternativ zu der Yesterdays Weather Methode gibt es noch weitere Möglichkeiten die Dauer der Bearbeitung zu schätzen. Allen dem Autor bekannten Methoden sind jedoch eben so Fehleranfällig wie die Yesterdays Weather Methode, und lassen sich zumeist nicht ad hoc anwenden. Die meisten dieser Methoden ziehen Vergleiche zu anderen Projekten, da jedoch jedes Projekt anders verläuft und sich die Leistungen eines Entwicklers zu einem anderen sehr stark unterscheiden können lassen sich allgemeine Zahlen nur schlecht nutzen. Methoden die eine detailliert Schätzung erfordern und Aufgaben in kleine Teilaufgaben zerlegen, können den Nachteil haben, dass beim Zerlegen zeitaufwendige Teilaufgaben vergessen werden. Die geringe Größe der Aufgaben, die in SoDa gefordert wird, sollte es den meisten Entwicklern ermöglichen eine Aufgabe, soweit es überhaupt vom Beginn des Sprints möglich ist, zu überblicken und die Bearbeitungsdauer zu schätzen.

#### 4.5.2.3 Priorisierung

Die Priorität einer Anforderung sollte aus Sicht des Kunden bestimmt werden, der Entwickler sollte immer versuchen die Anforderungen zuerst zu verwirklichen, denen der Kunde die größte Wichtigkeit beimisst. Dadurch wird die Zuversicht des Kunden über die Qualität des Projektes gestärkt, und eine bessere Zusammenarbeit wird dadurch ermöglicht.

#### 4.5.2.4 Zeit

Ein wichtiger Punkt für die Planung des Sprints sind neben den Schätzungen der Bearbeitungsdauer auch die Zeit die dem Entwickler während des Sprints zur Verfügung steht. Die Entscheidung, wieviele Stunden pro Tag ein Entwickler für sein Projekt aufbringen kann obliegt natürlich nur ihm, jedoch stimmt der Autor den Prinzipien des Extreme Programmings zu, dass ein ausgeruhter Entwickler weniger Fehler macht.



#### 4.5.2.5 Konflikt

Das Festlegen der Anforderungen die im nächsten Sprint erfüllt werden sollen, ist mehr als ein einfaches Anordnungsproblem. Ein Sprint sollte möglichst ein spezifisches Ziel erfüllen, das für den Kunden greifbar ist. Es gibt mehrere Möglichkeiten, wie man darauf reagieren kann, wenn eine Anforderung nicht mehr ganz in den Sprint passt. Diese Anforderung sollte jedoch auf keinen Fall "einfach" mit in den Sprint genommen werden.

**Priorität neu Anordnen** Eine Möglichkeit ist die Priorität der Anforderung anzupassen, falls die Anforderung für den Kunden so wichtig ist, dass sie innerhalb dieses Sprints fertig werden muss, sollte die Priorität so angehoben werden, dass andere Anforderungen "herausfallen". Falls die anderen Anforderungen so wichtig sind, dass sie Dringend erfüllt werden müssen, sollte die Priorität der Anforderung abgesenkt werden, so dass Anforderungen mit einer kürzeren Bearbeitungsdauer Teil des Sprints werden können.

**Anforderungen aufteilen** Eine weitere Möglichkeit wäre es, die Anforderung in kleinere Aufgaben zu teilen, falls das möglich ist. Zum Beispiel die Anforderung "Exportfunktion in wichtige Dateiformate" zu spezifizieren und zuerst nur einige wenige zu verlangen, und die anderen auf einen späteren Sprint zu verschieben.

#### 4.5.3 Tagesplanung

Jeden Morgen während des Sprints sollte jeder Entwickler sich drei wichtige Fragen stellen:

- Was habe ich gestern geschafft?
- Was habe ich vor heute zu tun?
- Was könnte mich daran behindern dieses Ziel zu erreichen?

Um sich diese Antworten wirklich vor Augen zu führen sollte der Entwickler sich für jede dieser Fragen eine kurze Antwort aufschreiben oder sie aufzunehmen und sie sich danach anzuhören. Da diese Fragen ein essentieller Teil des Entwicklungsprozesses sind, muss sich der Entwickler mit ihnen auseinandersetzen. Danach sollte er das Burn Down Chart aktualisieren und überprüfen, ob er sich noch im Zeitplan befindet.

## 4.5.4 Sprintendeplanung

Zum Ende eines Sprints sollte eine Retrospektive des letzten Sprints stattfinden. Das Ziel dieser Planung ist es aus den vergangenen Geschehnissen zu lernen und den Prozess und die Handlungen des Entwicklers an die Anforderungen des Projektes anzupassen und ihm so ein einfacheres Entwickeln der Software zu ermöglichen. Der Entwickler sollte bei jeder Sprintendeplanung eine Mind Map erstellen, die die wichtigsten positiven und negativen Punkte des letzten Sprints aufzeigt, sobald dem Entwickler nichts mehr einfällt, sollte er auf die Aufzeichnungen über die Probleme aus den täglichen Treffen zurückgreifen und diese und weitere die ihm währenddessen einfallen in die Mind Map eintragen. Die Probleme aus dieser Mind Map die nicht bereits überwunden wurden, sollte noch einmal aufgeschrieben werden und versucht werden Lösungen für sie zu finden. Die Sachen die gut im letzten Sprint gelaufen sind, sollten, ebenso, aufgeschrieben werden und im nächsten Sprint wiederholt werden. Entwickler die bereits Erfahrung mit kognitiven Maps haben sollten versuchen diese für das Auffinden von Stärken und Schwächen nutzen, da diese den positiven und negativen Aspekt der Punkte hervorhebt.

### 4.5.4.1 Die goldene Regel

SoDa ist eine agile Methode für die Entwicklung von Software, das heißt jedoch nicht nur, dass die entstehende Software an die sich verändernden Anforderungen angepasst wird, sondern auch dass der Entwicklungsprozess selbst agil ist. Wenn der Entwickler ein Problem sieht oder einen Teil des Entwicklungsprozesses nicht durchführen kann, sollte er den Prozess anpassen.

## 4.5.5 Sprintende und unfertige Anforderungen

Falls am Ende eines Sprints noch Anforderungen des Sprint Backlogs nicht erfüllt sind, gibt es zwei grundlegende Möglichkeiten mit diesem Problem umzugehen. Man könnte die Un-erfüllten Anforderungen übernehmen und im nächsten Sprint erfüllen, der Entwickler muss dann die Restzeit die er für die Lösung braucht schätzen und angeben. Die zweite Möglichkeit ist es den Code, der für die Erfüllung dieser Anforderung gedacht war, zu löschen, und die Anforderung im nächsten Sprint erneut von Beginn an zu bearbeiten. Die zweite Alternative erscheint zwar äußerst radikal, hat jedoch ihre Vorteile darin, dass ein falscher Lösungsansatz nicht über längere Zeit verfolgt wird und verhindert so das ein Problem über einen langen Zeitraum von der falschen Seite aus angegangen wird. Der Autor dieser Arbeit wird beide Möglichkeiten während des Versuches nutzen und seine Meinung in den

abschließenden Kapiteln wiedergeben. Den Sprint zu verlängern, bis die Anforderung abgeschlossen ist, ist keine Alternative, dies würde nur zu Verspätungen des Projektes und zu einem wahrscheinlichen Scheitern des Projektes führen.

## 4.6 $CO_2$ Communication, Organisation and Orientation

Zwischen den Sprints sollte es kurze Phasen, von 2-4 Tagen geben, in denen der Entwickler sehr stark mit dem Kunden kommuniziert, um neue Anforderungen aufzudecken oder alte Anforderungen mit größerem Detail zu versehen. Während dieser Zeit werden alle organisatorische Maßnahmen vorgenommen, die nötig sind um die nächste Sprintanfangsplanung vorzubereiten. Diese Phasen sind der eigentliche agile Grundstein in dem sich das Projekt neu Orientiert um das Ziel zu finden.

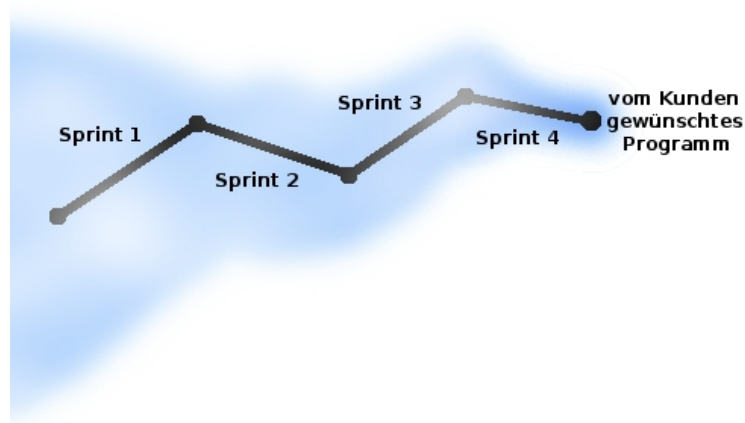


Abbildung 4.5: Darstellung des mit jeder  $CO_2$  Phase immer genauer werdenden Anforderungen, die am Ende zum korrekten Programm führen.

# 5 Unterstützungssystem für Autoren

Der vorgestellte Entwicklungsprozess wird im Zuge dieser Bachelorarbeit am Beispiel der Entwicklung eines Unterstützungssystem für Autoren evaluiert. Ein Unterstützungssystem für Autoren, soll Autoren belletristische Literatur unterstützen. Dabei ist die Zielgruppe, die der Autoren die den Computer als Arbeitswerkzeug gewählt haben und Standardtextverarbeitungs- oder Spezialsoftware nutzen um ihre Werke zu erstellen. Ein solches System ist besser an die Bedürfnisse eines Autors angepasst als eine Standardsoftware, da Autoren anderen Anforderungen haben als die Nutzer für die diese Systeme gebaut werden. Ein solches Unterstützungssystem sollte dem Autoren bereits die typische Aufteilung von Geschichten in Szenen, Kapiteln und Büchern vorgeben und ihm helfen so Ordnung in seine Geschichte zu bringen und sollte Exportfunktionen bieten die es einem Autoren ermöglichen in wichtige Formate für Veröffentlichungen zu speichern.

## 5.1 SoDa in der Entwicklung

Der Hauptzweck der Entwicklung des Unterstützungsystems ist die Evaluation des Entwicklungsprozesses von SoDa. Aus diesem Grund soll in diesem Kapitel auch ein kurzer Einblick in die praktische Anwendung SoDas gegeben werden. Die Schlüsse der Evaluation und die daraus folgenden Veränderungen sollen jedoch erst in Kapitel 6 gezogen werden, während dieses Kapitels eine Übersicht über die Probleme die während des Projektes aufgetreten sind und die Software an sich darstellt.

### 5.1.1 Erheben und Festhalten von Anforderungen

Das Erheben von Anforderungen ist einer der essentiellsten und schwierigsten Teile der Softwareentwicklung, oft werden bereits hier Fehler gemacht, die zu einem späteren Scheitern des Projektes führen. Die Erkennung von falsch verstandenen Anforderungen ist eines der Hauptanliegen der agilen Entwicklung und auch SoDa macht hier keine Ausnahme, deshalb verfügt SoDa über spezielle Analysesprints. Nachdem der Entwickler sich, in ersten Gesprächen, ein grobes Bild der zu entwickelnden Software gemacht hat, kann er bereits erste Aufgaben für Analysesprints entwerfen, die zu weiteren Aufgaben führen.

### 5.1.1.1 Beispiel

Während der Entwicklung des Unterstützungssystems wurden Konkurrenzprodukte untersucht. Dabei wurden sowohl funktionale als auch nichtfunktionale Anforderungen beachtet, die Ergebnisse dieser Untersuchungen findet der geneigte Leser in Kapitel 5.4. Um funktionale Anforderungen zu finden wurde vor allem auf die Liste der Features der Software acht gegeben. Während der Erhebung und Überprüfung der nicht funktionalen Anforderungen wurde ein Text mit 10.202 Worten verwendet, um eine angefangene Geschichte zu Symbolisieren, das erste Buch "Faust: Der Tragödie erster Teil" hat zum Vergleich etwas über 30.000 Worte. Weitere 287 Wörter wurden hinzugefügt um die Geschwindigkeit bei der Bearbeitung einer solchen Geschichte zu überprüfen.

## 5.1.2 Erfüllen und Testen von Anforderungen

Da SoDa dem Test-First Ansatz folgt werden die Tests geschrieben bevor der sie erfüllende Code geschrieben wurde. Dabei wurde iterativ gearbeitet, so dass zuerst nur ein Test geschrieben wird, der einen Teil der Anforderungen überprüft, dieser dann erfüllt und dann ein weiterer Teil der Anforderungen getestet wird. So war es möglich die Arbeit in kleine Übersichtliche Schritte aufzuteilen, die für den menschlichen Geist einfach zu erfassen sind. Hierdurch ist es möglich Fehler in den Anforderungen leichter zu erkennen. Die Tests werden als JUnit Testsuiten erstellt und bestanden aus anwendungsfallbasierenden Tests.

