FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Planung und Implementierung von angemessenen agilen Testmethoden und Testautomatisierungsstrategien für ein teilprojektreiches Technologieintegrationsprojekt

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Boris Wrubel, BSc
Matrikelnummer 9947268

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Roland Breiteneder

Wien, 3. Mai 2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Planung und Implementierung von angemessenen agilen Testmethoden und Testautomatisierungsstrategien für ein teilprojektreiches Technologieintegrationsprojekt

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Boris Wrubel, BSc
Matrikelnummer 9947268

ausgeführt am
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Roland Breiteneder

Wien, 3. Mai 2013

Eidesstattliche Erklärung

Boris Wrubel, BSc
Kuefsteingasse 33, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Dank gilt Thomas Grechenig für die Anregung der Arbeit und Bereitstellung des Themas.

Roland Breiteneder danke ich für die Unterstützung bei der Themeneingrenzung und die Möglichkeit, die Erkenntnisse aus den Prozessverbesserungen in einem industriellen Projekt in dieser Arbeit verwenden zu dürfen.

Ich bedanke mich bei allen, die mich während meines Studiums unterstützt haben, allen voran bei meiner Familie: Sie hat mich stets moralisch und auch tatkräftig unterstützt. Ohne sie hätte ich dieses Studium nicht abschließen können.

Besonders möchte ich mich bei meiner Frau, Agnes, bedanken, die mit ihrem unermüdlichen Einsatz die Arbeit Korrektur gelesen und durch ihr unablässig forschendes Hinterfragen diese positiv beeinflusst hat.

Dank auch an alle Studienkollegen, die mich begleitet haben und mit mir nicht nur für die zahlreichen Lehrveranstaltungen gelernt, sondern diese auch mit mir erfolgreich abgeschlossen haben.

Außerdem bedanke ich mich bei Martina Malzer für die verlässliche und geduldige Umsetzung sämtlicher Grafiken und Stefan Taber, der all meine Latex-Probleme mit Leichtigkeit gelöst hat.

Abschließend gilt mein Dank auch all jenen Freunden, die niemals aufhören, an mich zu glauben.

„Jetzt sind die Tester dran“

Harry Sneed
in seiner Keynote zur iqnite Deutschland 2012

Kurzfassung

Agiles Testen ist ein in der derzeitigen Theorie und Praxis des Softwaretestens wenig behandeltes Gebiet, obwohl die agile Softwareentwicklung eine Realität geworden ist, die nach dazu passenden Testansätzen verlangt. Damit verändern sich nicht nur die Aufgabengebiete im Test, es kommen auch ganz neue Schwerpunkte und Verantwortungen zum Tragen.

Was dieser Wandel für den agilen Test bedeutet, wird in der vorliegenden Arbeit anhand eines konkreten Fallbeispiels erläutert. Die Analyse zeigt diverse Probleme, die entstehen können, wenn klassisches Testvorgehen auf agile Softwareentwicklung trifft und sie macht umso mehr deutlich, dass im Softwaretest ein Paradigmenwechsel angesagt ist. Gleichzeitig werden Lösungsansätze herausgearbeitet und in Folge ein beispielhafter agiler Testansatz vorgestellt.

Der hier vorgestellte agile Testansatz beinhaltet einen Testprozess für agile Vorgehensmodelle und zeigt darüber hinaus auf, wie sich Testautomatisierung effizient in diesen integrieren lässt. Im Zusammenhang damit wird dargestellt, wie sich der Softwaretest die in der iterativen Entwicklung bereits etablierte Methode der Continuous Integration zunutze machen kann. Des Weiteren wird gezeigt, welche Wichtigkeit der Einsatz von passenden Werkzeugen im agilen Test hat und welche Bedeutung daher der Evaluierung dieser Tools zukommt. Abschließend wird der Wandel des Rollenbildes des Testers im agilen Umfeld behandelt, denn agile Testmodelle verlangen auch nach einem neuen Jobprofil des Testers.

Schlüsselwörter

Agiler Test, Agiler Testprozess, Testautomatisierung, Behaviour Driven Development

Abstract

Agile testing is a topic that is rarely dealt with in the current theory and practice of software testing, although agile software development has become a reality which also demands appropriate approaches to testing. This not only changes the task areas in the test, but also brings entirely new focal points and responsibilities to bear.

What this change means for the agile test will be explained in this paper using a practical project by way of example. The analysis reveals various problems that can arise when classic test procedure meets agile software development, and makes it even clearer that it is time for a paradigm change in software testing. At the same time, solution approaches are formulated and a exemplary agile test approach presented.

The here introduced agile test approach includes a test process for agile models and also reveals how test automation can be efficiently integrated into this. In this context, it will be shown how software test can make use of the continuous integration method which is already established in iterative software development. Furthermore, the importance of using appropriate tools in the agile test will be shown and therefore the significance of evaluating these tools. Finally, the change of the tester's role in the agile environment will be dealt with because agile test models also demand a new job profile for the tester.

Keywords

agile test, agile test process, test automation, Behaviour Driven Development

Inhaltsverzeichnis

Aus Gründen der besseren Lesbarkeit wird in der vorliegenden Arbeit nur eine geschlechtsspezifische Form verwendet.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

API Application Programming Interface (deutsch: Programmierschnittstelle)

BDD Behaviour Driven Development (deutsch: Verhaltensgetriebene Softwareentwicklung)

CAT Certified Agile Tester

CTFL Certified Tester Foundation Level

DAkkS Deutsche Akkreditierungsstelle

EEPROM Electrically Erasable Programmable Read-Only Memory (deutsch: elektrisch löschbarer programmierbarer Nur-Lese-Speicher)

FDD Feature Driven Development (deutsch: Funktionsgetriebene Entwicklung)

iSQI International Software Quality Institute GmbH

ISTQB International Software Testing Qualifications Board

LCSAJ Linear Code Sequence and Jump

NFC Near Field Communication (deutsch: Nahfeldkommunikation)

RFID Radio Frequency Identification (deutsch: Identifizierung mittels elektromagnetischer Wellen)

SAM Secure Access Module

SLA Service Level Agreement (deutsch: Dienstleistungsvereinbarung oder Dienstgütevereinbarung)

SUT System Under Test (deutsch: Das zu testende Softwaresystem)

TDD Test Driven Development (deutsch: Testgetriebene Entwicklung)

XP eXtreme Programming

1 Einleitung

Die agile Softwareentwicklung hat längst breite Akzeptanz gefunden und gilt als jenes Entwicklungsprinzip, das am schnellsten auf Änderungen in den Anforderungen moderner Softwareentwicklung reagieren kann. Zahlreiche agile Vorgehensmodelle wurden bereits entwickelt und über die Jahre erprobt. Sowohl die theoretische Literatur als auch die praktischen Erfahrungsberichte sind umfassend und bilden die gesamte Palette an Projektgrößen, -umfeldern, -komplexitäten und -anforderungen ab.

Gleichzeitig wurde dem Softwaretest, der in klassischen Vorgehensmodellen als Teildisziplin bereits verankert ist, im agilen Umfeld bisher eher wenig Beachtung geschenkt. Historische Parallelen sind dabei nicht von der Hand zu weisen: Auch bei klassischen Vorgehensmodellen hat es jahrzehntelang gedauert, bis der Softwaretest seinen Stellenwert etablieren und sich gegenüber der Softwareentwicklung behaupten konnte. In der agilen Gegenwart ist der Softwaretest in einer ganz ähnlichen Situation, denn während sich die agile Softwareentwicklung stetig neu erfindet und weiterentwickelt, befindet sich der agile Test noch in einer initialen Selbstfindungsphase.

In der Literatur zu agilen Vorgehensmodellen wird der Softwaretest zwar immer erwähnt, jedoch wird meist der Test nicht als eigene Disziplin dargestellt, sondern als Teil des Entwicklungsprozesses gesehen. Über den Einsatz von dedizierten Testern, also Projektmitarbeitern, die sich ausschließlich um die Qualität kümmern, ist sich die bisherige Literatur nicht wirklich einig. In der neueren Literatur werden aber bereits eigene Tester für das Projektteam gefordert ([82], [19], [51]). Trotz allem, hat die breite Diskussion und Anwendung von agilen Teststrategien bisher gesamthaft gesehen eine Nebenrolle gespielt. In dieser Arbeit gilt jedoch - frei nach Harry Sneed: „Jetzt sind die Tester dran“ [74]. Denn einen funktionierenden agilen Testansatz zu etablieren ist von großer Wichtigkeit für den Gesamterfolg von industriellen Projekten.

Auch das International Software Quality Institute GmbH (iSQI) hat die Veränderung erkannt und einen Lehrplan mit Zertifizierungsmöglichkeit für agile Tester erstellt, den sogenannten Certified Agile Tester (CAT). Dieser behandelt Testmethoden und Aspekte des agilen Tests, jedoch fehlt in diesem Lehrplan ein agiler Testprozess. In der vorliegenden Arbeit wird jedoch mit einem beispielhaften agilen Testprozess ebendiese Lücke geschlossen.

Diese Arbeit widmet sich anhand eines industriellen Entwicklungsprojektes der praktischen Anwendung und den Vorgehensweisen des agilen Tests. Dabei werden im Fallbeispiel nicht nur praktische Ansätze diskutiert, sondern gleichzeitig Aspekte aufgezeigt, die in einem agilen Testprozess von Bedeutung sind.

Dabei zeigt sich, dass der Tester im agilen Umfeld neue Aufgaben wahrnimmt, koordinierend zwischen Entwicklern und Product Owner agiert und deutlich mehr Verantwortung übernimmt, als bisher in klassischen Vorgehensmodellen. Somit wandelt sich nicht nur die Testtätigkeit sondern das komplette Rollenbild des Testers. Wie sehr sich der Aufgabenbe-

reich des Testers in agilen Projekten im Vergleich zu den klassischen, meist sequentiellen Modellen, wandelt, wird ebenfalls erklärt.

Ziel dieser Arbeit ist es, die relevanten Themenbereiche für den Softwaretester herauszuarbeiten und einen Leitfaden zu liefern, wie diese in einem Projekt umgesetzt werden können.

Die relevanten Themen der Arbeit beziehen sich auf folgende Teilbereiche des Softwaretests:

- Testprozess im agilen Umfeld
- Agile Testautomatisierung
- Continuous Integration für den agilen Test nutzen
- Auswahl der Werkzeuge im agilen Test
- Rollenbild des agilen Testers

Jeder der Teilbereiche geht von einer Aufarbeitung und Analyse der Probleme an der Projektsituation aus und bietet ganz konkrete Vorschläge, wie Lösungen implementiert werden können. Daraus ergibt sich jeweils ein Teil des theoretischen Grundgerüstes des agilen Testvorgehens, welches hier als Leitfaden herausgearbeitet wird.

Die Arbeit ist in folgende Bereiche untergliedert: Einführend werden in Kapitel 2 die Grundlagen des Softwaretests sowie der agilen Softwareentwicklung erläutert. Da für den Praxisteil relevant, wird ebenso Grundwissen zur Near Field Communication vermittelt.

Es folgt eine Projektbeschreibung in Kapitel 3, um einen Überblick über das Projekt und die praktischen Herausforderungen zu bekommen. In diesem Kapitel zeigt sich, welche Probleme ein fehlender agiler Testprozess in einem agilen Projekt erzeugen kann.

Kapitel 4 geht dann im Detail auf die einzelnen Probleme ein und liefert Lösungsansätze, die unter Umständen auch in anderen agilen Projekten angewendet werden können. Der daraus hervorgehende beispielhafte agile Testprozess dient nicht nur als Leitfaden, sondern soll auch dazu anregen, sich mit ebensolcher Intensität mit dem agilen Testen auseinanderzusetzen, wie es in der agilen Softwareentwicklung seit Jahren selbstverständlich ist.