### 5.1.2.1 Beispiel

Während der Erstellung eines frühen Prototyps, um die Machbarkeit der Visionen des Autoren zu überprüfen, wurde die Anforderung 3 erfüllt, diese beschreibt die Ordnung von Szenen nach bestimmten Kriterien, namentlich der Anfangszeit der Szene. Im ersten Testfall wurde hierfür der einfachste Fall angenommen, dass keine Szenen vorhanden sind, und nur eine leere Liste herausgegeben wird. Der zweite Fall, dass alle Szenen in der selben Reihenfolge geschehen in der sie in der Geschichte auftauchen, konnte ähnlich einfach gelöst werden in dem die Szenenliste ohne Veränderung ausgegeben wird. Die nächsten Schritte waren eine Verschiebung von zuerst einer und dann einer beliebigen Anzahl von Positionen. Folgend der Erfüllung der Anforderung gab es noch eine Refaktorisierung, in dem die Arbeit der Sortierung in eine neue Listenklasse verschoben wurde, die Szenen nach ihrer Anfangszeit sortiert.

| ID | Name   | Priorität | Schätzung | Anforderungen   | Notizen  |
|----|--|-----------|-----------|---|--|
| 36 | yWriter 4 - Analyse der funktionalen Anforderungen       | 30        | 2 Stunden | Funktionale Anforderungen aus den Feature Listen des Entwicklers und der Software selbst herausfinden und Eintragen |  |
| 37 | yWriter 4 - Analyse der nicht-funktionalen Anforderungen | 25        | 4 Stunden | Funktionale Anforderungen aus Aufgabe 36 auf Qualität überprüfen.   | Geschichte Eintragen und das letzte Kapitel per Hand eintragen um Reaktionsgeschwindigkeit zu prüfen. Erweiterte Features wie Notizfunktionen oder exportierbare Dateiformate überprüfen. Siehe auch Checkliste "Software Analyse" |

Tabelle 5.1: Ein kleines Beispiel für Analyseaufgaben.

## 5.2 Die Vision

Der Autor dieser Arbeit hatte zu Beginn dieses Projektes die Idee ein Unterstützungssystem für Autoren zu entwerfen, das es Autoren ermöglichen sollte seine Arbeit leichter zu verrichten und ihm lästige Aufgaben, wie die Formatierung, abnehmen und es ihm erleichtern sollte seine Geschichte zu strukturieren.

### 5.2.1 Anforderungen

Das Unterstützungssystem soll es Autoren unter einfachsten Bedingungen ermöglicht ihre Geschichte und ihre Gedanken niederzuschreiben, zu ordnen und anzusehen. Weiterhin soll es dem Autoren möglich sein seine Arbeit schnell und einfach in Formate umzuwandeln in denen er sie veröffentlichen kann.

#### 5.2.1.1 Funktionale Anforderungen

Die Vision des Autors hat eine Vielzahl von funktionalen Anforderungen. Die weit über die Fähigkeiten eines normalen Textverarbeitungssystems für Büros hinausgehen.

Tabelle 5.2: Eine Liste der funktionalen Anforderungen der Vision des Autors dieser Arbeit

| ID | Name                                  | Priorität | Schätzung | Use Case   | Notizen |
|----|---------------------------------------|-----------|-----------|--|---------|
| 2  | Teilung in Bücher, Kapitel und Szenen | x         | -         | Benutzer erstellt ein neues Buch und wird durch einen Dialog geführt, in dem er Buchname, Art[Kurzgeschichte, Roman, Erzählung] und Genre einträgt. Am Ende des Dialogs wird ein namenloses Kapitel erstellt und geöffnet, in dem der Autor durch Doppelklick eine neue Szene erstellen kann, die durch Doppelklick geöffnet wird. Siehe auch Use Case "Teilung in Bücher, Kapitel und Szenen" |         |

Fortsetzung auf nächster Seite...

Tabelle 5.2 – Fortgesetzt

| ID | Name                   | Priorität | Schätzung | Use Case  | Notizen |
|----|------------------------|-----------|-----------|---|---------|
| 3  | Zeitskala              | x         | -         | Benutzer versieht seine Szenen mit Start- und Endzeitpunkten und ruft die Zeitskala Funktion auf. Ein Fenster öffnet sich in dem die Szenennamen oder die ersten Worte einer Szene in Kästen in chronologischer Reihenfolge angezeigt werden. Siehe auch Use Case "Zeitskala" |         |
| 4  | Notizfunktion          | x         | -         | Der Benutzer erstellt per Tastenkombination oder Mausclick auf ein Icon eine Neue Notiz, die sich zum Schreiben öffnet. Siehe auch Use Case "Notizfunktion"   |         |
| 5  | Charakterinformationen | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Charakter aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Charakterinformationen"    |         |
| 6  | Ortsinformationen      | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Ort aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Ortsinformationen"               |         |

Fortsetzung auf nächster Seite...



Tabelle 5.2 – Fortgesetzt

| ID | Name                                      | Priorität | Schätzung | Use Case   | Notizen |
|----|---|-----------|-----------|--|---------|
| 7  | Objektinformationen                       | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Objekt aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Objektinformationen" |         |
| 8  | Beziehungen                               | x         | -         | Der Benutzer trägt eine Beziehungsänderung zwischen zwei zuvor erzeugten Objekten oder Personen in einer Szene über das Optionsmenü ein. Siehe dazu auch Use Case "Beziehungen"  |         |
| 9  | Automatische Formatierung                 | x         | -         | Der Benutzer druckt seine Geschichte aus. Überschriften, Hervorhebungen Zeilen- und Seitenumbrüche werden automatisch von der Software erzeugt. Siehe dazu auch Use Case "Automatische Formatierung"   |         |
| 10 | Anpassung der Formatierung durch $\LaTeX$ | x         | -         | Der Benutzer geht in das Optionsmenü und verändert durch Eingabe von $\LaTeX$ Befehlen die Formatierung. Siehe dazu auch Use Case "Anpassung der Formatierung durch $\LaTeX$ "   |         |

Fortsetzung auf nächster Seite...

Tabelle 5.2 – Fortgesetzt

| ID | Name   | Priorität | Schätzung | Use Case  | Notizen |
|----|--|-----------|-----------|---|---------|
| 11 | Anpassung der Formatierung durch Einstellungen | x         | -         | Der Benutzer geht in das Optionsmenü und verändert durch Auswahl von Schriftart, Schriftgröße, Zeilenabstand und Papierformat die Formatierung. Siehe dazu auch Use Case "Anpassung der Formatierung durch Einstellungen"   |         |
| 12 | Überprüfungsfunktion für zeitliche Richtigkeit | x         | -         | Der Benutzer verschiebt eine Szene vor eine Szene die mit beide mit Zeitinformationen versehen sind, die verschobene Szene ist auf einen Späteren Zeitpunkt geplant als die Szene über die hinweg verschoben wird, daher warnt die Software mit einem Dialogfenster und zeigt die nicht verschobene Szene an. Siehe dazu auch Use Case "Überprüfungsfunktion für zeitliche Richtigkeit" |         |
| 13 | Objekte, Personen und Kausalität               | x         | -         | Der Benutzer verschiebt eine Szene in der Person/Objekt A und Person/Objekt B vorkommen vor eine Szene in der die Verbindung zwischen Person/Objekt A und Person/Objekt B verändert wird. Der Benutzer wird dann gewarnt und die Szene in der diese Beziehungsveränderung getan wird, wird angezeigt. Siehe dazu auch Use Case "Objekte, Personen und Kausalität"                       |         |

Fortsetzung auf nächster Seite...

Tabelle 5.2 – Fortgesetzt

| ID | Name              | Priorität | Schätzung | Use Case  | Notizen |
|----|-------------------|-----------|-----------|---|---------|
| 19 | Versionskontrolle | x         | -         | Auf Befehl des Benutzers oder in vom Benutzer definierten Zeitabständen wird eine vollständige Kopie des geöffneten Buches gemacht, die mit Namen und einer automatisch generierten Nummer gespeichert wird. Siehe dazu auch Use Case "Versionskontrolle" |         |

**Bücher, Kapitel & Szenen** Das Programm soll Geschichten als Objekte behandeln, die in Bücher, Kapitel und Szenen eingeteilt werden, um dadurch eine einfache Strukturierung zu ermöglichen. Da alle Geschichten, die dem Autor bekannt sind, diese oder eine sehr ähnliche Struktur aufweisen, ist diese Festlegung keine Einschränkung.

**Zeitskala und Szenenskala** Um die Strukturen der Geschichte besser zu Visualisieren sollten die Szenen in einer linearen Liste darstellbar sein, in denen es auch möglich ist Szenen, Kapitel und Bücher zu verschieben um die Reihenfolge, wie sie in der gedruckten oder exportierten Form auftauchen zu verändern. Es sollte einem Autor auch möglich sein die Szenen nach Zeitlichen Abläufen zu sortieren, wenn er für die Szenen relative oder absolute Zeitwerte eingibt. Da sich die zeitlichen Abläufe und die Reihenfolge im Werk selbst unterscheiden können, gibt dies einen neuen Blickwinkel auf die Geschichte und kann einem Autor so helfen Lücken oder Fehler zu erkennen.

**Notizen und Informationen** Als Unterstützung zu Handgeschriebenen Notizen sollte das Programm auch über Möglichkeiten verfügen zu Objekten der Geschichte Notizen hinzuzufügen. Diese Objekte umschließen sowohl die bereits erwähnten Szenen, Kapitel, Bücher und Geschichten, als auch Personen und andere wichtige Objekte. Diese Notizen sollten von einfachen Texten, in denen der Autor beliebig Notizen einfügen kann, bis zu Notizkarten für Personen, in denen, zusätzlich zu einem allgemeinen Notizfeld, bereits einige vordefinierte Felder existieren, reichen.

**Beziehungen** Ein weitaus weitreichenderes Ziel ist es die Beziehungen von wichtigen Personen, Orten und Objekten der Handlung abzubilden. Der Autor müsste dafür diese Beziehungen und Veränderungen in diesen Beziehungen händisch angeben. Könnte dadurch aber unter Umständen tiefere Einsichten in seine eigene Geschichte bekommen, die es ihm erleichtern sie zu schreiben.

**Formatierung** Während des Prozesses des kreativen Schreibens sollte die Formatierung des Textes keine Rolle spielen und soweit möglich von der Software selbst übernommen werden. Der Autor wählt für seine Geschichte ein bestimmtes Format, dass dann für die Geschichte automatisch erzeugt wird. Die Software erzeugt ein  $\text{\LaTeX}$ Dokument in denen die Bücher, Kapitel und Szenen aufgereiht sind und entsprechend den Einstellungen formatiert ist, aus diesem Dokument können weitere Dateitypen erzeugt werden.

**Überprüfungsfunktion für zeitliche Richtigkeit** Falls ein Autor die zeitliche Übersichtsfunktion benutzt kann er beim Verschieben einer Szene oder eines Kapitels darauf hingewiesen werden, welche Teile der Geschichte zwischen dem alten Zeitpunkt und dem neuen Zeitpunkt liegen, und damit überprüft werden müsse, ob es hier Veränderungen geben muss. Dadurch können Inkonsistenzen leichter aufgespürt werden und etwaige Bearbeitungszyklen verkürzt werden.

**Objekte Personen und Kausalität** Wenn der Autor die Möglichkeit nutzt Beziehungen zu modellieren kann die Software den Autor warnen, wenn Szenen so verschoben werden, dass dadurch Beziehungen zerbrochen werden, zum Beispiel ein Objekt mit einer Person vorkommt bevor die Szene auftaucht die dieses mit dieser Person in Verbindung gebracht hat. Ähnliche Warnfunktionen sollten auch für das Löschen von Charakteren, Szenen und Objekten möglich sein, so das alles an dem das zu Löschende beteiligt war zur Änderung angezeigt werden kann. Dadurch könnten einfache Fehler, wie das auftauchen einer Waffe, ohne das sie zuvor erwähnt wird, verhindert werden. Dies ist für die meisten Autoren ein wünschenswertes Ziel, da es eine viel beschriebene Regel ist dieses sogenannte "Deus ex machina" zu vermeiden.

**Versionskontrolle** Da das schreiben von Geschichten eine langwierige und kreative Aufgabe ist, ist es unverzichtbar sicherzustellen, dass keine Daten verlorengehen. Oft kann es auch gewünscht sein alte Versionen der Geschichte wiederherzustellen, um Änderungen rückgängig zu machen. Eine solche Funktionalität sollte sowohl automatisch als auch auf Wunsch Sicherheitskopien erstellen.

### 5.2.1.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen der Software sollen vor allem sicherstellen, dass ein Autor seine Geschichte schreiben kann ohne dabei von der Software gestört zu werden. Dies bedeutet, dass während des Schreibprozesses keine Wartezeiten entstehen dürfen und die Software einfach zu bedienen sein muss. Die Stabilität dieser Software muss nicht über die hinaus gehen, die man von ähnlicher Anwendersoftware gewohnt ist und spezielle Ausfallsicherungen für Hardwareausfälle sind ebenfalls in der Vision des Autors nicht vorgesehen. Kompatibilität zu anderen Softwaresystemen und die Übertragbarkeit ist nur in einem sehr geringem Maße notwendig.

**Zuverlässigkeit** Die Zuverlässigkeit der Software muss vor allem für die Wiederherstellbarkeit gegeben sein. Fehler des Programms oder Abstürze die nicht die Zerstörung des Datenspeichers, zum Beispiel der Festplatte, zur Folge haben, dürfen nicht zu einem Verlust gespeicherter Daten führen.

**Benutzbarkeit** Da die Software für computeraffine Schriftsteller erstellt wird, sollten Personen die Erfahrung mit gängiger Textverarbeitungssoftware haben nach einer Einführung innerhalb von wenigen Stunden die Software zum Erstellen von Geschichten nutzen können. Grundlegende Funktionen, wie das Erstellen einer neuen Geschichte, eines Kapitels oder das Schreiben von Szenen sollten für einen Benutzer mit solchen Kenntnissen innerhalb weniger Minuten zu erschließen sein.

**Aufruf der Notizfunktion** Die Notizfunktion sollte schnell und unkompliziert aufgerufen werden können, um Ideen sofort niederschreiben zu können, wenn sie auftauchen und um schnellen Zugriff auf die dort gespeicherten Informationen zu haben. Diese Vorgänge dürfen den eigentlichen Prozess des Schreibens nicht mehr behindern als notwendig.

**Effizienz** An die Effizienz der Software werden nur geringe Anforderungen gestellt, die Software sollte ähnlichen Anforderungen genügen wie eine Standardtextverarbeitungssoftware.

**Änderbarkeit** Da die Software allgemeine Werkzeuge zur Verfügung stellt und eine starke Veränderung des Schriftstellerhandwerks in den nächsten Jahrzehnten nicht abzusehen ist, sind die Anforderungen an die Änderbarkeit, von Seiten der Benutzer sehr gering.

**Übertragbarkeit** Auch Anforderungen an Übertragbarkeit und Konformität sind relativ gering, da die Software von Heimnutzern und auf Personalcomputern genutzt wird, ist es vor allem notwendig die Software auf marktführenden Betriebssystemen laufen zu lassen.

## 5.3 Benutzer

Um Software zu entwickeln, die sinnvoll genutzt werden kann, ist es notwendig die Anforderungen auf den Kreis der Benutzer abzustimmen, dies ist nur durch direkten Kontakt zu dem gewünschten Benutzerkreis möglich. Da der Autor dieser Bachelorarbeit keinen spezifischen Kundenkreis hat, richtet er sich in verschiedenen Internetforen für Autoren an etwaige Nutzer von Unterstützungssystemen.

### 5.3.1 Aufbau der Benutzer und Kontakt

Da die Benutzer zu denen während der Analyse Kontakt aufgebaut wird, aktiv in Internetforen sind und an Befragungen zum Thema Unterstützungssysteme für Autoren teilnehmen, ist anzunehmen das diese Interesse an der Nutzung des Computers und anderer moderner Hilfsmittel zum Schreiben von Romanen benutzen wollen und vielleicht schon nutzen oder genutzt haben. Dadurch ist es anzunehmen, dass ein größerer Teil der dort gefundenen Benutzer Zustimmung zu neuen und computerbetonten Software äußern, als die Gesamtmenge der Autoren. Weiterhin deutet die Bereitschaft an einer Umfrage und Diskussionen zu diesem Thema teilzunehmen, auf eine höhere Bereitschaft hin eine solche Software in Erwägung zu ziehen.

### 5.3.2 Umfrageergebnisse

Grundsätzlich lässt sich aus der Art der Antworten und der weiteren Kommentare schließen, dass die meisten komplexeren Funktionen eher als belastend angesehen werden. Folgend werden die Daten ausgegeben die aus Fragen mit einer begrenzten Auswahlmöglichkeit entstanden sind. Werte über drei zeigen eine Akzeptanz der Benutzer, während werte unter drei eine durchschnittlich eher ablehnende Haltung anzeigen.

#### 5.3.2.1 Darstellung der Geschichte

**Q3P1.** Würdest du eine Software nutzen, in der Szenen zum Schreiben in einzelnen Fenstern dargestellt werden?

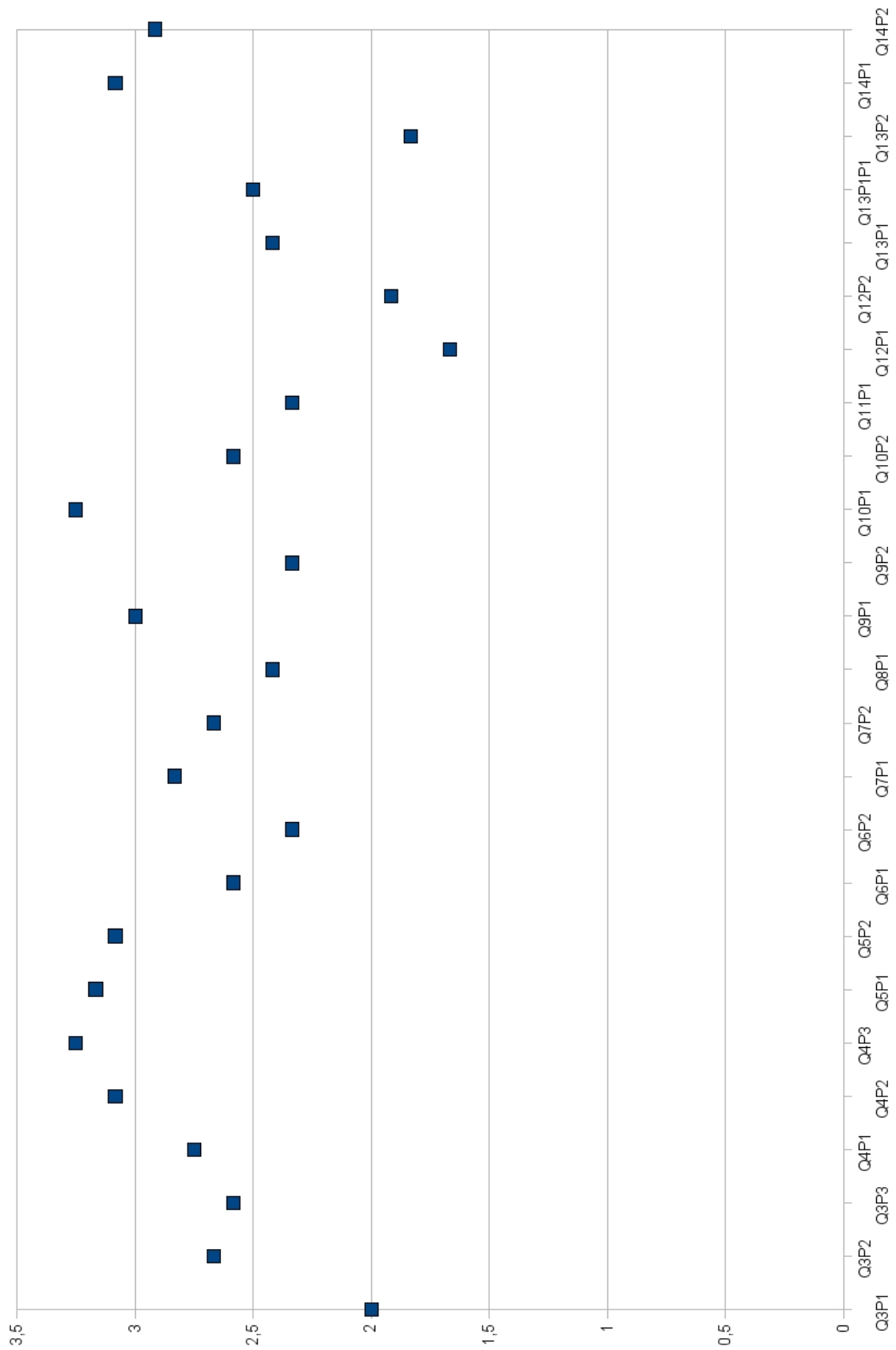


Abbildung 5.1: Ergebnisse der Umfrage, zur Bedeutung der Abkürzungen siehe Kapitel 5.3.2

**Q3P2.** Würdest du eine Software nutzen, in der Kapitel zum Schreiben in einzelnen Fenstern dargestellt werden?

**Q3P3.** Würdest du eine Software nutzen, in der die komplette Geschichte zum Schreiben in einem Fenstern dargestellt wird?

### **5.3.2.2 Notizfunktion und Informationsspeicherung**

**Q4P1.** Wäre für dich eine Funktion nutzbringend, die durch eine Tastenkombination ein Fenster für Notizen öffnet?

**Q4P2.** Würdest du Informationen wie Zusammenfassung, Ziel oder Funktionen zu Szenen innerhalb der Software abspeichern wollen?

**Q4P3.** Würdest du Informationen wie Zusammenfassung, Ziel oder Funktionen zu Kapiteln innerhalb der Software abspeichern wollen?

**Q5P1.** Wie sinnvoll siehst du eine Funktion an, in der verschiedene Informationen wie Name, Aussehen, Ziele oder Verhalten zu Charakteren niedergeschrieben werden können?

**Q5P2.** Würdest du die Personen die an einer Szene teilnehmen, und die aus deren Sicht es geschrieben wurde eingeben, um diese Informationen später wieder abzurufen, oder alle Szenen anzuzeigen an der eine bestimmte Person beteiligt ist?

**Q6P1.** Wie sinnvoll siehst du eine Funktion an, in der verschiedene Informationen wie Name, Aufbau, Funktion oder Beschreibung zu Orten niedergeschrieben werden können?

**Q6P2.** Würdest du den oder die Orte, an denen eine Szene spielt mit der Szene verbinden wollen, so dass du dir die Szenen, die an diesem Ort spielen, anzeigen lassen könntest?

**Q7P1.** Wie sinnvoll siehst du eine Funktion an, in der verschiedene Informationen wie Name, Beschreibung oder Funktion zu wichtigen Handlungsobjekten niedergeschrieben werden können?



**Q7P2.** Würdest du den oder die Objekte, die in einer Szene eine Rolle spielen mit der Szene verbinden wollen?

**Q8P1.** Würdest du Bilder, Audiodateien oder Videos zu den Charakter- Objekt, oder Ortsinformationen hinzufügen wollen?

### 5.3.2.3 Ausdruck

**Q9P1.** Für wie wichtig hältst du es, dass die Geschichte in verschiedenen Formaten ausdrückbar ist, zum Beispiel ein Format bei dem Platz für Anmerkungen durch Korrekturleser oder Redakteure ist?

**Q9P2.** Wie wichtig wäre es dir Listen von Orten, Objekten oder Personen ausdrucken zu können?

### 5.3.2.4 Statistiken

**Q10P1.** Würdest du eine Funktion nutzen, die Wörter oder Anschläge zählt und eine Statistik führt, um zu sehen wieviel du geschrieben hast?

**Q10P2.** Wenn Wörter oder Anschläge gezählt werden, würdest du dich darüber informieren lassen wollen, zum Beispiel durch ein Feld in der Statuszeile der Anwendung, dass du eine bestimmte Tagesquote erfüllt hast?

### 5.3.2.5 Planung

**Q11P1.** Würdest du eine eingebaute Funktion zum erstellen von Mind Maps nutzen?

### 5.3.2.6 Chronologie

**Q12P1.** Würdest du den Start- und Endzeitpunkt oder Dauer einer Szene, zum Beispiel in Stunden und Tagen, eingeben wollen, um dir eine chronologische Auflistung deiner Geschichte anzeigen zu lassen, die sich gegebenenfalls von der Reihenfolge des Auftauchens unterscheiden könnte? Eine Szene könnte am 1.12.2006 beginnen, oder am Tag 1 um 13:23 beginnen. Endzeitpunkt oder Dauer können dann in ähnlicher Weise eingegeben werden, das jeweils andere wird dann automatisch berechnet.

**Q12P2.** Wenn du die Informationen aus 4.1 eingeben würdest, würdest du dann bei der Verschiebung von Szenen in dieser Zeitskala darüber informiert werden wollen, welche Szenen zwischen dem alten Zeitpunkt und dem neuen Zeitpunkt liegen?

### 5.3.2.7 Kausalität

**Q13P1.** Würdest du Informationen über Beziehungen zwischen Personen und Objekten eingeben wollen, im Sinne von: Eine Person hat ein Objekt, eine Person kennt eine andere Person. Nach Eintragen von Änderungen während bestimmter Szenen könntest du dir so diese Beziehungen für jede beliebige Szene anzeigen lassen. Zum Beispiel: In Szene 1 verliert der Mörder die Mordwaffe am Tatort und der Kommissar nimmt die Mordwaffe in Szene 12 vom Tatort an sich. Du gibst ein das sich die Mordwaffe zu Beginn beim Mörder befindet, nach Szene 1 beim Tatort und nach Szene 12 beim Kommissar. Du könntest jetzt im Informationsfenster der Mordwaffe sehen, dass die Mordwaffe zuerst im Besitz des Mörders war, dann bis Szene 12 im Besitz des Tatorts und dann in den Folgenden Szenen beim Kommissar..

**Q13P1P1.** Würdest du, wenn du die Funktion aus 13.1. nutzen würdest, darüber informiert werden wollen, wenn du eine Szene über eine Beziehungsänderung der beteiligten Objekte, Personen oder Orte hinaus verschiebst oder Szenen erstellt werden, die den Beziehungen widersprechen könnten? Zum Beispiel: In Szene 14 untersucht der Kommissar die Mordwaffe, aus Gründen des Spannungsaufbaus wird diese Szene jedoch vor Szene 8 verschoben, das Programm würde jetzt warnen, dass der Kommissar die Mordwaffe erst in Szene 12 erhält. Ein weiteres Beispiel: Eine neue Szene wird eingefügt, in der der Kommissar auf der Straße am Mörder vorbei läuft, diese Szene wird eingefügt bevor die Beiden sich Kennenlernen, die Software erkennt an den Beteiligten Personen, dass zwei Personen beteiligt sind die sich noch nicht kennen und warnt deshalb den Autoren, obwohl dies natürlich nicht notwendig ist, da die Personen zwar in einer Szene vorkommen, aber nicht beteiligt sind

**Q13P2.** Würdest du die Funktionen die in 13.1. und 13.1.1. beschrieben sind nutzen, wenn nicht die Reihenfolge der Szenen ausschlaggebend für Warnungen wäre, sondern die unter 12.1. beschriebenen Zeitlichen Informationen?

### 5.3.2.8 Versionsspeicherung

**Q14P1.** Fändest du eine Funktion vorteilhaft, die in bestimmten Zeitabständen ein Backup erzeugt, so dass es möglich ist zu alten Versionen der Geschichte zurückzukehren?