2 Grundlagen

Dieses Kapitel behandelt die Grundlagen des Softwaretests inklusive des klassischen fundamentalen Testprozesses. Es werden zusätzlich die Teststufen, Testarten und Testmethoden erläutert. Die Grundlagen des Softwaretests sind in diesem Kapitel sehr umfassend ausgearbeitet. Einerseits wird auf viele der Bereiche auch im praktischen teil eingegangen, andererseits sollen diese Grundlagen einen annähernd vollständigen Leitfaden des Softwaretests bilden, welcher auch in anderen agilen Projekten eine Hilfestellung darstellt. Das Kapitel gibt außerdem einen Einblick in die Testautomatisierung. Weiters wird ein Überblick über die agile Softwareentwicklung gegeben und die technischen Grundlagen der Near Field Communication erörtert. Dieses theoretische Grundlagenkapitel bildet die Basis für das in Folge betrachtete Praxisbeispiel und ist der Ausgangspunkt für die Vorstellung eines beispielhaften agilen Testprozesses.

2.1 Grundlagen des Softwaretests

Softwaretest ist eine Teildisziplin der Softwarequalitätssicherung. Abbildung 2.1 zeigt, wie sich Softwaretest in die Softwarequalitätssicherung eingliedert. Dabei wird klar, dass gute Softwarequalität nur erreicht wird, wenn alle Maßnahmen der Qualitätssicherung gut harmonieren. Die drei Bereiche der Qualitätssicherung (siehe dazu auch Abbildung 2.1) sind dabei eng miteinander verbunden und auch der Softwaretest ist von den anderen Bereichen beeinflusst oder hat Auswirkungen auf diese.

Statische Maßnahmen
: Als statische Maßnahmen werden jene bezeichnet, die einen Teil des Systems prüfen, ohne dabei diesen Teil zur Ausführung zu bringen. Unter diesen Punkt fallen sämtliche Reviewarten wie Walkthrough [5], Inspektion [22], [31], technisches Review [77] und informelles Review [77]. Ebenso kommen hier verschiedene Code-Metriken zum Einsatz, wie zum Beispiel die zyklomatische Zahl [53], um die Komplexität eines Softwaremoduls zu bestimmen.

Dynamische Maßnahmen
: Unter dynamischen Maßnahmen werden in der Softwarequalitätssicherung jene Methoden verstanden, bei denen das System Under Test (deutsch: Das zu testende Softwaresystem) (SUT) zur Ausführung kommt. Daher werden sowohl die Analyse als auch der Softwaretest während der Ausführung vorgenommen.

Organisatorische Maßnahmen
: Die organisatorischen Maßnahmen werden von Grechenig u. a. in [34] als Maßnahmen beschrieben, die notwendige Rahmenbedingungen schaffen, um analytische Maßnahmen überhaupt zu ermöglichen. Diese wirken qualitätslenkend, indem sie geeignete Methoden und Werkzeuge vorgeben, damit das Entstehen von Fehlern bereits im Voraus verhindert wird.

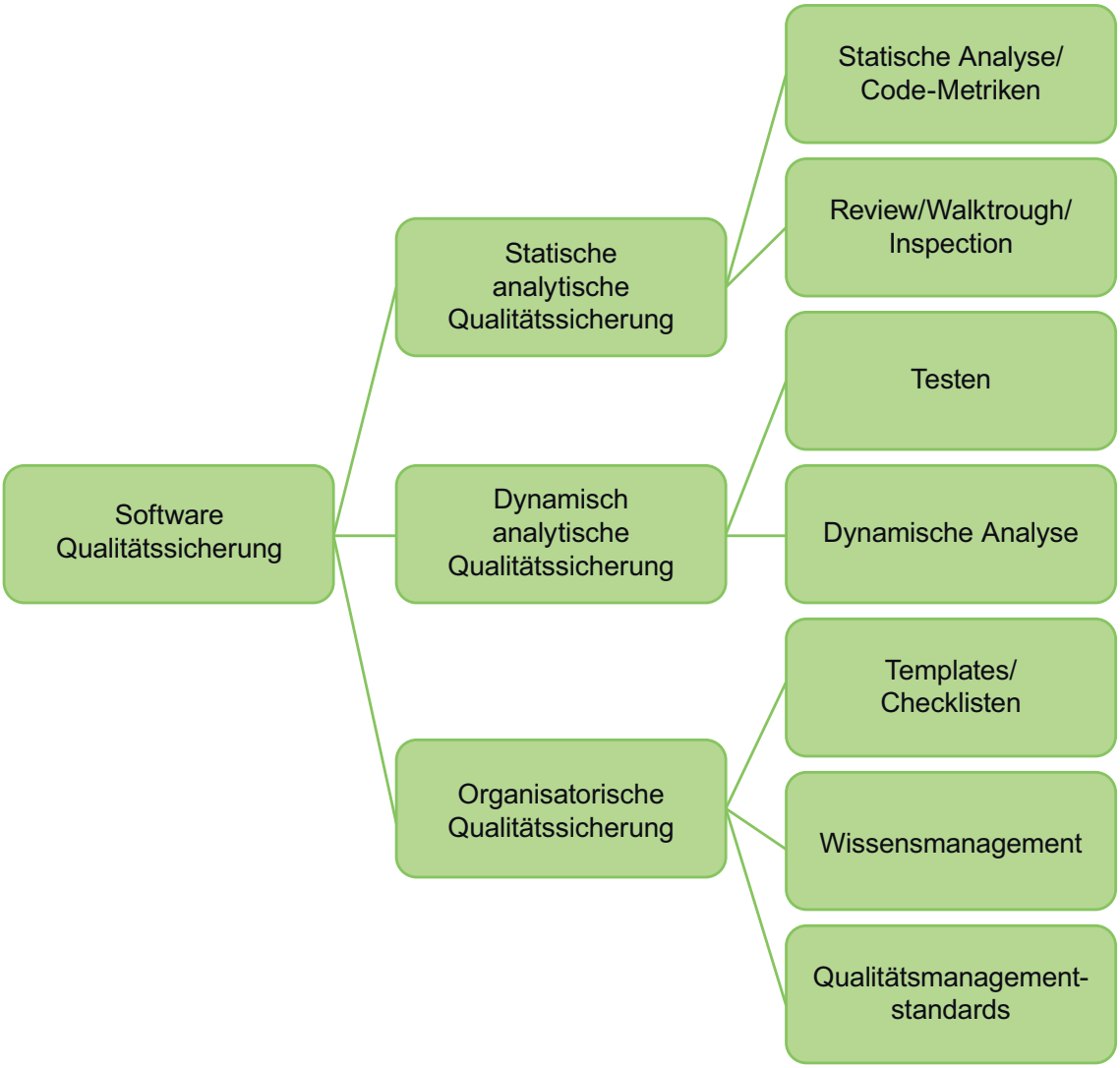

Abbildung 2.1: Software Qualitätssicherungsmethoden nach Frühauf, Ludewig und Sandmayr

Grechenig u. a. erläutern in [34], dass der Softwaretest schon längst eine Expertendisziplin darstellt. Auch Sneed, Baumgartner und Seidl in [75] bestätigen, dass die Notwendigkeit des Softwaretests unumstritten ist. Die Aufgabe des Softwaretests ist neben der Aufdeckung von Fehlern, Vertrauen in die Software, die getestet wird, zu schaffen. Denn eine vollständige Fehlerfreiheit von nicht trivialer Software kann auch durch Testen nicht sichergestellt werden, siehe dazu [34] und [65]. Da die SUTs eine hohe Komplexität aufweisen, ist eine möglichst umfassende Aufdeckung von Fehlern nur durch systematisches Testen möglich, indem Abweichungen von der Spezifikation aufgezeigt werden. Diese Testaktivitäten müssen stets sorgfältig geplant werden, da sich Testaktivitäten mitunter genau so aufwändig gestalten, wie der Entwicklungsprozess. Crispin und Gregory beschreiben in diesem Zusammenhang in [19], wie wichtig die Kommunikationskultur zwischen Testern und Entwicklern ist, da die Behebung der Fehlerzustände keineswegs die Aufgabe des Testers ist, sondern in der Verantwortung des Entwicklers liegt.

Im Jahre 2002 wurde das International Software Testing Qualifications Board (ISTQB) gegründet, um erstmals einen standardisierten Lehrplan für Softwaretester zu erstellen. Heute haben diese Inhalte weltweit Bedeutung und werden stetig überarbeitet und erweitert, um den zahlreichen Weiterentwicklungen der letzten zehn Jahre in der Softwareentwicklung gerecht zu werden, welche klarer Weise auch Auswirkung auf den Softwaretest haben.

Bei der Ausarbeitung des Lehrplans für die Zertifizierung zum Certified Tester Foundation Level (CTFL) haben Spillner und Linz in [77] folgende sieben Grundsätze zum Softwaretest postuliert:

1. Testen zeigt die Anwesenheit von Fehlern
Testen liefert den Beweis, dass Fehlerwirkungen in der zu testenden Software, dem Testobjekt, vorhanden sind. Gleichzeitig wird die Wahrscheinlichkeit verringert, dass weitere unentdeckte Fehlerzustände in diesem Testobjekt vorhanden sind. Jedoch kann keine Fehlerfreiheit durch Testen bewiesen werden.

2. Vollständiges Testen ist nicht möglich
Ausgenommen bei trivialen Testobjekten ist ein vollständiges Testen mit allen möglichen Eingabewerten und unter Betrachtung von allen möglichen Vorbedingungen nicht möglich. Denn durch die Kombination von mehreren Eingabefeldern wächst die mögliche Anzahl an Testfällen ins Unendliche. Beim Testen wählt man somit ein Stichprobe nach Priorität und Risiko, um die Aufwände in einem sinnvollen Rahmen zu halten.

3. Mit dem Testen frühzeitig beginnen
Eine immer wieder durch die Praxis bestätigte Faustregel der Softwareentwicklung besagt: Je früher ein Fehler gefunden wird, desto weniger Kosten werden durch seine Beseitigung verursacht. Mit dem Testen sollte daher bereits vor dem ersten lauffähigen Release begonnen werden.

4. Häufung von Fehlern
In Teilen des Testobjektes, in denen vermehrt Fehlerwirkungen auftreten, ist die Wahrscheinlichkeit, dass sich dort weitere Fehlerzustände befinden, deutlich höher, als in anderen Teilen.

5. Wiederholungen haben keine Wirksamkeit / Zunehmende Testresistenz
Tests lassen in ihrer Wirksamkeit nach, wenn sie immer nur wiederholt werden, denn sie berücksichtigen nicht neue Entwicklungen am System. Um die Effektivität der Tests nicht absinken zu lassen, müssen Tests mit neuen kombiniert oder vorhandene Tests ergänzt und erweitert werden. Vorhandene Tests sollen daher regelmäßig überprüft werden, um eine hohe Testabdeckung zu gewährleisten.

6. Testen ist abhängig vom Umfeld
Jedes System hat spezifische Eigenschaften und einen originären Kontext. Aufgrund dieser Tatsache lassen sich zwei unterschiedliche Systeme nicht exakt gleich testen. Die Tests müssen immer auf das zu prüfende System angepasst werden.

7. Trugschluss: Keine Fehler bedeutet ein brauchbares System
Ein (scheinbar) fehlerfreies System bedeutet nicht, dass das System auch den An-

forderungen des Nutzers entspricht. Um effizient und effektiv benutzbare Systeme zu bauen, ist es notwendig, den Nutzer in den Entwicklungsprozess einzubinden.

2.1.1 Fundamentaler Testprozess

Spillner u. a. haben in [78] den fundamentalen Testprozess wie in Abbildung 2.2 definiert. Dieser ist auch in allen ISTQB Lehrplänen in dieser Form vertreten und gilt als der Standard für die Implementierung eines Testprozesses. Im Folgenden sind die einzelnen Prozessphasen nach Spillner u. a. erläutert. Der fundamentale Testprozess wird an dieser Stelle nur grob umrissen, da in Folge im agilen Umfeld Abwandlungen und Varianten davon angewandt und auch in größerer Detailliertheit beschrieben werden.