**Q14P2.** Wenn die Funktion aus 14.1. vorhanden ist, fändest du eine Funktion sinnvoll die Unterschiede zwischen zwei Versionen hervorhebt?

### 5.3.3 Benutzerwünsche

Das grundlegende Bedürfnis der meisten Befragten ist es eine Software zu Benutzen, die es ihnen ermöglicht einfachen Text zu schreiben und zu verwalten. Über 70 Prozent der befragten benutzt keine Software die Speziell für Schriftsteller geschrieben ist, oder nutzt sie nur zur Verwaltung von Ideen oder Geschichten. Alternativ zur speziell für Schriftsteller entworfenen Software werden von einem Großteil der befragten Benutzer bekannte Textverarbeitungssoftware.

#### 5.3.3.1 Funktionale Anforderungen

Fast alle der über das eigentliche Schreiben hinausgehende Funktionen werde als nicht oder nur in geringem Umfang nutzbringend empfunden. Funktionen zur Verwaltung und Wiederherstellungsfunktionen und in geringerem Maße auch Informationen zur Speicherung von Notizen werden von vielen der Befragten als sinnvoll angesehen.

#### 5.3.3.2 Nichtfunktionale Anforderungen

Viele der befragten Autoren erwarten von einer Software vor allem eine einfache Bedienung und Erfahrungen bestehenden Produkten für Autoren werden oft als schlecht zu Bedienen oder ablenkend beschrieben. Gleichzeitig wird oft eine zu hohe Einschränkung im Arbeitsablauf dieser Produkte bemängelt.

## 5.4 Analyse bestehender Software

Der Autor fand auf der Suche nach bestehender Software zur Unterstützung von Autoren vier Produkte, die sich im Einsatz des Benutzerkreises befindet mit dem der Autor Kontakt hat. Zwei dieser Softwareprodukte sind "Freeware" Produkte die speziell für diesen Einsatz geschrieben wurden. "Karlsoft TeVeRo Step 2" ist ein deutsches Programm, das anscheinend nicht mehr weiterentwickelt wird, jedoch einige Funktionen bietet. yWriter4 ist eine Software des Autors Simon Haynes, dass er aus persönlicher Not heraus geschrieben hat und über eine Vielzahl an Funktionen verfügt. Writer's Café ist das einzige kommerzielle Tool das der Autor analysiert und verfügt über einen immensen Funktionsumfang der auf zwei einzelne

Programme aufgeteilt ist. Die vierte und letzte Software wurde eigentlich für den das schreiben von Drehbüchern entwickelt worden, verfügt jedoch über eine Vielzahl von Funktionen die auch von Autoren anderer Geschichten genutzt werden kann und ist flexibel genug auch das schreiben anderer Geschichtsformen zu unterstützen.

### 5.4.1 Vorgestellte Software

Die vier Softwareprodukte werden hier kurz mit ihren Features vorgestellt um an ihnen zu zeigen wie das Anforderungsprofil durch die Anforderungen bestehender Software verändert wurde.

#### 5.4.1.1 TeVeRo Step 2

TeVeRo Step 2 ist ein Programm das kostenlos verfügbar ist und von Matthias Kahlow entwickelt, inzwischen jedoch anscheinend nicht mehr weiterentwickelt wurde. Es ist das einzige Produkt das nur in Deutsch erhältlich ist. Es ähnelt am meisten von den vier Programmen einem Standardtextverarbeitungsprogramm und stellt keine Trennung von Kapiteln, Szenen oder ähnlichen Strukturen zur Verfügung. Das Programm stellt einen sehr übersichtlichen Funktionsumfang zur Verfügung, der die eigentliche Schreibfunktion unterstützt, anstatt sie zu benutzen.

**Features** Die funktionalen Features von TeVeRo Step 2 sind sehr begrenzt, doch besitzt es dadurch eine größtenteils einfache Bedienoberfläche.

**Handlungsübersicht** TeVeRo Step 2 speichert die Handlungsübersicht in einer stichwortartigen Liste, in der jeder Punkt einem Absatz zugeordnet werden kann. Die Liste kann während des Schreibens angezeigt werden und ermöglicht so eine schnelle Übersicht, sowohl über den Handlungsfaden, als auch über die Geschichte.

**Charaktereigenschaften** Die Software ermöglicht eine Speicherung von frei beschreibbaren Eigenschaften der Charaktere, und liefert hier Textfelder für Charaktereigenschaften und Aussehen. Beschreibungen können, zum Zwecke des Vergleiches, nebeneinander dargestellt werden.

**Wortzählung** Worte und Anschläge pro Tag werden von der Software überwacht und es wird ermittelt ob ein frei wählbares Mindestpensum an Anschlägen erreicht wurde und wie die Tendenz der gemachten Anschläge zum Vortag ist. Dies soll dem Autoren helfen dein Fortschritt seiner Arbeit zu überwachen.

**Fazit** TeVeRo Step 2 stellt eine einfache Oberfläche zur Verfügung, die sich auf das eigentliche Schreibfeld konzentriert, weitergehenden Funktionen sind, wie die Storyübersicht, entweder an den Seiten des Fensters zu finden, oder über Kontextmenüs zu erreichen. Dadurch rückt bei diesem Programm das Schreiben selbst in den Vordergrund, jedoch macht die schlechte Erreichbarkeit der Charakterinformationen den Nutzen der Funktion eher fragwürdig.

#### 5.4.1.2 yWriter 4

Der yWriter 4 ist ein kostenlos nutzbares Programm von Spacejock Software, dass laut Aussage des Entwicklers wie eine Programmierumgebung aufgebaut ist. Die Szenen sind einzelne Dateien, die beim Drucken oder Exportieren zusammengefügt werden. Die Geschichten werden in Projekten gespeichert, die sich als Ordnerstrukturen Speichern lassen. Der yWriter bietet durch seine gute Strukturierung eine gute Übersicht über die Kapitel und Szenen und erlaubt es einem Autoren viele Informationen zu speichern. Leider wird es durch diese enorme Anzahl an Möglichkeiten sehr unübersichtlich, und viel Platz auf dem Bildschirm wird von diesen Punkten eingenommen. Außerdem bietet es nur wenige Features, die die eingegebenen Informationen ausnutzen.

**Features** yWriter 4 verfügt über eine Vielzahl von Eingabemöglichkeiten, dessen Informationen jedoch wenig genutzt werden.

**Darstellung von Szenenabfolge als Storyboard** Der yWriter kann Szenen in einem graphischen Storyboard darstellen, in der es möglich ist die Szenen, geordnet nach Reflektorfür, also der Person aus dessen Perspektive diese Szene geschrieben ist, zu sehen und anzuordnen.

**Erzählperspektive** Jeder Szene kann ein Charakter zugewiesen werden, aus dessen Sicht die Szene geschrieben ist und weitere Charaktere, die in dieser Szene vorkommen, damit es möglich ist eine schnelle Übersicht über die vorkommenden Charaktere zu gewinnen. Leider ist eine umgekehrte Suche, also alle Szenen in denen ein Charakter vorkommt oder aus dessen Perspektive sie geschrieben ist, nicht möglich.

**Orte und Objekte** Ähnlich wie Charaktere zu Szenen zugewiesen werden können, können Szenen an einem oder mehreren Orten spielen und Objekte können in ihr vorkommen und so wie bei den Charakteren ist es hier mit einem Mausklick möglich gespeicherte Informationen über Orte und Objekte zu betrachten. Die umgekehrte Suche nach allen Szenen die an einem Ort, oder mit einem Objekt spielen, ist nicht möglich.

**Automatisierte Aufgabenlisten** Da jeder Szene ein Status, wie "erste Revision" oder "Fertig", zugewiesen werden kann, ist das Programm in der Lage anzuzeigen welche Arbeit noch getan werden muss, und in welchem Zustand sich das komplette Werk befindet.

**Szenekarten, Listen von Orten, Objekte und Charakteren druckbar** Es wird die Möglichkeit geboten Szenekarten, die Kurzbeschreibungen und Beschreibungen von Szenen beinhalten, falls solche geschrieben wurden, zu drucken und es damit erleichtern die Übersicht über die Geschichte zu behalten. Eine ähnliche Funktion existiert auch für Orte und Objekte, jedoch wird hier nur eine Liste der Name erzeugt, deren Beschreibungen unterdrückt wird.

**Druckfunktion** Das Programm verfügt über verschieden Druckfunktionen, die auf die spezielle Nutzung des Ausdrucks abgestimmt sind, als fertiges Buch, einen vollständigen Druck mit allen Informationen, oder einen Ausdruck der Platz für Anmerkungen eines Editors bietet.

**Bewertung von Szenen** Das Programm bietet die Möglichkeit Szenen nach frei definierbaren Kriterien durch Zahlen zu Bewerten, zum Beispiel Spannung, und kann eine Kurve generieren, durch die es dem Autoren möglich ist eine Übersicht über den Verlauf seiner Geschichte zu erhalten.

**Wortzählung** Es wird eine Statistik über die Anzahl der geschriebenen Wörter geführt, und die Möglichkeit ein tägliches Minimum festzulegen, von Wörter die geschrieben werden müssen, dessen Einhaltung vom Programm überwacht wird. Dadurch kann der Autor seinen fortschritt überwachen.

**Problematische Wörter** Dem Autor ist es möglich Wörter, die als problematisch empfunden werden, zu speichern. Diese werden dann vom Programm gemeldet, falls sie von ihm genutzt werden.

**Fazit** Der yWriter kann mit einer Vielzahl von Funktionen, die vor allem für professionelle Schriftsteller interessant sind, aufwarten und stellt eine schlichte Oberfläche zur Verfügung die oft jedoch an einer Vielzahl von Eingabefeldern leidet. Die Möglichkeit große Mengen an Informationen über die Geschichte zu speichern kann einem Autor zwar sehr helfen, lässt jedoch das Gefühl zurück das viele der eingegeben Daten ungenutzt bleiben und es oft übersichtlicher erscheint diese Informationen gesondert niederzuschreiben. Für den beruflichen Einsatz sind jedoch die vielfältigen Druckfunktionen und die Statistiken ein unverzichtbares Hilfsmittel um schnell mit dem Verleger kommunizieren zu können und den Fortschritt des eigenen Buches zu überblicken.

#### 5.4.1.3 Celtx

Celtx ist ein Programm zur Unterstützung der Pre-Produktion von Theaterstücken, Filmen oder andere Medien, das unter der Celtx Public License, einer leicht abgewandelten Form der Mozilla Public License, veröffentlicht ist. Der eigentliche Verwendungszweck der Software liegt zwar nicht im erstellen von Romane, jedoch stellt es auch für diesen Zweck einige Werkzeuge zur Verfügung.

**Features** Celtx verfügt über eine klare und graphisch orientierte Benutzeroberfläche und viele Funktionen die vor allem Schriftstellern mit einem plastischen Schriftstil zu gute kommen.

**Werkzeuge zur Entwicklung des Handlungsfadens** Die Software verfügt über die Möglichkeit zur Erstellung eines graphischen Storyboards, wie es von Film- und Fernsehenproduktionen bekannt ist. Hierzu können Bilder in das Programm geladen werden, mit Beschreibungen versehen und beliebig angeordnet werden. Zusätzlich können die Bilder mit ihren Beschreibungen abgespielt werden.

**Informationen über Charaktere, Orte, Szenen** Auch Celtx verfügt über die Möglichkeit Informationen über Charaktere zu speichern, hierbei stellt es, zusätzlich zu einem allgemeinen Notizbereich eine große Zahl von Feldern in denen Informationen über Verhalten, Charakter und Aussehen von Charakteren gespeichert werden. Eine ähnliche Vielfalt von Informationen ist auch über die Orte und Szenen speicherbar.

**Medienunterstützung** Da Celtx für die Erstellung von Filmen, Serien und Theaterstücken gedacht ist, ist es in der Lage verschiedene multimediale Dateiformate zu nutzen. Zu den meisten Informationen, wie Charakteren, Szenen und Orten lassen sich Medien hinzufügen. Bilddateien der meisten gängigen Formate werden innerhalb des Programms angezeigt, Audio und Videodateien werden mit den dafür vorgesehenen Programmen geöffnet.

**Kollaborations- und Backup-System** Das Celtx-Projektteam stellt eine Online Umgebung zur Verfügung auf der Autoren ihre Arbeit speichern können. Da es Möglich ist den Zugriff auf die eigenen Celtx Projekte zu Beschränken, kann dieser Ort sowohl zur Datensicherung als auch zur Kollaboration mit anderen Autoren oder zur Veröffentlichung genutzt werden.

**Fazit** Auch wenn Celtx nicht für das schreiben von Romanen gedacht ist, sind viele der Funktionen sehr ähnlich zu den speziell für Autoren geschriebener Programme. Zwar sind viele der Werkzeuge von Celtx sehr graphisch, dies kann jedoch sehr hilfreich für Autoren sein die sehr graphisch denken oder einen plastischen Schreibstil haben. Auch kann es natürlich Autoren helfen, die über real existierende Personen oder Orte schreiben und so Photos und Karten direkt speichern können. Als einziger Nachteil ist eine fehlende Verknüpfung zwischen den Geschichten und dem Planungswerkzeug mit den Szenen zu sehen, der dadurch entsteht das Celtx seine Hauptfunktionen nicht in diesem Bereich hat.

#### 5.4.1.4 Writer's Café

Writer's Café ist das einzige kommerzielle Tool, das im Zuge dieser Bachelorarbeit auf seine Merkmale untersucht wurde. Es verfügt über eine verspielte graphische Oberfläche, die versucht Notizblöcke und Ebenholzuntergrund nachzuahmen. Weiterhin verfügt es über eine Anzahl von Möglichkeiten um den Kopf frei zu bringen, wie eine Solitärevariante, Zitate über das Schreiben oder ein Programm das Stichworte liefert, über die ein Autor schreiben kann. Da diese Funktionen jedoch auf modernen Computern mit Internetanschluss leicht nachzuahmen sind und der Nachweis des Nutzens dieser Funktionen den Rahmen dieser Bachelorarbeit sprengen würde, werden sie nicht als Features aufgeführt.

**Features** Writer's Café verfügt über die größte Bandbreite von Funktionen, besitzt dadurch jedoch auch die komplexeste Bedienoberfläche.



**Storylines** Writer's Café nutzt Karten um Szenen darzustellen, die in mehreren Handlungsfäden aufgeteilt werden können. Ähnlich wie im Bereits beschriebenen yWriter können auch bei Writer's Café ein Schauplatz und an der Szene beteiligte Personen ausgewählt werden. Weiterhin können auch Beschreibungen und ein Bild das die Szene darstellen soll hinzugefügt werden.

**Charakter- und Ortsinformationen** Writer's Café speichert Informationen über Charaktere und Schauplätze in vorgefertigten Karten, die Platz für die meisten wichtigen Informationen lassen und ein Notizfeld für weitere Informationen.

**Scrapbook** Zur Planung von Handlungen und zum Entwickeln von Ideen, verfügt Writer's Café neben einem Notizbuch auch über ein Mind Mapping Programm, das Bilder und Mindmap-Blasen in verschiedenen Formen und Farben unterstützt.

**Fazit** Writer's Café verfügt über viele nützliche Funktionen, die vor allem etwas ausgereifter wirken als bei yWriter oder TeVeRo Step 2. Die Handhabung ist in vielen Fällen intuitiv und ist relativ schnell. Andererseits ist das schwere Auffinden von bestimmten Einstellungen und das sehr kleine Textfeld zum schreiben der eigentlichen Szenen eher hinderlich.

## 5.4.2 Fazit

Die vier getesteten Programme verfolgen vier vollkommen unterschiedliche Ansätze und doch haben die Vier Programme viele Ähnlichkeiten in ihren Funktionen. Alle bieten die Möglichkeit Informationen über Storyline und Charaktere zu speichern und so die Funktion von Notizbüchern zu digitalisieren, damit es einem am Computer arbeitendem Autoren einfacher ist seine Arbeit durchzuführen. Das einzige Dateiformat, das von allen Programmen unterstützt wird, ist das Rich Text Format, doch verfügen vor allem die noch aktiv entwickelten Varianten über  $\LaTeX$  oder andere einfache Wege die Dokumente auch in andere Formate wie das Portable Document Format von Adobe Systems zu überführen.

### 5.4.2.1 Wichtige Features

Aus den Features, die die verschiedenen Softwareprodukte zur Verfügung stellen, hat der Autor anhand von Häufigkeit des Vorkommens und Erfahrungen mit den Funktionen die wichtigsten Ausgewählt um sie als weitere Features zu übernehmen.

**Funktionale Features** Die Bandbreiten der funktionalen Features der einzelnen Softwaresysteme ist sehr unterschiedlich, während einige nur ein sehr geringes Maß an Funktionen anbieten, die über das eigentliche Schreiben hinausgeht bieten andere Softwareprodukte ein großes Maß an Funktionen die den Autor bei der Findung von Ideen oder bei der Strukturierung der Geschichte unterstützen sollen. Hier aufgeführt sind die Features die der größte Teil der Softwaresysteme zur Verfügung stellen und damit von vielen unterschiedlichen Herstellern als wichtig angesehen werden.

Tabelle 5.3: Eine Liste der wichtigen funktionalen Anforderungen bestehender Software

| ID | Name                   | Priorität | Schätzung | Use Case   | Notizen |
|----|------------------------|-----------|-----------|--|---------|
| 4  | Notizfunktion          | x         | -         | Der Benutzer erstellt per Tastenkombination oder Mausklick auf ein Icon eine Neue Notiz, die sich zum Schreiben öffnet. Siehe auch Use Case "Notizfunktion"  |         |
| 5  | Charakterinformationen | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Charakter aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Charakterinformationen" |         |
| 6  | Ortsinformationen      | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Ort aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Ortsinformationen"            |         |

Fortsetzung auf nächster Seite...

Tabelle 5.3 – Fortgesetzt

| ID | Name                      | Priorität | Schätzung | Use Case  | Notizen |
|----|---------------------------|-----------|-----------|---|---------|
| 7  | Objektinformationen       | x         | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Objekt aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Objektinformationen"  |         |
| 14 | Wortzählung und Statistik | x         | -         | Der Benutzer will seinen Fortschritt erfahren und öffnet über das "Datei"-Menü das Statistikfenster. Das Fenster zeigt die insgesamt Anzahl von Wörtern und die Anzahl von Worten und Anschlägen die an diesem Tag geschrieben wurden. Siehe dazu auch Use Case "Wortzählung und Statistik"   |         |
| 15 | Druckfunktion             | x         | -         | Der Benutzer möchte die Informationen, die er über Charaktere, Orte, Objekte oder Szenen eingegeben hat ausdrucken und öffnet das erweiterte Druckmenü. Dort Charakterinformationen, Orte, Objekte oder Szenen aus und eine automatisch formatierte Liste der ausgewählten Einträge wird gedruckt. Siehe dazu auch Use Case "Wortzählung und Statistik" |         |

Fortsetzung auf nächster Seite...

Tabelle 5.3 – Fortgesetzt

| ID | Name             | Priorität | Schätzung | Use Case   | Notizen |
|----|------------------|-----------|-----------|--|---------|
| 16 | Planungsfunktion | x         | -         | Funktion um es dem Benutzer zu ermöglichen eine Story zu planen. |         |

**Wortzählung und Statistik** Drei der vier getesteten Programme verfügten über eine Möglichkeit der Wörterzählung und der Statistik über den eigenen Fortschritt. Da natürlich für professionelle Autoren von der Anzahl der verkauften, und damit natürlich auch von der Anzahl der veröffentlichten, Bücher der Lebensunterhalt abhängt, stellt diese Funktion für sie natürlich einen Indikator ihres eigenen Fortschrittes und der Arbeitsleistung da. Gerade wenn nicht nur die Länge der Geschichte, sondern auch Anschläge, und damit Veränderungen, gezählt werden

**Storyplanung** Alle getesteten Programme unterstützten in irgendeiner Form die Planung des Handlungsfadens. Zwar benutzen nicht alle Autoren eine solche Funktion, aber viele Quellen empfehlen das so genannte "plotting". Die Unterstützung dieses Verhaltens sollte also auch im Funktionsumfang jeder Software vorhanden sein, die sich nicht nur an einen speziellen Kreis von Autoren wendet.

**Charakter-, Szenen und Ortsinformationen** Die vier getesteten Programme unterstützen die Speicherung von Informationen über Charaktere, Szenen und Orte, dies scheint somit eine wichtige Funktion und unterstützt den Autoren, da auf diese Weise wichtige Informationen an einer Stelle schnell erreichbar gespeichert werden, was sowohl den Schreibprozess als auch Überarbeitungen erleichtert.

**Druckfunktion für Szenekarten, Listen von Orten, Objekte und Charakteren** Zwar ist diese Funktion nur in einem Programm vorhanden, jedoch ist es, aus persönlicher Erfahrung des Autoren, oft besser einen Ausdruck als physische Repräsentation in der Hand zu haben, als alle Daten nur in digitaler Form vorliegen zu haben. Da das Vorhandensein der notwendigen Daten bereits gefordert ist, ist diese Funktion mit geringen Aufwand zu erhalten.

**Nichtfunktionale Features** Die prioritäten nichtfunktionaler Features sind nur schwer aus bestehender Software zu entnehmen, jedoch lassen sich durch sie etwaige Anforderungen erkennen, wenn die Software diese nicht erfüllt.

**Einfache Erreichbarkeit von Informationen** Dem Autoren ist bei der Untersuchung der vorhandenen Software an mehreren Stellen aufgefallen, dass viele Schritte notwendig sind um Informationen über Charaktere und Orte zu erhalten und so der kreative Schreibprozess unterbrochen werden muss. Für einen Autoren wäre es jedoch wünschenswert schnell Notizen über diese Themen schreiben und Aufrufen zu können und einfachen Zugriff auf sie zu haben.

## 5.5 Analyse der benötigten Software

Welche Software benötigt wird, hängt vor allem von den Wünschen der Benutzer und der Beschränkungen durch Zeit und Fähigkeiten der Entwickler und nicht zuletzt der Beschränkungen der modernen Computer ab. Daher ist es nicht verwunderlich, dass die Anforderungen die aus der Benutzerumfrage entstanden sind, hier die höchsten Prioritäten erhalten, während andere niedrigere Prioritäten erhalten, oder ganz aus der Liste der Anforderungen verschwinden.

### 5.5.1 Features

Die Anforderungen die an ein Unterstützungssystem gestellt werden gehen sehr weit auseinander, da einige Benutzer nur die nötigsten Funktionen nutzen wollen, während andere offen für sehr weitreichende und Aufwendige Features sind. Daher muss ein Kompromiss zwischen den einzelnen Standpunkten gefunden werden, der darin liegt grundlegende Features die von einem Großteil der Benutzer erwünscht sind zuerst zu implementieren und weiterführende Funktionen als unwichtiger einzustufen und so in die Software einzubinden, dass sie deaktiviert werden können oder erst aktiviert werden müssen.

### 5.5.2 Funktionale Features

Die Anforderungen, die benötigte Software erfüllen muss, richtet sich nach den Wünschen der späteren Benutzer und unterscheidet sich überraschenderweise sehr stark von den Anforderungsprofilen der meisten getesteten Software Produkte. Den befragten Benutzern genügte eine Erfüllung minimaler funktionaler Anforderungen, da sie die Probleme vor allem in der Bedienbarkeit der Software sahen und nur grundlegende Funktionalität die das schreiben ermöglichte brauchten.

Tabelle 5.4: Eine Liste der funktionalen Anforderungen, die von den Autoren gefordert werden.

| ID | Name                          | Priorität | Schätzung | Use Case  | Notizen |
|----|-------------------------------|-----------|-----------|---|---------|
| 18 | Teilung in Bücher und Kapitel | 170       | -         | Benutzer erstellt ein neues Buch und wird durch einen Dialog geführt, in dem er Buchname, Art[Kurzgeschichte, Roman, Erzählung] und Genre einträgt. Am Ende des Dialogs wird ein namenloses Kapitel erstellt und geöffnet in dem der Autor die Geschichte schreiben kann. Siehe such Use Case "Teilung in Bücher und Kapitel" |         |
| 14 | Wortzählung und Statistik     | 160       | -         | Der Benutzer will seinen Fortschritt erfahren und öffnet über das "Datei"-Menü das Statistikfenster. Das Fenster zeigt die insgesamt Anzahl von Wörtern und die Anzahl von Worten und Anschlägen die an diesem Tag geschrieben wurden. Siehe dazu auch Use Case "Wortzählung und Statistik"                                   |         |
| 19 | Versionskontrolle             | 150       | -         | Auf Befehl des Benutzers oder in vom Benutzer definierten Zeitabständen wird eine vollständige Kopie des geöffneten Buches gemacht, die mit Namen und einer automatisch generierten Nummer gespeichert wird. Siehe dazu auch Use Case "Versionskontrolle"   |         |

Fortsetzung auf nächster Seite...

Tabelle 5.4 – Fortgesetzt

| ID | Name   | Priorität | Schätzung | Use Case  | Notizen |
|----|--|-----------|-----------|---|---------|
| 4  | Notizfunktion                                  | 140       | -         | Der Benutzer erstellt per Tastenkombination oder Mausklick auf ein Icon eine Neue Notiz, die sich zum Schreiben öffnet. Siehe auch Use Case "Notizfunktion"   |         |
| 9  | Automatische Formatierung                      | 130       | -         | Der Benutzer druckt seine Geschichte aus. Überschriften, Hervorhebungen Zeilen- und Seitenumbrüche werden automatisch von der Software erzeugt. Siehe dazu auch Use Case "Automatische Formatierung"  |         |
| 11 | Anpassung der Formatierung durch Einstellungen | 120       | -         | Der Benutzer geht in das Optionsmenü und verändert durch Auswahl von Schriftart, Schriftgröße, Zeilenabstand und Papierformat die Formatierung. Siehe dazu auch Use Case "Anpassung der Formatierung durch Einstellungen"   |         |
| 15 | Druckfunktion                                  | 110       | -         | Der Benutzer möchte die Informationen, die er über Charaktere, Orte, Objekte oder Szenen eingegeben hat ausdrucken und öffnet das erweiterte Druckmenü. Dort Charakterinformationen, Orte, Objekte oder Szenen aus und eine automatisch formatierte Liste der ausgewählten Einträge wird gedruckt. Siehe dazu auch Use Case "Wortzählung und Statistik" |         |

Fortsetzung auf nächster Seite...

Tabelle 5.4 – Fortgesetzt

| ID | Name                           | Priorität | Schätzung | Use Case   | Notizen |
|----|--------------------------------|-----------|-----------|--|---------|
| 5  | Charakter-<br>informationen    | 100       | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Charakter aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Charakterinformationen" |         |
| 7  | Objektinformationen            | 90        | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Objekt aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Objektinformationen"       |         |
| 8  | Beziehungen                    | 80        | -         | Der Benutzer trägt eine Beziehungsänderung zwischen zwei zuvor erzeugten Objekten oder Personen in einer Szene über das Optionsmenü ein. Siehe dazu auch Use Case "Beziehungen"  |         |
| 17 | Multimediateien<br>und Notizen | 70        | -         | Hinzufügen und entfernen von Multimediateien zu Notizen Sie auch Use Case "Multimediateien und Notizen".   |         |

Fortsetzung auf nächster Seite...