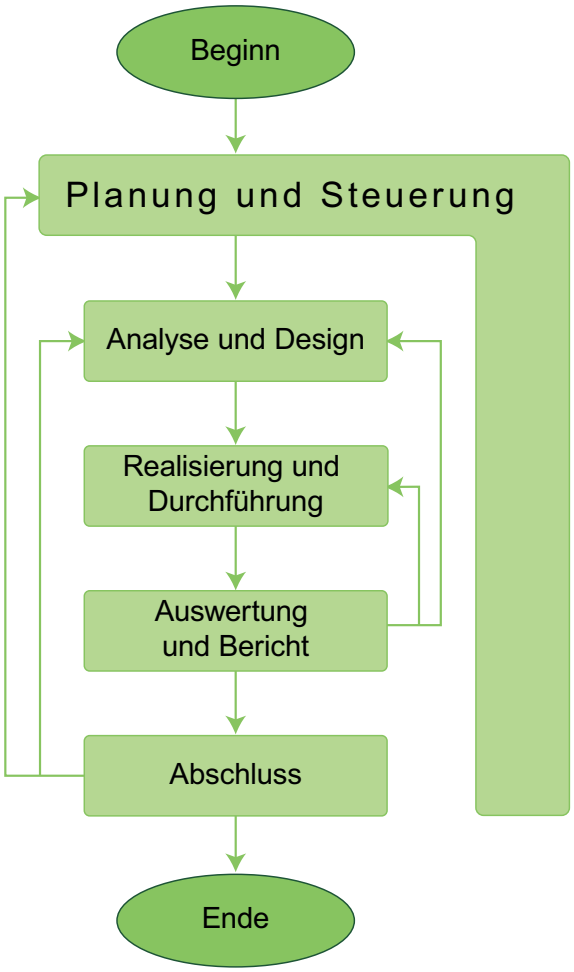

Abbildung 2.2: Fundamentaler Testprozess nach Spillner u. a.

Testplanung und Steuerung

Neben der Softwareentwicklung selbst ist das Testen die umfangreichste Aufgabe in einem Softwareprojekt. Daher soll so früh wie möglich mit der Testplanung und dem Test begonnen werden - idealer Weise gleich zu Beginn des Projektes. Denn es müssen Aufgaben, Zielsetzung und Ressourcen geplant und im Testkonzept festgehalten werden. Der Testprozess muss laufend überwacht und gegebenenfalls an Ereignisse und Probleme angepasst werden.

Analyse und Design
In dieser Phase wird analysiert, was und wie intensiv getestet wird. Darauf aufbauend werden Testfälle erstellt. Testfälle sollen immer einen Verweis auf die dazugehörigen Anforderungen haben, damit Änderungen in diesen auch in den entsprechenden Testfällen nachgezogen werden können.

Realisierung und Durchführung
In diesem Schritt werden die Testfälle konkretisiert (also mit konkreten Eingabewerten belegt) und zu Testsequenzen gruppiert. Wichtig in der Durchführung ist eine ordnungsgemäße Protokollierung, um Fehlerbehebungen schnell und effizient durchführen zu können.

Auswertung und Bericht
In Testauswertungen und -berichte fließen eine Reihe von Metriken ein, die bei der Durchführung protokolliert werden. Hieraus muss sich beispielsweise einfach ablesen lassen, ob ein zuvor definiertes Testziel erreicht wurde oder nicht.

Abschluss
Im Anschluss an die Testdurchführung und Berichterstattung sind die Ergebnisse festzuhalten, zu archivieren und an die zuständigen Personen weiterzuleiten, damit die weitere Planung darauf angepasst werden kann.

2.1.2 Teststufen

Boehm stellt in [11] anhand des V-Modells Testaktivitäten als gleichwertige Tätigkeit zur Entwicklung dar. Auch wenn nach einem anderen Vorgehensmodell entwickelt wird, die aufgezeigten Teststufen haben in jedem Entwicklungsprojekt ihre Gültigkeit. Spillner und Linz beschreiben die Teststufen in [77] wie folgt:

Komponententest
In der ersten Teststufe, dem Komponententest, werden die programmierten Softwarebausteine einem umfassenden Test unterzogen. Diese Komponenten werden isoliert getestet, um mögliche Fehlerwirkungen konkret dieser Komponente zuordnen zu können.

Integrationstest
Für die zweite Teststufe ist es notwendig, dass Komponententest durchgeführt und gefundene Fehler bereits eliminiert wurden. In der Entwicklungsphase der Integration werden verschiedene Komponenten zu größeren Modulen zusammengefügt, welche miteinander interagieren. Ob diese Interaktion korrekt funktioniert, ist mit Hilfe der Integrationstests sicherzustellen.

Systemtest
Der Systemtest soll sicherstellen, dass das entwickelte Produkt den spezifizierten Anforderungen entspricht, sämtliche Komponenten miteinander harmonieren und auch externe Schnittstellen und Systeme ordnungsgemäß mit dem entwickelten System einwandfrei funktionieren.

Abnahmetest
Die letzte Teststufe, der Abnahmetest, soll Vertrauen in die Software erzeugen und ist die Möglichkeit des Kunden, Fehlerwirkungen aufzuzeigen bevor die Verantwortung in seine Hände fällt. Oft ist der Abnahmetest jener Test, der die Erfüllung der vertraglich festgehaltenen Funktionalität testet.

2.1.3 Die vier Testarten nach ISTQB

Nach Spillner und Linz [77] lässt sich Softwaretest in vier Testarten unterteilen. Diese vier Testarten wurden ebenfalls in den ISTQB-Lehrplan [26] übernommen.

2.1.3.1 Funktionaler Test

Unter funktionalem Test sind alle Prüfmethoden einzuordnen, bei denen das Testobjekt mittels Ein- und Ausgabeverhalten getestet wird. Das gewünschte Verhalten (Sollwert, Sollverhalten) ist zuvor in den Anforderungen festgehalten. Nach ISO 9126 [48] sind folgende Merkmale als funktional einzuordnen (Erläuterung nach [6]):

Angemessenheit
Bei Angemessenheitstests wird überprüft, ob Funktionen für die spezifizierten Aufgaben geeignet sind oder nicht. Dies wird auf Basis von Anwendungsszenarien durchgeführt. Es ist wichtig, dass diese gewählten Anwendungsszenarien der tatsächlichen Arbeitsweise der zukünftigen Benutzer des Systems entsprechen, sonst kann die Angemessenheit nicht effektiv getestet werden. Unter anderem wird geprüft, ob die Software im Gebrauch nicht zu umständlich ist, wie sie mit bereits installierter Software zusammenarbeitet, ob der Speicherverbrauch angemessen ist und Ähnliches. Es ist die Aufgabe des Testers, sich in die Anwendungsszenarien hineinzuversetzen, um die Situation, Umgebung und Fähigkeiten der zukünftigen Anwender zu verstehen.

Richtigkeit
Das korrekte Verhalten einer Software wird entweder aus den Spezifikationen abgeleitet oder aus dem Fachwissen des Testers. Je nach Testfall kann die Beschreibung eines solchen Verhaltens sehr detailliert und ausführlich ausfallen, wie beispielsweise bei der Berechnung einer Versicherungsprämie, oder nur sehr allgemein sein, etwa wenn es zu überprüfen gilt, ob eine adäquate Fehlermeldung angezeigt wird.

Interoperabilität
Die zu testende Software muss auf allen vorgesehenen Umgebungen fehlerfrei funktionieren. Zu diesen Umgebungen zählen Hardware, Betriebssystem, Netzwerkkonfiguration sowie jede andere installierte Software oder auch andere Hardware, die einen Einfluss auf den Betrieb der zu testenden Software haben kann. Je höher die Interoperabilität ist, desto leichter ist der Test in einer anderen Umgebung. Je geringer sie jedoch ist, desto mehr (manuelle) Konfigurationen müssen vorgenommen werden.

Ordnungsmäßigkeit
Oftmals muss die Software gesetzlichen Bestimmungen oder anwendungsspezifischen Normen folgen.

Sicherheit
Sicherheitstests werden unterteilt in funktionale und technische Sicherheitstests. Die funktionalen Sicherheitstests sind für gewöhnlich durch die Spezifikationsanforderungen abgedeckt und betrachten Szenarien, in denen der Benutzer über die Oberfläche Zugang zu Daten oder Funktionen erhält, zu denen er nicht berechtigt ist. Technische Sicherheitstest beschäftigen sich damit, wie das System attackiert werden kann. Die Auswirkungen dieser technischen Sicherheitslücken sind breit gefächert und reichen von einem Ausfall des Systems, bis hin zur Extraktion von Daten aus dem System. Für ausführliche Informationen zu Sicherheitstests empfiehlt der Autor [83], [44] und [41].

2.1.3.2 Nicht funktionaler Test

Die nicht funktionalen Anforderungen an ein System beschreiben im Grunde wie funktionale Anforderungen umgesetzt werden sollen. In der ISO 9126 [48] sind folgende Merkmale als nicht funktional aufgelistet. Spillner und Linz beschreiben nach [77] die nicht funktionalen Anforderungen wie folgt:

Zuverlässigkeit
Die Fähigkeit eines Systems, die Funktionalität über einen Zeitraum in ein festgelegten Leistungsniveau zu bewahren.

Benutzbarkeit
Wie hoch ist der Aufwand für unterschiedliche Nutzergruppen, das System korrekt zu bedienen? Dazu gehört die Erlernbarkeit, Bedienbarkeit und Verständlichkeit. Damit Anwender effektiv mit einem System arbeiten können, muss sich dieses so verhalten, wie sie es erwarten. Dieses erwartete Verhalten ist in Konventionen und Standards festgehalten. Der wichtigste Standard für Benutzbarkeit ist die ISO 9241 [47]. Die Deutsche Akkreditierungsstelle (DAkkS) hat einen Leitfaden für Benutzbarkeit (engl. Usability) publiziert, der den Gestaltungsrahmen für den Usability-Engineering-Prozess beschreibt und Methoden entwickelt, um die Wirksamkeit dieses Prozesses zu prüfen.

Effizienz
Effizienztests erheben, mit welchen Ressourcen und in welcher Zeitspanne das System die vorgegebenen Ergebnisse erzielen kann. Effizienztest ist der Überbegriff für alle nicht funktionalen Tests, wie beispielsweise Last-, Performance-, Stresstests.

Diese nicht funktionalen Anforderungen werden meist in sogenannten Service Level Agreement (deutsch: Dienstleistungsvereinbarung oder Dienstgütevereinbarung)s (SLAs) festgehalten. Diese spezifizieren die Anforderungen und legen auch die Grenzwerte fest, bis zu denen die SLAs erfüllt werden.

Einige Beispiele für nicht funktionale Tests sind:

Lasttest Messung des Systemverhaltens in Abhängigkeit steigender Systemlast

Performanztest Messung der Verarbeitungsgeschwindigkeit oder Antwortzeit für bestimmte Anwendungsfälle

Stresstest Was passiert bei Überlastung des Systems?

Sicherheitstest[1] Test gegen unberechtigten Zugriff

Robustheit Gegenüber Fehlbedienung, ungültige Eingaben, Fehlerbehandlung aber auch Hardwareausfall

Stabilitätstests Wie verhält sich das System im Dauerbetrieb?

Benutzbarkeitstests Lässt sich die Benutzeroberfläche intuitiv und leicht bedienen?

Sehr ausführlich geht Balzert in [4] auf nicht funktionale Anforderungen ein und erklärt, wie diese sich gegenseitig beeinflussen.