Tabelle 5.4 – Fortgesetzt

| ID | Name                                      | Priorität | Schätzung | Use Case  | Notizen |
|----|---|-----------|-----------|---|---------|
| 6  | Ortsinformationen                         | 60        | -         | Der Benutzer markiert ein oder mehrere Worte und wählt über das Kontextmenü Ort aus, daraufhin öffnet sich ein Dialogfenster in dem der Benutzer Name, alternative Namen und weitere Informationen eintragen kann. Siehe dazu auch Use Case "Ortsinformationen"   |         |
| 10 | Anpassung der Formatierung durch $\LaTeX$ | 50        | -         | Der Benutzer geht in das Optionsmenü und verändert durch Eingabe von $\LaTeX$ Befehlen die Formatierung. Siehe dazu auch Use Case "Anpassung der Formatierung durch $\LaTeX$ "  |         |
| 13 | Objekte, Personen und Kausalität          | 40        | -         | Der Benutzer verschiebt eine Szene in der Person/Objekt A und Person/Objekt B vorkommen vor eine Szene in der die Verbindung zwischen Person/Objekt A und Person/Objekt B verändert wird. Der Benutzer wird dann gewarnt und die Szene in der diese Beziehungsveränderung getan wird, wird angezeigt. Siehe dazu auch Use Case "Objekte, Personen und Kausalität" |         |
| 16 | Planungsfunktion                          | x         | 30        | Funktion um es dem Benutzer zu ermöglichen eine Story zu planen.  |         |

Fortsetzung auf nächster Seite...

Tabelle 5.4 – Fortgesetzt

| ID | Name   | Priorität | Schätzung | Use Case  | Notizen |
|----|--|-----------|-----------|---|---------|
| 12 | Überprüfungsfunktion für zeitliche Richtigkeit | 20        | -         | Der Benutzer verschiebt eine Szene vor eine Szene die mit beide mit Zeitinformationen versehen sind, die verschobene Szene ist auf einen Späteren Zeitpunkt geplant als die Szene über die hinweg verschoben wird, daher warnt die Software mit einem Dialogfenster und zeigt die nicht verschobene Szene an. Siehe dazu auch Use Case "Überprüfungsfunktion für zeitliche Richtigkeit" |         |
| 3  | Zeitskala                                      | 10        | -         | Benutzer versieht seine Szenen mit Start- und Endzeitpunkten und ruft die Zeitskala Funktion auf. Ein Fenster öffnet sich in dem die Szenennamen oder die ersten Worte einer Szene in Kästen in chronologischer Reihenfolge angezeigt werden. Siehe auch Use Case "Zeitskala"   |         |

### 5.5.3 Nichtfunktionale Features

Die Aussagen der Benutzer und die Erfahrungen des Autors mit den bestehenden Softwareprodukten haben schnell zu der Erkenntnis geführt, dass es vor allem notwendig ist eine Software zu erstellen mit der unbedarfte Nutzer von Computern schnell in der Lage sind umzugehen und die keinerlei Hürden während des eigentlichen Schreibprozesses dem Autoren in den Weg stellt.

#### 5.5.4 Notwendigkeit eines weiteren Produktes

Die Notwendigkeit einer weiteren Software hängt vor allem von dem Erfüllungsgrad der Anforderungen durch etablierte Programme ab. Die meisten funktionalen Anforderungen werden wie in Tabelle 5.5 gezeigt von allen Programmen erfüllt, und vielen Anforderungen die nicht benötigt werden, werden noch zusätzlich geliefert. Dies weist zwar darauf hin, dass es nicht notwendig ist eine Software zu schreiben. Jedoch weisen die Aussagen der Benutzer und die überwiegend negative Haltung zu erweiterten Funktionen und Erfahrungsberichte der Benutzer daraufhin, dass die Nichterfüllung der nicht-funktionalen Anforderungen für viele Benutzer diese Softwaresysteme weniger attraktiv macht, als ein System mit weniger Funktionen. Hierbei stellt vor allem die hohe Komplexität und damit der Arbeitsaufwand während des Schreibens selbst eine Hürde für die Benutzung dieser Softwaresysteme dar. Ein Softwareprodukt, das die Forderungen der Benutzer erfüllt sollte also nicht wie die Vision des Autoren ein System mit hoher Funktionalität sein, sondern ein System, das nur Kernfunktionalitäten zur Ver-

fügung stellt jedoch eine schnelle und unkomplizierte Bedienung zur Verfügung stellt.

Tabelle 5.5: Ein Vergleich der getesteten Softwaresysteme mit der Vision des Autors und den Ergebnissen der Analyse

| Feature   | Benötigt | Vision | TeVeRo Step 2  | yWriter 4 | Celtx          | Writer's Café |
|---|----------|--------|----------------|-----------|----------------|---------------|
| Teilung Bücher, Kapitel und Szenen (18)               | K        | S      | B              | S         | G              | S             |
| Wortzählung und Statistik (14)                        | x        | x      | x              | x         | x              | x             |
| Notizfunktion (4)                                     | x        | x      | x <sup>a</sup> | x         | x              | x             |
| Automatische Formatierung (9)                         | x        | x      | x              | x         | x              | S             |
| Anpassung der Formatierung durch Einstellungen (11)   | x        | x      | x              | x         | x              | x             |
| Druckfunktion (15)                                    | x        | x      |                | x         |                |               |
| Charakterinformationen (5)                            | x        | x      | x              | x         | x              | x             |
| Objektinformationen (7)                               |          | x      |                | x         | x              |               |
| Beziehungen (8)                                       |          | x      |                |           | x <sup>b</sup> |               |
| Multimediateien und Notizen (17)                      |          |        |                |           | x              |               |
| Ortsinformationen (6)                                 |          | x      |                | x         | x              | x             |
| Anpassung der Formatierung durch $\text{\LaTeX}$ (10) |          | x      |                |           | x              |               |
| Objekte, Personen und Kausalität (13)                 |          | x      |                |           |                |               |
| Planungsfunktion (16)                                 |          | x      |                | x         | x              | x             |
| Überprüfungsfunktion für zeitliche Richtigkeit (12)   |          | x      | x              |           |                |               |
| Zeitskala (3)   |          | x      |                |           |                |               |

Planungsfunktion kann für Notizen genutzt werden.

Mind Map kann für die Darstellung von Beziehungen genutzt werden

### 5.5.5 Das Unterstützungssystem für Autoren

Die momentane Ausprägungen der Anforderungen, das bisherige Ergebnis des Entwicklungsprozesses, legen eine Erweiterung zu einer standard Bürosoftware nahe, da sie eine

geringe Einarbeitungszeit und damit eine geringe Hürde bietet für Personen die bereits mit dieser Bürosoftware arbeiten. Die Auswertung der Umfrage hat ergeben, dass die meisten Autoren mit OpenOffice arbeiten, daher sollte die Erweiterung auf diese Software aufbauen.

#### 5.5.5.1 Anforderungen

In den ersten Versionen der Software sollen die grundlegenden Funktionen der Software soweit erfüllt werden, dass es möglich ist weitere Anforderungen, oder Veränderungen der bestehenden Abzuändern. Hierzu muss ein Benutzer die Funktionalität und Bedienung einer Software schnellstmöglich durchschauen können, um dann über ihren Nutzen zu entscheiden. Diese erste Version der Software sollte vor allem auch explorative Funktion haben.

**Funktionale Anforderungen** Die wichtigsten Funktionalen Anforderungen, die bereits in den vorangegangenen Kapiteln beschrieben wurden, gehen nur wenig über die Funktionalität des eigentlichen Schreibens hinaus, daher erfüllt die geplante Software die meisten geforderten Funktionalitäten ohne Einschränkung.

**Teilung Bücher, Kapitel und Szenen** OpenOffice benutzt Dokumente zur Darstellung geschriebener Texte, diese können in einem Zentraldokument verlinkt werden, so ist es möglich, eine Einteilung nach Kapiteln und oder Szenen zu erlauben. Je nach Wunsch des Autoren bietet es so die größte Anpassbarkeit.

**Wortzählung und Statistik** OpenOffice überwacht selbstständig die Anzahl von Wörtern, Zeilen, Zeichen und Absätzen innerhalb eines Dokumentes. Dadurch stellt es die meistgeforderte statistische Auswertung selbstständig zur Verfügung. Weitere Statistiken, wie Tagespensum oder Vergleiche mit der Vergangenheit sind momentan nicht möglich und müssten durch Erweiterungen nachgereicht werden.

**Versionskontrolle** OpenOffice verfügt über eine eingebaute Art der Versionskontrolle, die es ermöglicht alte Versionen wiederherzustellen. Zusätzlich zu dieser Funktionalität soll die Erweiterung eine Möglichkeit zur Datensicherung bieten, um einem Autoren das höchste Maß an Sicherheit und Komfortabilität zu bieten

**Notizfunktion** Hier bietet OpenOffice ebenfalls bestehende Funktionalität, mit der es möglich ist Notizen zu erzeugen die beim Druck, oder bei der Umwandlung in das PDF Format nicht auftauchen.

**Automatische Formatierung** In einem geringen Maße bietet OpenOffice auch die Möglichkeit der automatischen Formatierung, zwar sind leichte Mängel bei Zeilen- und Seitenumbrüchen zu erkennen, diese tauchen jedoch bei allen geprüften Produkten auf.

**Anpassung der Formatierung durch Einstellungen** OpenOffice verfügt über eine große Anzahl von Einstellungsmöglichkeiten, die über die Funktionalität einiger der getesteten Autorentensysteme hinausgeht.

**Druckfunktion** OpenOffice stellt nur ein geringes Maß an verschiedenen Druckfunktionen zur Verfügung, diese Funktionen empfindet der überwiegende Teil der Benutzer jedoch als Ausreichend. Daher wird zumindest die erste Version der Erweiterung keinerlei zusätzliche Funktionalität erzeugen.

**Charakterinformationen** OpenOffice bietet keine Funktion mit der es möglich ist vorgegebene Charakterinformationen einfach zu speichern, diese Schwäche, aus Sicht eines Autors, kann zwar durchaus auch ohne Probleme umgangen werden, in dem er ein weiteres Dokument erstellt zum Zwecke der Speicherung dieser Informationen, wird jedoch durch die Erweiterung abgenommen werden. Zu Demonstrationszwecken wird diese Funktion zuerst nur Name und Beschreibung der Charaktere speichern, Erweiterungen dieser Funktion werden in späteren Versionen zur Verfügung stehen.

**Nichtfunktionale Anforderungen** Dadurch, dass die meisten Benutzer bereits Erfahrungen mit OpenOffice oder einer ähnlichen Software haben, ist es deutlich einfacher die nicht-funktionalen Anforderungen zu erfüllen, da die Bedienbarkeit, das eigentliche Manko der existierenden Produkte, bereits durch die Grundkenntnisse der Benutzer abgedeckt wird und sie nur noch den Umgang mit der Erweiterung erlernen müssen.

**Zuverlässigkeit** Die Anforderungen an die Zuverlässigkeit des Produktes wurden schon zu Beginn korrekt eingeschätzt und werden von OpenOffice erfüllt, durch die ausgereifte Programmierschnittstelle für Erweiterungen ist es nicht anzunehmen, dass diese die Stabilität sehr beeinflussen wird.

**Benutzbarkeit** Zwar besitzt OpenOffice eine hohe Einarbeitungszeit um die Funktionen ausreichend bedienen zu können um längere Geschichte zu schreiben. Jedoch verfügt bereits der Großteil des Benutzerkreises auf den dieses Produkt abzielt über die notwendigen Kenntnisse, so dass nur geringer Aufwand für diese Personen besteht um den Umgang mit der Erweiterung zu lernen.

**Effizienz** Die von OpenOffice benötigten Systemressourcen werden von den Benutzern als angemessen angesehen, da diese Software sich vielfach bereits im Einsatz befindet. Die Auswirkungen auf das Verhalten der Software, die von der Erweiterung ausgeht, müssen so gestaltet werden, dass sie kaum merklich sind.

**Änderbarkeit** Die Änderbarkeit ist durch die Programmierschnittstelle und die offene Form der Programmierung von OpenOffice deutlich ausgeprägter als von den Benutzern wahrscheinlich benötigt wird.

**Übertragbarkeit** Die Übertragbarkeit wird durch die Grundfunktionalität von OpenOffice die Anforderungen weit übersteigen, da die Erweiterung in einer plattformunabhängigen Sprache entwickelt werden kann und nur wenige bis keine Veränderungen für das Portieren in andere Umgebungen notwendig ist

## 6 Evaluierung von SoDa

Während der Evaluierung des SoDa Entwicklungsprozess gab es starke Veränderungen der Anforderungen und die Notwendigkeit zur Nutzung unbekannter Technologien, die endgültigen Anforderungen unterschieden sich in einem Maße von den ersten Vermutungen des Autoren, dass die ersten Prototypen nichts mit der gewünschten Software gemein haben. Man könnte also meinen das dies etwas schlechtes ist, aus der Sicht der Evaluierung ist dies jedoch ein positiver Zug der Entwicklung, erst durch diese massiven Probleme und Herausforderungen konnten sich die Schwächen des Entwicklungsprozesses auf tun und Wege zu ihrer Lösung gefunden werden. Durch eben jene Schwierigkeiten wurden auch faktisch viele Probleme offenbar und der Autor hat Lösungsstrategien, für Probleme die öfter in Entwicklungen vorkommen können, entwickelt.

### 6.1 Erfahrungen

SoDa ist keine perfekte Methode, verfügt jedoch über viele positive Eigenschaften und ist schnell und einfach an neue Gegebenheiten anzupassen. Während der Evaluierung gab es starke Veränderung der Anforderungen und eine komplette Neustrukturierung der Software die damit verbunden war. Trotzdem war der Autor in der Lage auf diese Änderungen dynamisch zu reagieren und ein Anforderungsprofil zu erstellen, dass den Wünschen der Benutzer genügt. Zwar wurde während der Evaluierung der eigentliche Prozess von SoDa nicht verändert, um Testbedingungen zu erreichen, obwohl SoDa direkt dazu auffordert um den Prozess an die Gegebenheiten der spezifischen Entwicklungsaufgaben anzupassen. Die vielversprechendsten, der während der Evaluierung aufgetauchten, Veränderungsvorschläge sind in Kapitel [6.3](#) aufgeführt.

#### 6.1.1 Versuchsaufbau

Aufgrund von finanziellen und räumlichen Möglichkeiten war der Autor nicht in der Lage eine Pinnwand zu nutzen, stattdessen hat er, wie als Alternative vorgeschlagen, digitale Dokumente genutzt um sie zu ersetzen. Um übersicht über die Aufgaben zu behalten die an einem Tag erledigt werden mussten, wurden die täglichen Aufgaben tabellarisch ausgedruckt.



Auf diese Art und Weise diente dieser Ausdruck auch als Checkliste und konnte dazu genutzt werden bereits im Laufe eines Tages Verspätungen festzustellen. Durch dieses frühe Bemerkens von Problemen lässt sich ein frühes Gegensteuern erreichen. Wie innerhalb der Beschreibung des SoDa Prozesses vorgeschlagen wurden alle Daten durch eine Versionskontrollsoftware gesichert, wodurch jederzeit alte Versionen wiederhergestellt werden konnten und für die Entwicklung früher Prototypen wurde durch in die Entwicklungsumgebung integrierte Software an Coding Standards Vorschläge des Programmiersprachenherstellers angepasst.

### 6.1.2 Nicht striktes SoDa

Nicht striktes SoDa, also SoDa in dem halbfertige Sprintergebnisse in den nächsten Sprint übernommen werden können, hat oft den Nachteil, dass falsche Ansätze übernommen werden und daher weiter entwickelt werden, obwohl sie mehr Zeit benötigen als ein kompletter Neuanfang. Der Autor dieser Arbeit hatte aufgrund der Größe der Aufgabe, der geringen Sprintlänge und der feingranularen Aufteilung der Anforderungen selten das Problem eines falschen Ansatz, der später revidiert werden musste.

### 6.1.3 Striktes SoDa

Jedes unfertige Ergebnis eines abgeschlossenen Sprints zu verwerfen ist oft wenig befriedigend und erscheint in Fällen, in denen diese Anforderung noch nicht mehr Zeit verbraucht hat, als geschätzt und die Verspätung durch andere Anforderungen erzeugt wurde. Zwar schützt dieses Verfahren in einem gewissen Maß davor falsche Ansätze weiter zu entwickeln, kann jedoch in den erwähnten Fällen schädlich sein. Um dieses Problem zu beheben wurde eine weitere Alternative des halbstrikten SoDa erdacht. Hierzu findet sich mehr in Unterkapitel [6.3.2.3](#)

## 6.2 Einsatzfähigkeit

Die Erfahrungen, die der Autor mit dieser Methode während seiner Bachelorarbeit gemacht haben, zeigen auf, dass diese Methode es erlaubt auch auf einem unbekanntem Feld und bei vielen Veränderungen der Anforderungen Softwareentwicklung zu betreiben. Dabei bietet es durch die erstellten Dokumente die Möglichkeit die Veränderungen der Anforderungen und der bisherigen Entwicklung nachzuvollziehen und daraus Schlüsse für zukünftige Planung zu ermöglichen.

## 6.2.1 Einsatzfähigkeit für Studenten

Da der Autor zum Zeitpunkt dieser Arbeit selbst noch Student ist, und diese Arbeit das erste Projekt dieser Größenordnung ist, das er selbst planen musste, kann er aus seinen Erfahrungen viel auf die möglichen Erfahrungen anderer Studenten schließen, die sie bei der Nutzung von SoDa haben werden. Basierend auf der Idee eine agile Entwicklungsmethode für eine Bachelorarbeit zu kreieren, ist SoDa für die Arbeit an studentischen Arbeiten einsetzbar, jedoch ist es, um jegliche Form der Softwareentwicklung mit dieser Methode zuzulassen, nicht spezifisch darauf ausgerichtet. Durch diese Generalisierung entstehen einige Nachteile, die Studenten die wenig oder keine Erfahrung mit agilen Methoden haben zu einem ernstzunehmendem Problem werden könnten. Hier wäre zu nennen das keine Zeit für das eigentliche schreiben der akademischen Arbeit Zeit eingeplant wird, aber auch dass SoDa größere Freiheiten im zu gehenden Weg bereit hält als ein Wasserfallmodell. Letzteres liegt zwar inhärent in der Art der agilen Entwicklung und der Komplexität der Anforderungen eines modernen Softwaresystems, kann jedoch trotzdem einen unerfahrenen Entwickler vor unüberwindbare Hürden stellen.

### 6.2.1.1 Schreiben Akademische Arbeit

Im der bisherigen Beschreibung des SoDa Prozesses wurde das Schreiben der eigentlichen akademischen Arbeit nicht betrachtet und wurde als ein Teil der Arbeit angesehen, die während der Entwicklung anfällt. Da jedoch bereits die Schätzung der Dauer einzelner Aufgaben sehr Fehleranfällig ist, ohne zusätzliche Zeit für das schreiben der akademischen Arbeit eingeplant werden muss, wäre es zu überdenken einen unabhängigen Zeitraum einzuführen in dem die Arbeiten an der akademischen Arbeit vorgenommen werden. Die Ideen zu einem solchen Prozess sind im Kapitel [6.3.2.1](#) beschrieben.

### 6.2.1.2 Linearisierung des agilen Prozesses

Durch die Agilität des Prozesses ergeben sich viele Vorteile, deren Nutzen gerade in Bereichen in denen die Entwickler mit unbekanntem Anforderungen oder neuen Technologien konfrontiert sind, entscheidend sein können. Die Anpassbarkeit dieses Prozesses bringt jedoch nicht nur Vorteile, da durch diese auch ein Fehlen von Struktur hervorgerufen wird, das ihnen das metaphorische Genick brechen kann. Da es in SoDa keine Planung eines Endzeitpunktes gibt, dieser jedoch in einer akademischen Arbeit mit absoluter Härte festgelegt ist, ist gerade die Zeitplanung ein extremer Nachteil eines agilen Prozesses für diese Form der Arbeiten. Eine Möglichkeit zur Verbesserung dieses Problems ist in Kapitel [6.3.1.2](#) beschrieben, eine vollkommene Lösung erscheint dem Autoren jedoch unmöglich, ohne auf die Vorteile der Agilität zu verzichten.

## 6.2.2 Einsatzfähigkeit für Entwickler

Einige der für Studenten erwähnten Argumente gegen die Einsatzfähigkeit und für Veränderungen des Prozess gelten auch für Entwicklungsprozesse außerhalb von akademischen Arbeiten, jedoch ist hier das Feld der Möglichkeiten viel größer. Die Aspekte die für eine Linearisierung und für stärkere Regeln sprechen gelten vor allem für junge Entwickler und Arbeitsfelder in denen ein fest strukturierter Arbeitsablauf notwendig ist. Das Problem ein fertiges Produkt zu einem bestimmten Zeitpunkt zu liefern tritt bei SoDa durch die zyklische Entwicklung weniger auf, das Projekt des letzten Zyklus wird unter Umständen weniger Funktionalität liefern als sich der Kunde erhofft hat, jedoch Lauffähig sein und höchst priorisierte Anforderungen erfüllen. Bei herkömmlicher Entwicklung hätte ein Kunde in dem Fall einer unfertigen Software nur die Möglichkeiten die Entwicklung einzustellen und damit kein fertiges Produkt zu bekommen oder weiteres Geld in die Entwicklung zu investieren, durch die Vorteile der zyklischen Entwicklung erhält er eine dritte Alternative. Unabhängig davon sollte ein Entwickler den zeitlichen Ablauf seines Projektes bereits zu Beginn Abschätzen, dies muss schon deshalb geschehen um Abzuschätzen ob dieser Aufwand als Einzelperson zu bewältigen ist und ob sich die Entwicklung dieser Software für den Kunden lohnt. Für erfahrene Entwickler kann es auch möglich oder notwendig sein Teile des Entwicklungsprozesses zu Verändern und sie agiler oder weniger Regelbehaftet zu gestalten. Da der Autor selbst jedoch nicht zu dieser Gruppe gehört kann er nur wenig über diese Fälle aussagen, möchte aber darauf hinweisen das ein wichtiger Teil von SoDa seine Anpassbarkeit ist und jede Veränderung die Funktioniert eine gute Veränderung ist.

## 6.3 Veränderungsvorschläge

In den beiden folgenden Abschnitten werden Veränderungsvorschläge beschrieben, die während der Evaluation des Entwicklungsprozesses aufgetreten sind. Sie wurden in zwei Gruppen aufgeteilt, Veränderungen die notwendig sind um einen geregelten Ablauf der Entwicklung zu gewährleisten und diejenigen die wahrscheinlich oder unter bestimmten Umständen nutzbringend sind.

### 6.3.1 Notwendige Veränderungen

Während der Evaluierung des SoDa Entwicklungsprozess sind viele Schwachstellen hervorgetreten, die durch Veränderungen berichtigt werden müssen. Dieser Abschnitt beschreibt Veränderungen die notwendig sind um einen effektiven Ablauf zu gewährleisten.

### 6.3.1.1 Daily $CO_2$

Oft ist es nicht möglich innerhalb von Entwicklungssprints Kommunikation auf die notwendige Klärung von Fragen zu den Anforderungen des aktuellen Sprints zu beschränken. Die Gründe hierfür können Inhärent des zu lösenden Problems, in Einstellungen oder zeitlichen Beschränkungen des Kunden, oder in anderen äußeren Umständen liegen. Um dieses Problem zu lösen kann man eine tägliche Stunde in den Arbeitsablauf einfügen in der sich der Entwickler nur um die Kommunikation mit Benutzern und Kunden kümmern kann, während die restliche Zeit der ungestörten Entwicklung vorbehalten ist.

### 6.3.1.2 Vorplanung des zeitlichen Ablaufs

Um fertige Ergebnisse zu einem festgelegten Endzeitpunkt zu haben ist eine weitreichende Planung notwendig, die vom Beginn zu diesem Endzeitpunkt reicht. Ein solches Handeln widerspricht jedoch Grundsätzlich der Philosophie der agilen Entwicklung, da, wie schon mehrfach erwähnt, zu keinem Zeitpunkt die Anforderungen eines Softwareproduktes vollständig verstanden werden können. In Fällen, in denen ein definiertes Ergebnis zu einem bestimmten Zeitpunkt fertig sein muss, ist agile Entwicklung damit weniger geeignet, um die Vorteile agiler Entwicklung trotzdem Nutzen zu können ist es daher notwendig eine Anzahl Meilensteine zu erstellen, die genutzt werden kann um während der Entwicklung zu überprüfen ob man innerhalb der Entwicklung mit genügender Geschwindigkeit vorankommt. Falls es hier zu Verspätungen kommt, können so Maßnahmen, wie verlängerte Arbeitszeit oder Streichung von Anforderungen, bzw. eine Veränderung des Endzeitpunktes frühzeitig abgesprochen und vorgenommen werden.

### 6.3.1.3 Unabhängigkeit von Analyseschätzungen

Während der Entwicklung ist dem Autoren aufgefallen, dass die Qualität der Einschätzungen zwischen Analysesprints und Entwicklungssprints sehr unterschiedlich ist. Der Faktor  $\frac{gAIS}{SPIS}$  Springt zwischen Analysesprints und Entwicklungssprints sehr viel stärker als zwischen Analysesprints und Entwicklungssprints. Deshalb sollte diese Faktoren unabhängig berechnet werden.

**Schätzung für Analysesprints** Arbeitsstunden für Analysesprints werden

$$\text{Arbeitsstunden} = \frac{gAIS}{SPIS} * SP$$

Wobei gAIAS die gemessenen Arbeitsstunden des letzten Analysesprints, SPIAS die Storypoints des letzten Analysesprints und SP die Storypoints dieses Sprints sind.

**Schätzung für Entwicklungssprints** Arbeitsstunden für Analysesprints werden

$$\text{Arbeitsstunden} = \frac{gAIES}{SPIES} * SP$$

Wobei gAIES die gemessenen Arbeitsstunden des letzten Entwicklungssprints, SPIES die Storypoints des letzten Entwicklungssprints und SP die Storypoints dieses Sprints sind.

### 6.3.2 Mögliche Veränderungen

Im folgenden Abschnitt werden Veränderungen beschrieben die den Prozess weiter verbessern können, jedoch nur für eine kleine Teilgruppe der Einzelentwickler hilfreich ist, oder deren Nutzen nicht überprüft wurde und auch keine Fakten zu Verfügung stehen um diese neuen Veränderungen als zwingend Notwendig zu empfehlen.

#### 6.3.2.1 Unabhängiger Schreibprozess für akademische Arbeiten

Um bei akademischen Arbeiten die Zeit für das Schreiben der schriftlichen Ausarbeitung nicht während des Schätzprozess für die eigentlichen Entwicklungsaufgaben im "vorübergehen" abzuschätzen und um den Entwicklungsprozess ungestört von diesen Aufgaben ablaufen zu lassen, wäre es denkbar den Schreibprozess getrennt vom restlichen SoDa laufen zu lassen. Dies könnte in eigenständigen Sprints passieren oder indem die letzten, oder ersten, zwei Stunden eines Tages für das Schreiben der Ausarbeitung genutzt werden. Ob dieser Ansatz empfehlenswert ist, konnte innerhalb dieser Arbeit jedoch nicht festgestellt werden, da, wie bereits erwähnt, der Versuchsaufbau gegenüber dem initialen Vorschlag nicht geändert wurde.