2.1.3.3 Strukturbezogener Test

Beim strukturbezogenen Test wird die interne Struktur (Codezeilen, Datenaufbau, ...) analysiert und für die Testaktivitäten herangezogen. Ein Fehler kann in dynamischen Testverfahren nur aufgedeckt werden, wenn die fehlerhafte Stelle im Programmcode auch tatsächlich ausgeführt wird. Vorrangiges Ziel beim strukturellen Test ist es, dass der gesamte – oder annähernd gesamte – Code durch Testfälle abgedeckt ist. Die Code Überdeckung (engl. Code Coverage) ist im IEEE Standard für Software Unit Testing [45] festgehalten.

Pezze und Young halten in ihrem Werk [62] folgende vier Überdeckungsarten innerhalb einer Prozedur fest.

Anweisungsüberdeckung

Die am einfachsten umzusetzende Art der Überdeckung ist die Anweisungsüberdeckung. Für eine vollständige Abdeckung ist es notwendig, dass jede Anweisung mindestens einmal ausgeführt wird. Siehe dazu auch [89].

$$\text{Anweisungsüberdeckung} = \frac{\text{Anzahl ausgeführter Anweisungen}}{\text{Gesamtanzahl der Anweisungen}}$$

Jede Anweisung wird auch bei öfterer Durchführung nur einmal gezählt.

Zweigüberdeckungstest

Visualisiert man Programmcode mit Hilfe von Kontrollflussgraphen, so werden die Zweige sichtbar. Ziel der Zweigüberdeckung ist es, dass alle Zweige mindestens einmal durchlaufen werden. Da Zweige oft ohne Anweisung im Programmcode vorkommen, subsumiert eine vollständige Anweisungsüberdeckung nicht immer auch die Zweigüberdeckung, umgekehrt jedoch schon.

Die Zweigüberdeckung wird analog zur Anweisungsüberdeckung berechnet:

$$\text{Zweigüberdeckung} = \frac{\text{Anzahl ausgeführter Zweige}}{\text{Gesamtanzahl der Zweige}}$$

Bedingungsüberdeckung

Während bei der Zweigüberdeckung immer nur die ausgeführten Zweige betrachtet werden, egal wie die Entscheidungen zustande kommen, wird beim Bedingungsüberdeckungstest jede Entscheidung genauer betrachtet. Das bedeutet, dass jede Teilbedingung separat betrachtet wird. Die Testfälle müssen, um eine vollständige Überdeckung zu erreichen, jede Teilbedingung zumindest einmal mit den Wahrheitswerten Wahr und Falsch belegen.

$$\text{Einfache Bedingungsüberdeckung} = \frac{\text{Gesamtanzahl der Wahrheitswerte}}{2 * \text{Anzahl der Teilbedingungen}}$$

Pfadüberdeckung

Leider lassen sich viele Fehlerzustände in Programmen erst aufdecken, wenn man bestimmte Anweisungen in einer bestimmten Sequenz ausführt. Diese Sequenz wird auch als Pfad durch ein Programm bezeichnet. Um daher eine theoretisch vollständige Überdeckung zu erreichen, müssten alle möglichen Pfade im Programmcode ausgeführt werden. In der Praxis ist dies jedoch auf Grund der Komplexität in der Regel nicht möglich. Vor allem durch die Kombination von Entscheidungen und Schleifen wächst die Anzahl der möglichen Pfade ins scheinbar Unendliche an. In der Praxis werden daher nur überschaubare Komponenten solchen Überprüfungen unterzogen.

$$\text{Pfadüberdeckung} = \frac{\text{Anzahl ausgeführter Pfade}}{\text{Gesamtanzahl der Pfade}}$$

Die vier bisher genannten Überdeckungen liefern keine konkrete Herangehensweise, wie mit Schleifen umzugehen ist, beziehungsweise ob und wie diese in den einzelnen Überdeckungsgraden berücksichtigt werden. Eine Schleife kann als Aneinanderreihung von mehreren Anweisungen gesehen werden, jedoch ist die Anzahl der tatsächlichen Anweisungen erst zur Laufzeit bekannt. Genau diese Anzahl an Wiederholungen kann zu einem Fehlverhalten führen, deshalb sollten auch Schleifen in die strukturellen Tests aufgenommen werden. Im ISTQB Advanced Level Lehrplan [13] ist der Schleifentest lediglich mit dem Linear Code Sequence and Jump (LCSAJ) Verfahren [46] kurz erwähnt, jedoch nicht weiter ausgeführt. Hehn liefert in seinem Artikel [36] eine Vorschau auf die neue Version des ISTQB Advanced Level Lehrplanes (Version 2012, wurde noch nicht offiziell herausgegeben) und erwähnt, dass das LCSAJ in der Praxis nicht relevant ist und daher nicht mehr Teil des Lehrinhaltes ist. Daher findet sich im neuen Lehrplan, der Mitte 2013 herausgegeben wird, keine spezielle Technik, die Überdeckungsgrade von Schleifen explizit berücksichtigt.

Schleifenüberdeckung

Schleifen im Programm erzeugen unendliche Pfade oder zumindest eine sehr große

Anzahl an möglichen Pfaden. Da es in den meisten Fällen zu hohen Aufwand erzeugen würde, all diese Pfade mit Testfällen abzudecken, sollten zumindest folgende drei Fälle, wie eine Schleife durchlaufen werden kann, abgedeckt sein:

1. Die Bedingung der kopfgesteuerten Schleife ist so, dass die Anweisungen innerhalb der Schleife nicht ausgeführt werden.
2. Die Schleife läuft ein- bis i-mal durch, wobei i < n und n die maximale Anzahl der Durchläufe darstellt, die durch die Schleifenbedingung gegeben ist.
3. Die Anweisungen der Schleife werden vollständig – also n-mal – durchgeführt.

Diese Tests beachten allerdings in erster Linie die Bedingungsvariablen der Schleifen, jedoch nicht die Variablen, die in der Schleife verändert werden. Howden erörtert in [42], dass Bedingungen einer Schleife und deren Statements gemeinsam betrachtet werden müssen. Als weiterführende Literatur zu diesem Thema werden Beizer's Werk [10], Pezze und Young's Buch [62] und Bath und McKay's Buch [6] empfohlen.

Zur Messung der Codeüberdeckung von Testfällen hat sich in der Praxis Codecover[2] sehr bewährt. Codecover lässt sich in die Eclipse[3] Entwicklungsumgebung integrieren und zeigt Codezeilen, die nicht durchlaufen werden, farblich an. Das erleichtert den systematischen Aufbau der Komponententests ungemein.

2.1.3.4 Änderungsbezogener Test

Der änderungsbezogene Test wird weit verbreitet auch Regressionstests genannt. Im Laufe des Softwarelebenszyklus (siehe [37]), aber auch schon während der Entwicklung eines Softwaresystems, kommt es zu Änderungen im System und es werden der Software neue Teile und Funktionalitäten hinzugefügt. Zu diesen Änderungen gehört ebenso die Beseitigung von bereits aufgedeckten Fehlern. In all diesen Fällen muss das System erneut getestet werden. Dies betrifft auch bereits getestete und fehlerfreie Teile des Systems. Durch Veränderungen im Code können, durch Neben- bzw. Seiteneffekte, immer auch jene Teile betroffen sein, die zuvor als bereits fehlerfrei bezeichnet wurden. Eine sehr ausführliche Erklärung mit Beispiel ist unter [20] zu finden. In Zuge der Regressionstests sind alle drei bereits genannten Testarten (funktionaler, nicht funktionaler und struktureller Test) zu berücksichtigen. Der Umfang des Regressionstests muss individuell entschieden werden, je nachdem was und wieviel geändert wurde.

2.1.4 Testmethoden

Im agilen und traditionellen Test können die selben Testmethoden angewandt werden. Die Auswahl der Methoden obligt jedoch der Entscheidung des Testers. Nachfolgend werden

[2] Codecover ist ein Open Source Tool zur Messung von Codeüberdeckung. http://www.codecover.org
[3] Eclipse ist eine Open Source Entwicklungsumgebung. http://www.eclipse.org

wichtige Testmethoden kurz erläutert. Für ausführlichere Grundlagen zum Softwaretest und weitere Methoden empfiehlt der Autor [57] und [77].

2.1.4.1 Äquivalenzklassenbildung

Dass bei einem Testfall nicht alle möglichen Varianten getestet werden können, ist bereits bekannt. Aus diesem Grund wird versucht, alle möglichen Eingaben in Äquivalenzklassen einzuteilen. Unter Äquivalenzklasse versteht man Werte, die unterschiedlich sind, welche allerdings vom System auf die gleiche Weise verarbeitet werden.

Werden zum Beispiel drei Eintrittspreise angeboten, abhängig vom Alter der Person (Kind, Erwachsener, Senior), so muss das System alle Kinder gleich behandeln auch wenn diese unterschiedlichen Alters sind. Aus jeder Äquivalenzklasse wird ein Repräsentant gewählt und mit diesem werden die Testfälle durchgeführt. Man beachte, dass auch für ungültige Wertebereiche Äquivalenzklassen gebildet werden sollen, denn in weiterer Folge müssen auch mit diesen Testfälle durchgeführt werden. Ungültige Äquivalenzklassen in die Testfälle aufzunehmen, ist entscheidend, um das Verhalten der Software im Fehlerfall zu prüfen.

Abbildung 2.3 verdeutlicht die Äquivalenzklassen. Das Alter von sechs Personen ist auf der X-Achse aufgetragen. Innerhalb der drei gültigen Äquivalenzklassen (gÄK_1, gÄK_2, gÄK_3) muss sich das Testobjekt gleich verhalten. Um den Testfall durchzuführen, ist es ausreichend, pro Äquivalenzklasse mit einem Wert zu testen. Die beiden roten Bereiche zeigen die beiden ungültigen Äquivalenzklassen (uÄK_1, uÄK_2), da Personen ein Alter größer 0 haben müssen, beziehungsweise laut Spezifikation davon ausgegangen wird, dass das maximale Alter bei 110 Jahren liegt.

Abbildung 2.3: Veranschaulichung von Äquivalenzklassen

In der Praxis wird man sich nicht mit einem Repräsentanten pro Äquivalenzklasse zufrieden geben, da bekanntlich an den Grenzen der Äquivalenzklassen häufiger Fehler auftreten. Siehe dazu auch das folgende Kapitel 2.1.4.2 zur Grenzwertanalyse.

2.1.4.2 Grenzwertanalyse

In engem Zusammenhang mit den Äquivalenzklassen steht die Grenzwertanalyse. Wie Myers in [57] schreibt, werden viele Fehler an den Grenzen der Äquivalenzklasse gemacht. Deshalb werden an den Grenzen drei Testfälle durchgeführt: der Grenzwert und jeweils ein Wert über und unter dem Grenzwert. Grenzen müssen nicht immer Zahlenwerte sein: Ebenso stellt eine Eingabefeld mit 30 Zeichen eine Grenze dar oder ein virtueller Warenkorb, der mit maximal zehn Artikeln gefüllt werden kann. Die entsprechenden Grenzwerte für das Beispiel in Abbildung 2.3 sind in Tabelle 2.1 zu finden.[4]

Äquivalenzklasse	Grenzwerte	
	untere	obere
uÄK_1		-1
		0
		1
gÄK_1	-1	12
	0	13
	1	14
gÄK_2	12	64
	13	65
	14	66
gÄK_3	64	109
	65	110
	66	111
uÄK_2	109	
	110	
	111	

Tabelle 2.1: Grenzwerttabelle

In diesem Beispiel können die beiden ungültigen Äquivalenzklassen (uÄK_1, uÄK_2) an der unteren bzw. oberen Grenze mit den Minimal- bzw. Maximalwerten (mit deren jeweiligen Vorgängern und Nachfolgern) des verwendeten Datentyps ergänzt werden, jedoch müssen dafür ebendiese aus dem Quellcode eruiert werden.