#### 6.3.2.2 Planung und Diktiergeräte

Innerhalb der Planungen für Sprints und Tagen sollten Diktiergeräte, oder andere Aufzeichnungsmöglichkeiten benutzt werden. Bei der Erstellung dieser Arbeit wurden sowohl Planungen mit Diktiergerät als auch ohne ausgeführt und der Autor ist zu dem Schluss gekommen, dass es vorteilhaft ist das tägliche Vorhaben und die Probleme die dabei entstehen können laut zu sprechen und zu hören. Als beste Möglichkeit hat sich dabei herausgestellt, die im

Kapitel 4.5.3 gestellten Fragen zu beantworten und sich die Antworten nach der Beantwortung aller Fragen erneut anzuhören. Hierdurch reflektiert man besser das Gesagte und die Informationen bleiben besser im Gedächtnis.

### **6.3.2.3 Halbstrikted SoDa**

Um die Vor- und Nachteile von striktem und nicht striktem SoDa miteinander zu verbinden, wäre es denkbar, dass nach dem Ende eines Sprints alle Ergebnisse verworfen werden, die bereits länger benötigt haben als ihre Schätzung als Gesamtdauer angegeben hat. Bei anderen Anforderungen wird die Restdauer bei Beginn des nächsten Sprints geschätzt und mit der Differenz der bisherigen Arbeitszeit zur Gesamtdauer verglichen. Ist die Restschätzung kleiner oder nur geringfügig höher wird das Ergebnis in den folgenden Sprint übernommen.

# 7 Fazit

*Solo Development is agile* ermöglicht was sein Name verspricht, agil zu Entwickeln ohne dabei an Gruppen gebunden zu sein. Zwar ist diese Form in den meisten Fällen nicht der klassischen agilen Entwicklung vorzuziehen, kann aber in den Fällen in denen die Aufgaben klein genug sind oder äußere Umstände dazu führen, dass als Einzelperson entwickelt wird, ein unverzichtbares Hilfsmittel darstellen. Die meisten noch existierenden Nachteile erscheinen inhärent in den momentanen Ansätzen der agilen Entwicklung, der Einzelentwicklung oder durch die Veränderungen der Evaluierungsphase gelöst zu sein. Somit ist es für die Aufgaben für die es gedacht war Einsatzfähig. Weitere Anpassungen werden natürlich unumgänglich sein, doch dies ist die eigentliche Stärke eines agilen Prozesses, dass er an die Gegebenheiten und Erfordernisse angepasst werden kann.

## 7.1 SoDa aus Sicht des Autors

SoDa hat an der Entwicklungsmethode des Autors viel geändert. In vorherigen "Solo" Projekten hat er zwar bereits einzelne Praktiken der agilen Entwicklung benutzt, bisher waren diese Anstrengungen jedoch eher ungerichtet und es fehlte vor allem an Planung. Bisherige Projekte verliefen zumeist so, dass nachdem ein Grundverständnis des Problems vorhanden war, angefangen wurde die bekannten Anforderungen zu erfüllen, neue Anforderungen wurden erfüllt sobald sie auftauchten. Dabei wurde testgetrieben, unter Beachtung eines simplen Design und mit Coding Standards gearbeitet, jedoch fehlten vor allem ausreichende Analysen der Anforderungen und Ordnung. SoDa hat geholfen Strukturen einzubringen und die chaotischen Prozesse zu ordnen. Zwar kostete die stärker gegliederte Arbeit viel Zeit, dies ist jedoch erheblich weniger als die zusätzliche Arbeitszeit, die durch Fehler des vorherigen Systems entstanden sind. Durch die klaren Strukturen und die expliziten Phasen Anforderungserhebung fällt es so deutlich leichter Anforderungen schneller zu erkennen und besser auf sie zu reagieren. Durch die Festlegung auf die zu erfüllenden Anforderungen viel es dem Autoren während dieser Arbeit auch deutlich einfacher sich an ein simples Design zu halten und nicht "voraus" zu programmieren.

## 7.2 Ausblick in die Zukunft

### 7.2.1 Die Zukunft von Solo Development is agile

Durch die Erfahrungen, die der Autor bisher mit SoDa gemacht hat, wird er diesen Prozess auch in Zukunft verwenden und weiter verbessern. Darüber hinaus wären noch weitere akademische Arbeiten denkbar die SoDa benutzen oder in denen die Entwicklung von SoDa vorangetrieben wird und nicht vom Autor dieser Arbeit stammen. Arbeiten über die agile Entwicklung einer spezifischen Software könnten auf diesen Prozess zurückgreifen und Arbeiten über die Integration von Daily  $CO_2$  und halbstriktem SoDa, sowie eigenen Ideen des Autors dieser theoretischen Arbeit, könnten folgen. Auch die Konzeption und Konstruktion einer, auf die Strukturen und Bedürfnisse von SoDa, zugeschnittenen Managementsoftware könnten diesem Projekt folgen, um die Arbeit mit SoDa zu vereinfachen. Zusätzlich zu diesen Arbeiten könnte SoDa in mehrere Versionen, für Anfänger und Experten, gegliedert werden um die in er Evaluierung kurz angesprochenen Diskrepanzen zu verringern. Dies könnte, zum Beispiel, durch Alternativregeln für Experten erreicht werden.

### 7.2.2 Die Zukunft der Autorensoftware

Als ein Gebiet in dem der Autor persönliches Interesse hat, wird die Autorensoftware, zumindest für seinen privaten Gebrauch, weiterentwickelt. Doch auch hier wären weitere Themen für Bachelorarbeiten ansiedelbar, da die Vorurteile der Autoren gegenüber speziell für Autoren geschriebener Software von den Mängeln in der Bedienbarkeit ausgeht. Es könnten also Versuche folgen, in denen, zum Beispiel durch das nutzen von explorativen Prototypen, eine Software erstellt wird, die die Autoren durch einfache Bedienbarkeit und klare Funktionalität von dem Gegenteil ihrer Meinung überzeugt.



# Literaturverzeichnis

- [Beck 1999] BECK, Kent: *Extreme Programming explained*. Addison-Wesley, 1999. – ISBN 0-201-61641-6
- [Beck u. a. 2001] BECK, Kent ; BEEDLE, Mike ; BENNEKUM, Arie van ; COCKBURN, Alistair ; CUNNINGHAM, Ward ; FOWLER, Martin ; GRENNING, James ; HIGHSMITH, Jim ; HUNT, Andrew ; JEFFRIES, Ron ; KERN, Jon ; MARICK, Brian ; MARTIN, Robert C. ; MELLOR, Steve ; SCHWABER, Ken ; SUTHERLAND, Jeff ; THOMAS, Dave: *Manifesto for Agile Software Development*. 2001. – URL <http://agilemanifesto.org>
- [Beck und Fowler 2000] BECK, Kent ; FOWLER, Martin: *Planning Extreme Programming*. Addison-Wesley, 2000. – ISBN 0-201-71091-9
- [Bell 2007] BELL, Peter: *Solo Scrums*. 2007. – URL <http://www.pbell.com/index.cfm/2007/6/17/Solo-Scrums>
- [Celtx 2008] CELTX: *Celtx*. 2008. – URL <http://www.celtx.com/>
- [Dean] DEAN, Michael L.: *Double Choco Latte*. – URL <http://dcl.sourceforge.net/>
- [GIMP] GIMP: *GNU Image Manipulation Program*. – URL <http://www.gimp.org/>
- [Haynes 2008] HAYNES, Simon: *yWriter 4*. 2008. – URL <http://www.spacejock.com/yWriter4.html>
- [Kahlow 1997] KAHLOW, Matthias: *TeVeRo Step 2*. 1997
- [Kile] KILE: *Kile*. – URL <http://kile.sourceforge.net/>
- [Kniberg 2007] KNIBERG, Henrik: *Scrum and XP from the Trenches*. Lulu.com, 2007. – URL <http://www.infoq.com/minibooks/scrum-xp-from-the-trenches>. – ISBN 978-1-4303-2264-1
- [Ltd 2007] LTD, Anthemion S.: *Writer's Café*. 2007. – URL <http://www.writerscafe.co.uk/>
- [Niedermeier und Niedermeier 2005] NIEDERMEIER, Elke ; NIEDERMEIER, Michael: *LaTeX. Das Parxisbuch*. Franzis, 2005. – ISBN 3-7723-6434-9
- [OpenOffice.org] OPENOFFICE.ORG: *OpenOffice.org*. – URL <http://www.openoffice.org/>

- 
- [Stein 1997] STEIN, Sol: *Über das Schreiben*. Zweitausendeins, 1997. – ISBN 3-86150-226-7
- [Wikipedia 2008] WIKIPEDIA: *Wikipedia Artikel Scrum*. 2008. – URL [http://en.wikipedia.org/w/index.php?title=Scrum\\_%28development%29&oldid=196060366](http://en.wikipedia.org/w/index.php?title=Scrum_%28development%29&oldid=196060366)
- [XPlanner ] XPLANNER: *XPlanner*. – URL <http://xplanner.org/>

# A Hilfsmittel

Für die Erstellung dieser akademischen Arbeit hat der Autor  $\LaTeX$  und die KDE  $\LaTeX$ Umgebung Kile[Kile]. Alle Grafiken, mit Ausnahme von Abbildung 5.1 wurden mit dem GNU Image Manipulation Program[GIMP] erstellt. Abbildung 5.1 wurde mit der Hilfe der OpenOffice.org[OpenOffice.org] Komponente Calc erstellt. Weiterhin wurden die  $\LaTeX$ Vorlagen der Hochschule für Angewandte Wissenschaften Hamburg genutzt.

Für die Programmierung der erwähnten Prototypen wurde die Eclipse Entwicklungsumgebung, sowie das OpenOffice.org Software Development Package.

# Glossar

**Analyse Backlog** Ein Dokument, in dem alle Anforderungen die in einem spezifischen Analysesprint bearbeitet werden festgehalten werden, kann während des Sprints nicht bearbeitet werden.

**Analysesprint** Eine Sprintart, in der Anforderungen erhoben werden.

**Benutzer** Die Person oder Gruppe die das Produkt benutzen soll, sobald es fertiggestellt ist.

**Black Box Test** Testen des äußeren Verhaltens, ohne Kenntnisse über den Aufbau der Software.

**Burn Down Chart** Ein Dokument, das den Verlauf eines Sprints wiedergibt.

**CO<sub>2</sub>** Phase zwischen zwei Sprints die zur Koordinatination und Orientierung verwendet wird.

**Coding Standards** Eine Sammlung von Standards, die die Formatierung eines Quellcodes bestimmen, kann Programmiersprachen und/oder Firmenspezifisch sein.

**Entwicklungssprint** Eine Sprintart, in der Anforderungen erfüllt werden.

**Glossar** Ein Dokument in dem Worte des Sprachgebrauchs des Benutzers festgehalten werden.

**Kunde** Die Person oder Gruppe die das Produkt in Auftrag gibt, diese Gruppe kann aus anderen Personen bestehen als die eigentlichen Benutzer des Systems

**Product Backlog** Ein Dokument, in dem alle Anforderungen festgehalten werden.

**Refactoring** Bezeichnet die Verbesserung der Struktur des Quellcodes ohne seine Funktion zu verändern.

**Sprint** Ein einwöchiger Zeitraum, in dem an einem spezifischen Ziel gearbeitet wird, währenddessen sich die Anforderungen, die in dieser Zeit bearbeitet werden, nicht ändern können.

**Sprint Backlog** Ein Dokument, in dem alle Anforderungen die in einem spezifischen Sprint bearbeitet werden festgehalten werden, kann während des Sprints nicht bearbeitet werden.

**Sprintendeplanung** Planung zum Ende eines Sprints, Rückblickend werden die Stolpersteine betrachtet und es wird versucht Lösungen zu finden. Gute und schlechte Ereignisse des letzten Sprints werden betrachtet.

**Sprintplanung** Planung eines Wöchentlichen Sprints, in der festgelegt wird, welche Ziele hier zu erfüllen sind.

**Sprinttyp** SoDa unterscheidet zwischen zwei Arten von Sprints, Analysesprints und Entwicklungssprints

**Tagesplanung** Planung der Aktivitäten des folgenden Tages, und Retrospektive des Vortages.

**Testgetriebene Entwicklung** Eine bestimmte Form der Entwicklung, in der zuerst Tests geschrieben werden, die dann von der Software erfüllt werden müssen.

**Use Case** Eine Beschreibungsmöglichkeit von funktionalen Anforderungen.

# Index

*CO*<sub>2</sub>, [51](#)

Anwendungsfallbasiertes Testen, [24](#)

Daily *CO*<sub>2</sub>, [92](#)

Halbstriktes SoDa, [94](#)

Nicht striktes SoDa, [89](#)

Product Backlog, [38](#)

Product Owner, [21](#)

ScrumMaster, [21](#)

Scrumtreffen, [20](#)

Sprint Backlog, [42](#)

Sprintendeplanung, [50](#)

Sprintplanung, [46](#)

Sprinttypen, [44](#)

Striktes SoDa, [89](#)

Tagesplanung, [49](#)

Testgetriebene Entwicklung, [24](#), [36](#)

Use Case, [22](#)

# Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 21. Juli 2008

Ort, Datum

Unterschrift