2.1.4.3 Ursache-Wirkungs-Graph Analyse

Bath und McKay beschreiben in [6], wie man Entscheidungstabellen und Ursache-Wirkungs-Graphen einsetzen kann, um systematisch Testfälle zu ermitteln. Es werden mögliche Eingangszenarien (Ursachen) ermittelt und die dazugehörigen Auswirkungen. Diese werden üblicherweise in einer Tabelle dargestellt. Tabelle 2.2 zeigt eine solche Darstellung anhand eines einfachen Beispiels.

Ein Personentransport unterscheidet seine Passagiere nach drei Altersklassen (Baby, Kind und Erwachsener). Babies wird kein separater Sitzplatz zugewiesen. Erwachsene Passa-

[4] Es wird davon ausgegangen, dass das Alter als Zahl ohne Dezimalstelle eingegeben wird

giere haben die Möglichkeit, eine Stammkundenkarte zu erwerben. Regelmäßig reisende Stammkunden werden zu Statuskunden, was ihnen weitere Vorteile bringt, wie zum Beispiel die Mitnahme von einem weiteren Gepäckstück.

Eingangsbedingungen	1	2	3	4	5	6	7	8
Baby	X							
Kind		X						
Erwachsener			X	X	X			
Stammkunde				X	X			
Statuskunde					X			
Handlungen / Wirkungen								
Sitzplatz zuweisen	N	J	J	J	J			
Gutschrift Stammkundenprogramm	N	N	N	J	J			
erlaubte Gepäckstücke	0	1	1	1	2			

Tabelle 2.2: Beispiel einer Entscheidungstabelle für eine Ursachen-Wirkungs-Analyse

In der Praxis lässt sich eine solche Ursachen-Wirkungs-Analyse nur schwer realisieren, da bei zu vielen Eingangsbedingungen die Übersicht leicht verloren geht. Um diese Methode auch in komplexen Systemen einsetzen zu können, müssen die Faktoren auf einige wenige, priorisierte Bedingungen reduziert werden. Dabei werden oft die Bedingungen nach Risiko ausgewählt. Eine weitere Reduzierung erfolgt durch Entfernen von unmöglichen Kombinationen, also jene Kombinationen, die durch die Spezifikation ausgeschlossen sind.

2.1.4.4 Zustandsbasiertes Testen

Beim zustandsbasierten Testen wird die zu testende Software in ihren möglichen Zuständen und deren Zustandsübergängen betrachtet, wie Bath und McKay in [6] näher beschreiben. Diese Zustände und die dazugehörigen Übergange können in einem Zustandsübergangsdiagramm dargestellt werden.

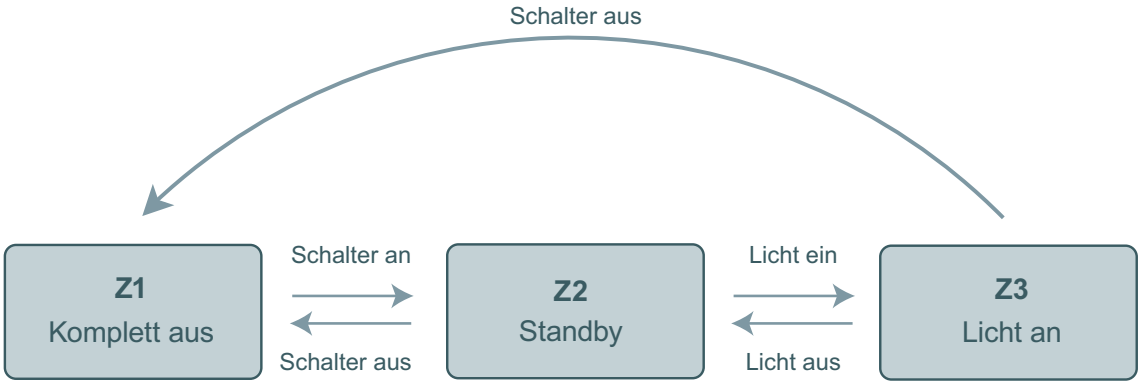

Abbildung 2.4: Zustandsübergangsdiagramm eines Funklichtschalters

Abbildung 2.4 stellt ein einfaches System einer Funkleuchte dar. Diese hat einen Schalter am Gehäuse mit dem die Funkleuchte komplett vom Netz getrennt wird und somit auch

nicht mehr auf Befehle via Fernbedienung reagiert. Wenn dieser Schalter jedoch aktiviert ist, kann mit der Fernbedienung das Licht ein und ausgeschaltet werden.

In Tabelle 2.3 werden die Zustandsübergänge nochmals in Tabellenform dargestellt.

	Ü1	Ü2	Ü3	Ü4	Ü5
Ausgangszustand	Z1	Z2	Z2	Z3	Z3
Ereignis	Schalter an	Schalter aus	Licht ein	Licht aus	Schalter aus
Wirkung	Standby	Komplett aus	Licht an	Standby	Komplett aus
Endzustand	Z2	Z1	Z3	Z2	Z1

Tabelle 2.3: Zustandsübergangstabelle 0-Switch

In weiterer Folge werden dann die Mehrfachübergänge erfasst und getestet. Im ersten Schritt, der bereits erläutert wurde, konnten fünf Übergänge identifiziert werden. Das Erfassen von einfachen Übergängen wird auch 0-Switch-Überdeckung genannt. 0 ist die Anzahl der Übergänge, die zwischen dem Ausgangs- und dem Endzustand liegen. Die nächste Stufe ist die 1-Switch-Überdeckung, bei der alle Zustandsübergänge mit jeweils einem Zwischenzustand berücksichtigt werden. Daher wird Tabelle 2.3 erweitert und man erhält Tabelle 2.4.

	Ü6	Ü7	Ü8	Ü9	Ü10	Ü11	Ü12	Ü13
Ausgangszustand	Z1	Z1	Z2	Z2	Z2	Z3	Z3	Z3
Ereignis	Schalter an	Schalter an	Schalter aus	Licht ein	Licht ein	Licht aus	Schalter aus	Licht aus
Wirkung	Standby	Standby	Komplett aus	Licht an	Licht an	Standby	Komplett aus	Standby
1-Switch Zustand	Z2	Z2	Z1	Z3	Z3	Z2	Z1	Z2
Ereignis 2	Schalter aus	Licht ein	Schalter an	Licht aus	Schalter aus	Licht ein	Schalter an	Schalter aus
Wirkung 2	Komplett aus	Licht an	Standby	Standby	Komplett aus	Licht an	Standby	Komplett aus
Endzustand	Z1	Z3	Z2	Z2	Z1	Z3	Z2	Z1

Tabelle 2.4: Zustandsübergangstabelle 1-Switch

2.1.5 Testautomatisierung

Testautomatisierung ist die Durchführung von Testfällen durch einen Automaten und ist ein weiterer wichtiger Bestandteil des Softwaretests. In iterativen Softwareentwicklungsmodellen sollen in jedem Release essentielle Testfälle durchgeführt werden, um sicherzustellen, dass durch Änderungen und Erweiterungen keine Fehler in bereits vorhandene Funktionen eingebaut wurden. Die Automatisierung dieser Testfälle ist ein wesentlicher Schritt zur Qualitätssicherung, da somit in jedem Release durch die ausgeführten Testfälle die entsprechende Funktionalität gewährleistet wird.

Prinzipiell ist Testautomatisierung in jeder Teststufe einsetzbar. Jedoch sind die Herangehensweise sowie die verwendeten Werkzeuge sehr unterschiedlich. Folgende acht Faktoren, die bei der Entscheidung für oder gegen eine Testautomatisierung berücksichtigt werden müssen, haben Seidl, Baumgartner und Bucsics in [72] aufgelistet:

Regressionstests
: Testfälle, die oft durchgeführt werden müssen, rechnen sich am ehesten für den automatisierten Regressionstest. Sobald die Testfälle automatisiert wurden, läuft die Ausführung größtenteils ohne personellem Aufwand. Der Initialaufwand für die Automatisierung rentiert sich jedoch erst nach einer bestimmten Anzahl an Testwiederholungen und ist somit für häufig getestete Testfälle am sinnvollsten. Wurden vor einer Wiederholung des Tests keine Änderungen an dem zu testenden System durchgeführt, hat sie jedoch keinen Effekt und reduziert nur scheinbar die Kosten. Weiters ist zu bedenken, dass Änderungen an dem zu testenden System auch Auswirkungen auf die Testautomatisierung haben können und mitunter Aufwand für die Anpassung der Automatisierungsskripte hervorrufen.

Lifecycle
: Die Lebensdauer des zu testenden Systems ist immer zu berücksichtigten. Systeme mit einer sehr kurzen Entwicklungsdauer können mitunter aus wirtschaftlichen Gründen für eine Testautomatisierung ungeeignet sein, da nur wenige Testdurchführungen stattfinden werden.

Entwicklungszyklus
: Die Anzahl der Entwicklungszyklen und die damit verbundenen Testdurchführungen spielen ebenso eine entscheidende Rolle bei der Entscheidung für die Testautomatisierung. Je mehr Entwicklungszyklen mit Testdurchläufen, desto eher ist eine Testautomatisierung wirtschaftlich attraktiv.

Bedienbarkeit
: Einerseits gibt es immer wieder Testabläufe, die nur mit sehr hohem Aufwand automatisierbar sind oder technisch gar nicht automatisierbar sind. Andererseits gibt es aber Testabläufe, die manuell nur schwer realisierbar sind, wie beispielsweise Vergleiche von großen Tabellenkalkulationen oder dem Test von Schnittstellen.

Kombinatorik
: Bei Testautomatisierung ist es leicht möglich, die Testdurchführungen untereinander zu kombinieren, wodurch eine höhere Testabdeckung erreicht werden kann.

Fehleranfälligkeit
Müssen Testabläufe immer wieder manuell getestet werden, können durch die sinkende Aufmerksamkeit der Tester Fehler passieren. Ein Testautomat wird niemals müde und erledigt die Testdurchführungen immer exakt gleich.

Nicht funktionale Tests
Gerade nicht funktionale Tests, wie Lasttests, können einfacher mit Testautomaten durchgeführt werden, denn ein Lasttest mit 1000 Benutzern ist organisatorisch kompliziert und sehr kostenintensiv.

Fehlerfindung
Bei der Testautomatisierung muss sichergestellt sein, dass Änderungen am Testobjekt die automatische Testdurchführung nicht zu stark beeinträchtigen.

Wenn man sich nun für die Testautomatisierung entschieden hat, sind folgende zwölf wichtige Eigenschaften, definiert von Meszaros, Smith und Andrea in [55], entscheidend:

Prägnanz So einfach wie möglich, jedoch nicht einfacher als das.

Reporting Ergebnisse werden automatisch zusammengefasst, ohne dass sie eine Person bearbeiten muss.

Wiederholbarkeit Ein wiederholter Ablauf der Testfälle muss gewährleistet sein, ohne dass manuelle Vor- oder Nacharbeit notwendig ist.

Robustheit Die Ergebnisse der Testläufe sind so gestaltet, dass diese nicht von externen Quellen beeinflusst werden.

Vollständigkeit Die Tests müssen alle spezifizierten Anforderungen der Software abdecken.

Notwendigkeit Alle Testschritte des Tests sollen notwendig sein, um das gewünschte Verhalten zu testen.

Verständlichkeit Jeder Schritt in den Testfällen ist klar und verständlich.

Effizienz Die Testläufe benötigen eine vertretbare Zeitdauer.

Eindeutigkeit Jede Fehlerwirkung muss eindeutig einer nicht erfüllten Funktionalität zuordenbar sein. Eine Fehlerwirkung im Komponententest erzeugt eine aus drei Parametern bestehende Definition (= defect triangulation), welche es erleichtert, die Ursache des Fehlers einzugrenzen.

Unabhängigkeit Jeder Testfall kann als eigenständiger Test oder gesammelt als Testsuite durchgeführt werden – in beliebiger Reihenfolge.

Wartbarkeit Testfälle sollen leicht verständlich und bei Bedarf erweiterbar oder veränderbar sein.

Nachvollziehbarkeit Der Testfall muss zu und von dem zu testenden Code sowie zu und von der entsprechenden Anforderung zuordenbar sein.

Die Auswahl der Werkzeuge für die Testautomatisierung soll genau und sorgfältig durchgeführt werden. Viele Werkzeughersteller versprechen mit ihren Werkzeugen die Lösung für alle Probleme und meinen gleichzeitig den Wartungsaufwand vollständig eliminiert zu haben. Für jedes Softwaresystem gibt es jedoch unterschiedliche Anforderungen an die Testautomatisierungswerkzeuge, daher muss eine Werkzeugauswahl immer genau mit den Gegebenheiten des zu testenden Systems geprüft werden. Ebenso sind für die unterschiedlichen Teststufen unterschiedliche Werkzeuge notwendig.

Oft stellt sich die Frage, welche zusätzlichen Fähigkeiten ein Testautomatisierer mitbringen sollte. Die klare Antwort ist: In der Testautomatisierung wird sehr oft auch programmiert. Aktuelle Tools reduzieren den Programmieraufwand auf ein Minimum, allerdings wird auf Grund der Individualität der zu testenden Systeme die Programmierung immer ein essentieller Teil bleiben. Denn durch das Hinzufügen von weiterem Programmcode zur Automatisierungssoftware wird ein zu testendes System oftmals erst automatisiert testbar. Daher ist es wichtig, dass der Testautomatisierer Kenntnisse in der verwendeten Programmiersprache des Automatisierungswerkzeugs mitbringt.

2.2 Agile Softwareentwicklung

2.2.1 Bedeutung von Agilität

Die erste Definition ist im agilen Manifest [9] niedergeschrieben. Coldewey hat die vier Merkmale des agilen Manifests [9] in seiner Kolumne [18] wie folgt ins Deutsche übersetzt:

1. Einzelpersonen und Interaktion sind [...] wichtiger als Prozesse und Werkzeuge.
2. Laufende Systeme sind [...] wichtiger als umfangreiche Dokumentationen.
3. Zusammenarbeit mit dem Kunden ist [...] wichtiger als Vertragsverhandlungen.
4. Die Fähigkeit, auf Änderungen zu reagieren ist [...] wichtiger als das Verfolgen eines Plans.

Diese Prinzipien haben sich daraus gebildet, dass immer komplexere Softwareprojekte umgesetzt werden und diese sich nicht mehr mit den Prozessen und Methoden der 1990er Jahre umsetzen lassen, ohne dass man in tausenden Seiten Dokumenten die Übersicht verliert. In den konservativen Prozessmodellen gibt es außerdem immer wieder die Diskussion, wie mit „Change Requests“ (= Änderungen in der Projektspezifikation) umgegangen werden soll. Diese Change Requests machen es teilweise notwendig, wieder zum ersten Schritt der Analyse zurückzukehren, was wiederum Kosten verursacht, über die sich Kunde und Auftragnehmer (= Dienstleister) einig werden müssen. Oft wird dadurch das Verhältnis zwischen Kunde und Dienstleister stark belastet, da Änderungen nur mehr mit erhöhtem Aufwand umgesetzt werden können; Spezifikation, Analyse, Dokumentation, etc. müssen oft geändert werden.

Die Schaffung neuer Software ist jedoch großteils ein kreativer Prozess, der sich nicht vollständig durch strikte Pläne steuern und kontrollieren lässt. Je kurzfristiger Pläne sind,

desto größer die Chance, dass diese auch eingehalten werden können. Je weiter die detaillierte Planung in die Zukunft reicht, desto mehr Faktoren nehmen darauf Einfluss, wie zum Beispiel Marktentwicklung, Mitarbeiter, Budgetsituationen, etc. Durch diese Veränderungen müssen Pläne wieder geändert und angepasst werden oder teils komplett neu entwickelt werden. Dadurch entstehen Kosten, um den Projektplan, das Team und den Kunden darauf abzustimmen. In der agilen Softwareentwicklung sind Pläne gut für das nächste Release[5]/ Inkrement[6]/ Sprint[7]. Diese sind realistisch und können zu einem hohem Grad eingehalten werden, da die Planung nur für einen kurzen Zeitraum getätigt wurde und diese zu einem hohen Grad nicht verändert werden müssen, da äußere Einflüsse erst spätere Zyklen beeinflussen. Essentiell bei agilen Methoden, wie beispielsweise Scrum, ist, am Ende einer Iteration ein lauffähiges und getestetes Produkt vorweisen zu können.

2.2.2 Erfolgsfaktor Agilität

Der wichtigste Erfolgsfaktor agiler Projekte sind die handelnden Personen. Sie sind der Schlüssel zum Projekterfolg und daher gelten in agilen Projekten diverse Leitsätze, die diese Grundeinstellung unterstützen.

Gernert hat in [28] dazu folgende Punkte herausgearbeitet:

- Ein motiviertes Team ist besser als perfekt formalisierte Regeln.
- Gute Kommunikation im Team bringt mehr als formalisierte Berichterstattung.
- Erreichte Ziele werden stärker gewichtet als sture Abhandlungen von Vorgaben.
- Zahlreiche Änderungen während der Projektdurchführung werden akzeptiert und nicht an Plänen festgehalten.
- Investition in die Verbesserung des Vertrauens im Team ist mehr wert als die Durchführung von Kontrollverfahren.
- Erfahrungen aus vorigen Projekten beeinflussen aktuelle und zukünftige Projekte stärker als Prozessmodelle und theoretische Abläufe.
- Angemessene Vorgehensweisen sind klüger als extreme und einschneidende Handlungen.
- Risiken werden vorher aktiv gemanagt statt Krisen danach bewältigt.

Zahlreiche definierte Prozesse waren früher das Erfolgsrezept für die Durchführung der Projekte. Immer mehr Projektmanager und -leiter sind jedoch der Auffassung, dass diese immer an das Projekt und die Situation angepasst werden sollten, um das Ergebnis maximal zu optimieren.

[5] Ein Release ist eine Version eines Softwareprogrammes.
[6] Als Inkrement wird ein zum Teil fertiges aber lauffähiges Softwareprogramm bezeichnet.
[7] Sprint ist ein Planungsabschnitt während der Entwicklung, wie er in dem Vorgehensmodell Scrum verwendet wird.

2.2.3 Agile Vorgehensmodelle

Die bekanntesten agilen Vorgehensmodelle in der Softwareentwicklung sind Scrum, eXtreme Programming, Crystal, Feature Driven Development, Test Driven Development und Behaviour Driven Development. Einige von den genannten sind keine reinen Vorgehensmodelle, sondern Sammlungen von Praktiken. Alle Modelle und Sammlungen werden hier kurz beschrieben, um einen Überblick über die wichtigsten Herangehensweisen zu geben.

Vorher muss jedoch zwischen traditionellen und agilen Modellen unterschieden werden. Bunse und Knethen unterscheiden in [12] Vorgehensmodelle nach Flexibilität und Detailgrad. Klassische Vorgehensmodelle, wie zum Beispiel das V-Modell, beschreiben die Prozesse in einem sehr hohen Detailgrad. Im Gegensatz dazu bietet beispielsweise Scrum ein flexibles Modell für die Umsetzung. Tabelle 2.5 stellt einige der deutlichsten Unterschiede von traditionellen und agilen Modellen einander gegenüber und soll die Extreme der beiden Herangehensweisen verdeutlichen. Natürlich liegen die vorgestellten und auch verschiedene traditionelle Modelle zwischen diesen Extremen.

Für alle Vorgehensmodelle gilt, dass diese an die jeweilige Projektsituation angepasst werden müssen. Damit die Anpassungen aber nicht zu Problemen führen, müssen die Verantwortlichen mit den jeweiligen Modell sehr gut vertraut sein.

Klassische Modelle	Agile Modelle
Strikte Trennung der Projektphasen	Alle Phasen in jeder Iteration
Änderungen bedeuten Phasenwiederholung	Änderungen sind geplant
Vollständige Planung bis Projektende	Planung nur für die nächste Iteration
Am Ende der Entwicklungsphase steht ein lauffähiges Produkt zur Verfügung	Nach jeder Iteration steht ein lauffähiges Produkt zur Verfügung

Tabelle 2.5: Vergleich von klassischen und agilen Modellen der Softwareentwicklung

2.2.4 User Stories

User Stories werden in vielen agilen Methoden verwendet, um Anforderungen zu repräsentieren, welche dann als Grundlage für Entwicklung und Test herangezogen werden. Auch für nicht funktionale Anforderungen (z.b. Performance, Sicherheit, Benutzerfreundlichkeit) können User Stories verfasst werden. Diese werden in der Literatur oft als „Beschränkungen“ bezeichnet, es gelten jedoch die selben Regeln wie für User Stories. Cohn nennt in [17] die wichtigsten drei Bestandteile einer User Story.

1. Beschreibung der Story, um zu dokumentieren, was umgesetzt werden soll.
2. Auflistung von wichtigen Punkten, die zur Verständlichkeit der User Story beitragen und notwendige Details herausarbeiten.
3. Definition der sogenannten Akzeptanztests, welche detailliert erklären, wann eine Story vollständig umgesetzt ist. Hier sollen auch weitere wichtige, individuelle Details für die Durchführung des Tests angeführt werden.

Wie Cohn in [17] hinweist, sollten auch folgende Punkte berücksichtigt werden:

Wer? Für welche Rolle ist die User Story erstellt worden

Was? Was ist der Zweck und wie verändert dieser mein System?

Warum? Warum möchte man diese User Story ausführen?

Daraus ergibt sich folgender formaler Aufbau einer User Story nach Waters in [81], welcher die niedergeschriebenen User Stories gleichzeitig klar definiert und gut lesbar macht:

Als [Rolle] möchte ich [Ziel], damit ich [Begründung].

Dabei erfüllt nach Cohn [17] eine optimal beschriebene User Story folgende sechs Eigenschaften:

Unabhängig
Die Story sollte möglichst unabhängig von anderen Stories sein. Dies erleichtert die Planung ungemein.

Verhandelbar
Story Cards sind keine schriftlichen Verträge oder Anforderungen, sondern eine Gedächtnisstütze.

Werthaltig für Käufer oder User
Stories, die sich auf Technik oder Vorteile für Entwickler begrenzen, sollen umformuliert werden, um den entsprechenden Nutzen für Kunden oder User herauszuarbeiten. Damit können diese Stories dann vom Kunden ordnungsgemäß bewertet werden.

Schätzbar
Der Umfang einer Story muss schätzbar sein.

Klein
Es gibt keine allgemein gültige Regel, die die Größe einer Story festlegt, da dies von den handelnden Personen abhängt. Stories, die zu komplex sind oder zu viele Anwendungsfälle beinhalten, sollen jedoch auf mehrere Stories aufgeteilt werden, das macht das Entwickeln und Testen einfacher.

Testbar
Stories müssen so geschrieben sein, dass diese testbar sind. Oft werden nicht funktionale User Stories mit unpräzisen Angaben erstellt, sodass diese nicht getestet werden können.

Der Umfang einer User Story muss stets so gewählt werden, dass nur notwendige Details beschrieben werden, aber trotzdem Entwickler alle Details kennen, um die User Story sinnvoll umzusetzen. Hier gilt: so viel wie notwendig, so wenig wie möglich.

Cohn [17] empfiehlt User Stories, die zu viel Inhalt oder zu viele Tätigkeiten enthalten, als epische Story anzusehen und in mehrere kleinere Stories zu unterteilen. Weiters empfiehlt er, mit den eigentlichen Zielen zu beginnen und von diesen Ziel-Stories ausgehend einzelne User Stories zu extrahieren. Lauesen [50] beschreibt weiters die Idee der geschlossenen Anforderungen. Eine geschlossene User Story ist eine Story, die ein Ziel verfolgt und dieses auch erreichen kann. Geschlossene User Stories haben auch einen psychologischen Effekt auf das Entwicklerteam, da diese geschlossenen und umgesetzten Stories nach der Abnahme als komplett erledigt betrachtet werden und dem Team das positive Gefühl vermitteln, etwas erreicht zu haben. Somit empfielt es sich, einige User Stories in mehrere kleinere, dafür aber geschlossene, Stories aufzuteilen.

Gegenüber anderen Methoden, Anforderungen zu spezifizieren, bieten User Stories wesentliche Vorteile, weswegen sie für agile Modelle bevorzugt werden. Cohn fasst die Vorteile wie folgt zusammen (vgl. [17]):

- User Stories stellen die verbale Kommunikation in den Vordergrund.
- User Stories sind allgemein verständlich.
- User Stories haben die richtige Größe für die Planung.
- User Stories eignen sich für die iterative Softwareentwicklung.
- User Stories ermutigen dazu, Details zurückzustellen.
- User Stories unterstützen opportunistisches Design.
- User Stories fördern partizipatorisches Design.
- User Stories bauen implizites Wissen auf.

Weiters definiert Cohn folgende Probleme, die im Zusammenhang mit User Stories auftreten können:

Abhängigkeiten zwischen User Stories
: Abhängigkeiten zwischen verschiedenen User Stories verhindern unter Umständen, dass einzelne Stories abgeschlossen werden können.

Ständiges Aufteilen von User Stories
: Zu allgemein gehaltene User Stories müssen in scheinbar unendlich viele Stories aufgeteilt werden.

Probleme beim Priorisieren
: Der Product Owner hat, auf Grund der zwischenzeitlich gelieferten Inkremente, Schwierigkeiten, die User Stories zu priorisieren, und ändert deshalb ständig seine Meinung.

Entwickler fügen Features hinzu, die nicht geplant sind
: Entwickler neigen dazu, Features zu bauen, die aus ihrer Sicht nützlich sind, die aber vom Kunden gar nicht benötigt werden. Dies kostet unnötig Zeit und durch Seiteneffekte können dadurch weitere Probleme und Fehler auftreten.

Zu früh werden Stories, die die Benutzeroberfläche betreffen, behandelt
Veränderungen in der Benutzeroberfläche sind viel aufwändiger, als Änderungen bei funktionellen Anforderungen. Die Benutzeroberfläche sollte erst implementiert werden, wenn das Design endgültig feststeht.

Zusammenfassend lassen sich aus der Sicht des Autors einige in der Praxis häufig auftauchende entscheidende Erfolgsfaktoren ableiten: User Stories sollten voneinander unabhängig sein, um diese in beliebiger Reihenfolge entwickeln zu können. Dies wird nicht immer möglich sein, sollte aber zumindest versucht werden. User Stories müssen außerdem testbar sein, um die Qualität im Projekt auf einem stetig hohen Niveau zu halten. Zu viele Details lenken vom Wesentlichen einer Story ab, denn die Kommunikation zwischen Entwickler und Kunde soll nicht durch zu viele Details leiden und womöglich „eingespart“ werden. Zu umfangreiche Stories können in mehrere kleinere Stories unterteilt werden.

2.2.5 Scrum

„Scrum ist viel mehr als nur ein Prozessmodell. Es ist eine Philosophie mit wenigen, klar definierten Regeln, um ein Ziel effektiv zu erreichen“ [32]. Der Begriff Scrum bedeutet übersetzt „Gedränge“ und kommt aus dem Rugby. Gloger interpretiert dies in [32] mit dem Zusammenhalt eines Rugby Teams, das versucht ein Ziel zu erreichen und dabei diszipliniert wenige einfache Regeln einhalten muss.

Scrum beschreibt ein komplette Entwicklungsmethode für Softwareprojekte (vgl. [43]). Für diese Methode werden Rollen und Artefakte für die Umsetzung definiert, um einen erfolgreichen Ablauf der Prozesse zu gewährleisten.

2.2.5.1 Ablauf in Scrum

Abbildung 2.5 zeigt den Scrumablauf nach Mountain Goat Software [76]. Die Anforderungen und Wünsche des Kunden an das Softwareprodukt werden in sogenannten User Stories (siehe 2.2.4) verfasst. Diese werden vom Product Owner priorisiert und im Product Backlog abgelegt (siehe Abschnitt 2.2.5.3).

Für jeden Sprint – dieser hat in der Regel eine Dauer von zwei bis vier Wochen – werden die zu entwickelnden Features ausgewählt (= Sprint Backlog, siehe Abschnitt 2.2.5.3), vom Scrum Team umgesetzt und als lauffähige Software präsentiert. Alle Aufgaben werden vom Team in Eigenverantwortung wahrgenommen. Täglich wird ein kurzes Scrummeeting (= Daily Scrum, siehe 2.2.5.6) abgehalten, um die Kommunikation im Team aufrecht zu erhalten und jedes Teammitglied über den Stand aller Arbeiten zu informieren.

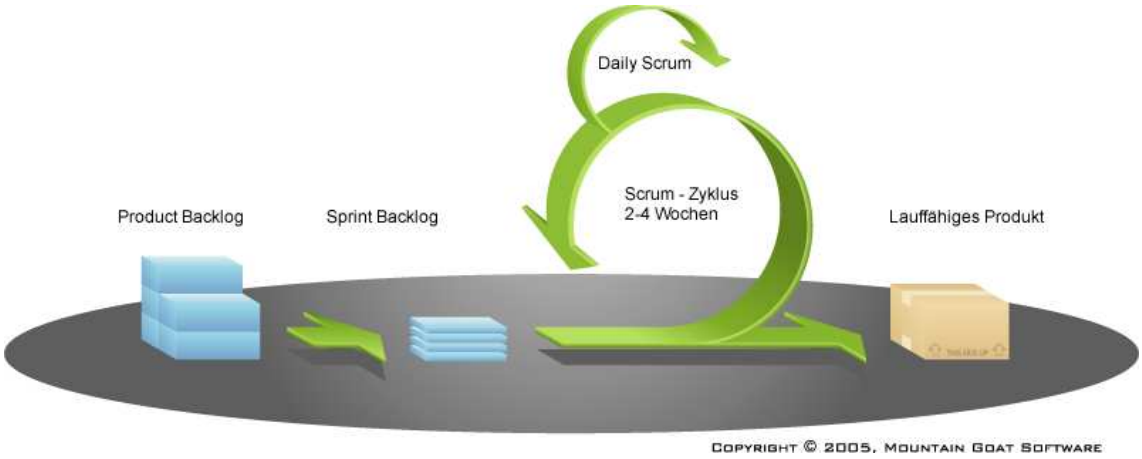

Abbildung 2.5: Scrum Ablauf nach Mountain Goat Software

2.2.5.2 Rollen in Scrum

Pichler definiert in [63] die Rollen in Scrum wie folgt:

Product Owner

Der **Product Owner**[8], beziehungsweise der Kunde, hat eine zentrale Aufgabe in Scrum. Durch seine Tätigkeit wird der Erfolg des Scrum Projektes am stärksten beeinflusst. Anforderungen werden nicht mehr zu Beginn des Projektes festgeschrieben, sondern werden im Laufe des Projektes vom Product Owner bestimmt und gelenkt. Die ständige Verfügbarkeit des Product Owners ist essentiell für den Erfolg des Projektes, vor allem dann, wenn das zu entwickelnde Produkt ein neues oder komplexes Produkt ist.

Team

Das **Team** erledigt alle Arbeiten, die für die Umsetzung des Projektes notwendig sind, um am Ende eines Sprints ein lauffähiges Produkt zu haben. Das Team hat sehr viele Freiheiten, wie die Aufgaben umgesetzt werden. Ebenso wird durch das Team selbst der Umfang jedes Sprints bestimmt. Diese Planung findet im Sprint Planning I statt.

Scrum Master

Der **Scrum Master** ist in erster Linie dafür verantwortlich, dass Scrum richtig umgesetzt wird. Weiters unterstützt er das Team und sorgt dafür, dass sich externe Einflüsse nicht negativ auf das Team auswirken. In den Meetings des Scrums dient er als Moderator und als Teamcoach, um dem Team eine möglichst effiziente Arbeitsweise zu ermöglichen. Es liegt somit in der Natur der Aufgabe, dass diese Person sehr gute soft und social skills mitbringt.

[8] Der Product Owner muss nicht nur aus einer natürlichen Person bestehen. Oft nehmen mehrere Personen diese Rolle ein.

Schwaber K. definieren diese drei Rollen in ihrem „The Scrum Guide“ [70]. Gloger erweitert in seinem Buch [32] die Rollen um drei weitere, diese werden hier allerdings vernachlässigt, da diese nicht zwingend notwendig sind.

2.2.5.3 Artefakte des Scrum-Modells

Pichler [63] definiert die Artefakte in Scrum wie folgt:

Product Backlog

Anforderungen in Scrum werden im sogenannten **Product Backlog** abgelegt. Dies ist eine sich ständig verändernde Liste, da sich Anforderungen im Laufe der Entwicklung ändern können, neue hinzugefügt und vorhandene gestrichen werden. Details zu grob formulierten Produktanforderungen werden mitunter erst dann ausformuliert, wenn ihr Umfang geschätzt werden muss. Jede dieser Anforderungen wird vom Product Owner priorisiert, diese Priorisierung bildet in Folge die Grundlage für die Auswahl der Anforderungen für den nächsten Sprint. Jedes Teammitglied bewertet dann mit sogenannten Storypoints den Aufwand für die Umsetzung einer solchen Anforderung. Somit ist für den Product Owner ersichtlich, wie aufwändig jede Anforderung ist. Oft werden für die Vergabe der Storypoints Zahlen der Fibonacci[9] Folge verwendet, um eine eindeutige Reihung der Anforderungen nach Aufwand zu erreichen.

Sprint Backlog

Der **Sprint Backlog** [70] beinhaltet genau jene Anforderungen, die vom Scrum Team ausgewählt wurden, um diese im aktuellen Sprint zu realisieren. Fehler, die in einem Sprint erkannt werden, kommen ebenfalls in den Sprint Backlog, da diese behoben werden müssen, um eine Anforderung korrekt umzusetzen. Der aktuelle Fortschritt wird ebenfalls zumindest täglich aktualisiert, um die noch ausständige Arbeit für den Sprint ablesen zu können.

Inkrement

Das **Inkrement** (vgl. [70]) bezeichnet das aktuell verfügbare Softwareprodukt, welches die bereits umgesetzten Anforderungen beinhaltet. Wichtig dabei ist, dass auch alle Anforderungen der vorhergehenden Sprints einwandfrei funktionieren und die Definition of Done zu erfüllen.

2.2.5.4 Definition of Done (DOD)

Nach Schwaber K. [70] und Panchal [60] ist die Definition of Done die Definition des Zeitpunktes an dem eine User Story fertig umgesetzt ist. Sie ist essentiell für ein funktionierendes Scrum Team. Die DOD wird in unterschiedlichen Teams anders formuliert sein, wichtig ist nur, dass diese Definition im Team selbst akzeptiert wird. Empfohlen

[9] Zahlen der Fibonacci Folge haben für diese Bewertung einen klaren Vorteil: Da jedes Teammitglied jede Zahl nur einmal pro Sprint verwenden kann, sind die Summen aus Zahlen der Fibonacci Folge weitestgehend eindeutig reihbar. Bewiesen wird das durch das Zeckendorf-Theorem. Siehe dazu [85] und [84].

wird eine Art Checkliste, welche Aktivitäten durchgeführt werden müssen, um die DOD zu erfüllen. Für DOD gibt es drei Levels:

- DOD für eine spezielle Anforderung (User Story)
- DOD für eine Iteration (alle User Stories, die in dieser Iteration geplant waren)
- DOD für ein Release (tatsächlich ausgeliefertes Produkt bzw. Release Candidate)

2.2.5.5 Sprint Planning

Nach Schwaber [69] findet das Sprint Planning I gemeinsam mit dem Product Owner statt, um die wichtigsten User Stories aus dem Product Backlog auszuwählen, die in den Sprint Backlog übernommen werden.

Im Sprint Planning II entscheidet das Team, wie die zuvor ausgewählten Anforderungen umgesetzt werden.

2.2.5.6 Tägliches Scrummeeting

Das tägliche Scrummeeting soll ein maximal 15 Minuten dauerndes Meeting sein, damit jedes Teammitglied folgende Informationen dem gesamten Team mitteilen kann:

- Was habe ich seit dem letzten Scrummeeting getan (gestern/heute)?
- Was plane ich für heute/morgen (bis zum nächsten Scrummeeting)?
- Was hindert mich an meiner Arbeit, welche Probleme sind aufgetreten?

Die wichtigste Regel ist, dass die Zeit eingehalten wird. Detaillierte Diskussionen zwischen einzelnen Personen sollen nicht während des Scrummeetings stattfinden, sondern danach im direkten Gespräch. Externe Teilnehmer sind nur in passiver Rolle gestattet und nur wenn dies zur Vermeidung von weiteren Meetings dient, das heißt, wenn dadurch Zeit für zusätzlichen Informationsaustausch eingespart wird.

Weiters wird auch der aktuelle Fortschritt im aktuellen Sprint festgehalten. Schwaber K. hatten den Einfall, die noch ausständigen Arbeiten in einem Sprint grafisch darzustellen.

Ein Beispiel für ein solches Burndown Chart ist in Abbildung 2.6 festgehalten. Die geplanten und zu erbringenden Leistungen für den Sprint werden als Soll-Leistung (= estimated storypoints) grün dargestellt. Der täglich erfasste Status ist rot gekennzeichnet. Steigende Storypoints werden durch auftretende Probleme, wie Fehler, Probleme mit Schnittstellen, Infrastrukturprobleme, etc. hervorgerufen, da diese zusätzlichen Arbeitsaufwand nach sich ziehen.

Gloger listet in seinem Scrum Buch [32] weitere in Verbindung mit Scrum verwendete Reports auf, wie Release-Burndown-Chart, Parking-Lot-Chart, Velocity-Chart oder das Task-Up-Chart.

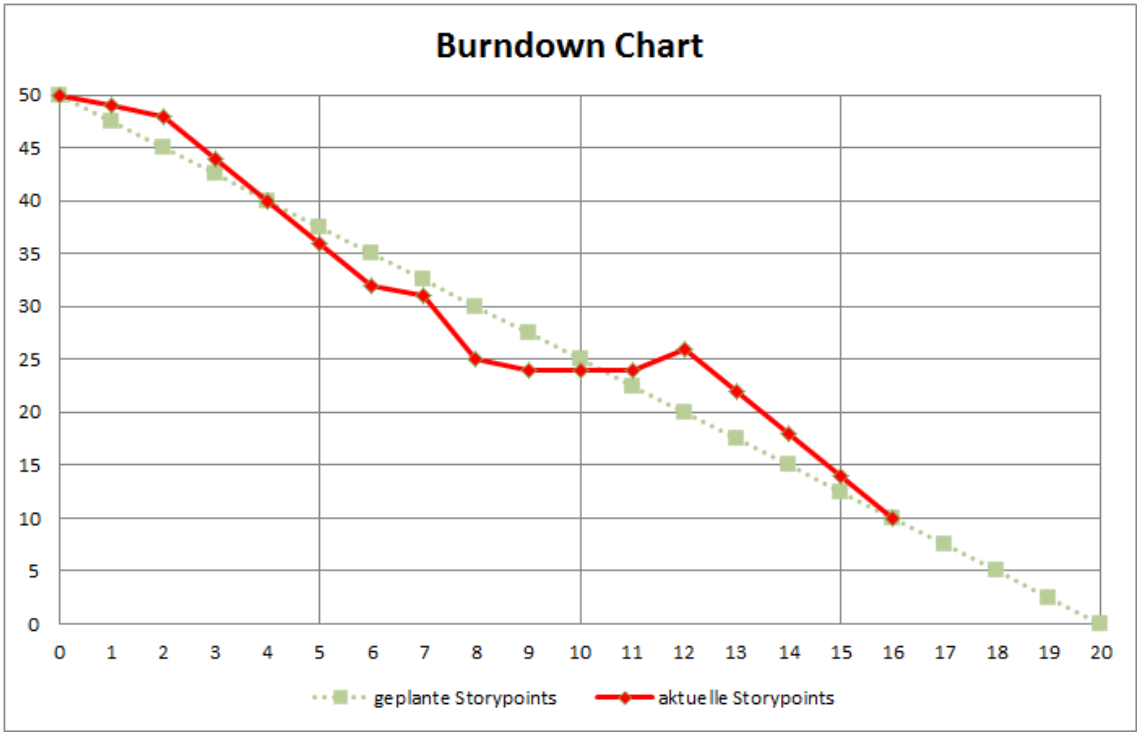

Abbildung 2.6: Burndown Chart

2.2.5.7 Sprint Review

Wie wichtig es ist, einen Sprint richtig abzuschließen, weil dies der Ausgangspunkt für den nächsten Sprint ist, beschreibt Gloger [32] sehr treffend. Am Ende eines Sprints werden die Ergebnisse dem Kunden präsentiert. Dabei werden neue Features demonstriert und gemeinsam mit dem Kunden festgestellt, ob das Sprintziel erreicht wurde. Das Ende eines Sprints (= Sprint Review) definiert außerdem eine klare Deadline für das Team. Zu diesem Zeitpunkt muss die Software lauffähig und getestet sein, um die Funktionalität zu gewährleisten. Für alle Beteiligten wird der Status des Projektes publiziert. Somit wissen alle, wie es um das Projekt steht und ob die Ziele des letzten Sprints erreicht wurden. Gegebenenfalls müssen Probleme durch sofortige Veränderungen gelöst werden, um sie in den folgenden Sprints zu vermeiden und aus Fehlern unmittelbar zu lernen. Oftmals entstehen aus der Präsentation des aktuellen Inkrements neue oder veränderte Anforderungen für das Product Backlog.

2.2.5.8 Sprint Retrospektive

Die Leistung des letzten Sprints wird im Team analysiert, um wichtige Werte für die nächste Sprintplanung zu bekommen. Beispielsweise wird erhoben, wie viele Storypoints umgesetzt werden konnten. Nach einigen Sprints lässt sich somit sehr gut abschätzen, wieviel das Team pro Sprint umsetzten kann. Allerdings darf man niemals Storypoints unter verschiedenen Scrum Teams vergleichen, da Teams die Aufwände immer unterschiedlich bewerten. Verbesserungs- und Änderungsvorschläge für die Arbeitsprozesse werden im Team herausgearbeitet, um diese im nächsten Sprint Planning zu berücksichtigen. Gloger [32] unterstreicht, dass erlangte Erkenntnisse und Änderungen ebenfalls in

den Product Backlog einfließen, dieser wird zu diesem Zeitpunkt erweitert, geändert und eventuell neu priorisiert.

2.2.5.9 Test in Scrum

Ken Schwaber und Jeff Sutherland, die Erfinder von Scrum, haben in ihrem Scrum Guide [70] keine Details niedergeschrieben, wie in Scrum getestet werden soll. Es ist nur definiert, dass der Softwaretest während eines Sprints laufend und parallel zur Entwicklung stattfindet. Mike Cohn [17] ergänzt, dass sich die Tester, basierend auf den Anforderungen, Akzeptanzkriterien herausarbeiten. Diese müssen erfüllt werden, damit die dazugehörige Anforderung am Ende des Sprints erfolgreich abgenommen werden kann. Die am häufigsten mit Scrum in Verbindung gebrachten Testtechniken sind der Test mit User Stories (siehe Kapitel 2.2.4) und sitzungsbasiertes Testen. Beim sitzungsbasierten Testen wird vorrangig explorativ getestet und es empfiehlt sich nur, wenn die Tester bereits umfassende Erfahrung im Testbereich aufweisen können und keine Testvorbereitung stattgefunden hat. Die Testfälle werden aus der Dokumentation der Testdurchführung erstellt und für spätere Testdurchläufe herangezogen. Für weitere Informationen zu explorativem Testen siehe [3].

Unabhängig davon mit welcher Methode getestet wird, gilt allgemein, dass gefundene Fehler im Product Backlog vermerkt und der entsprechenden User Story zugeordnet werden. Der Fehler wird außerdem umgehend dem verantwortlichen Entwickler mitgeteilt. Diese User Story gilt dann solange nicht als abgeschlossen, bis der Fehler behoben ist. Oft ist eine Behebung im aktuellen Sprint nicht möglich oder gewünscht, beispielsweise wenn der Fehler eine geringe Priorität aufweist. Dann wird eine „Bug Story“, also eine User Story, die den Fehler aufzeigt, dem Product Backlog hinzugefügt. Diese speziellen User Stories und gefundene Fehler sind in weiterer Folge wichtig für die Retests in jedem Sprint, um eine hohe Qualität zu gewährleisten. Weiters gilt es festzuhalten, welche Tests für einen Regressionstest in jedem Sprint erneut durchgeführt werden sollen. Es empfiehlt sich, User oder Bug Stories entsprechend zu kennzeichnen.

Stabile Komponenten der zu entwickelnden Software, das heißt Komponenten, die sich nicht mehr oder nur geringfügig verändern, eignen sich darüber hinaus zur Testautomatisierung [54]. Automatisierte Tests helfen, den von Sprint zu Sprint steigenden Testaufwand zu minimieren oder zumindest konstant zu halten. Klarer Weise muss bei all diesen Testdurchläufen sichergestellt werden, dass Features aus vorhergehenden Iterationen weiterhin einwandfrei funktionieren. Ausführlich wird dieses Thema in Kapitel 4.1 behandelt.

2.2.6 XP - eXtreme Programming

eXtreme Programming (XP) ist eine Entwicklungsmethodik die 1999 erstmals in Erscheinung getreten ist. Beck gilt als Vater von XP und hat 2004 die bereits 2. Auflage von seinem XP-Werk herausgebracht [7]. Seine Bücher sind weltweit erfolgreich und wurden auch ins Deutsche übersetzt. XP ist eine Ansammlung von Praktiken und Leitfäden, die das Ziel haben, den Entwicklungsprozess zu beschleunigen, die Qualität der zu entwi-

ckelnden Software zu erhöhen und gleichzeitig diverse Entwicklungsrisiken zu minimieren.

2.2.6.1 Die vier essentiellen XP-Werte

Kent Beck [7] nennt die vier Werte, auf denen XP basiert. Ohne diesen vier grundlegenden Werten ist es nicht möglich, XP erfolgreich einzusetzen.

Kommunikation
Die meisten Probleme in Projekten haben ihren Ursprung in einem Kommunikationsfehler. Dies können Kommunikationsprobleme zwischen Entwicklern, zwischen Entwicklern und dem Kunden oder zwischen Management und den Entwicklern sein. Durch diese Kommunikationsfehler werden in Folge oft falsche Entscheidungen getroffen. XP setzt auf unterschiedliche Verfahren, um die Kommunikation aufrecht zu erhalten und Missverständnisse auszuräumen. Falls es Probleme gibt, so ist ein Coach vorgesehen, um die Kommunikation wieder in Gang zu bringen.

Einfachheit
Einfache Lösungen zu erreichen ist nicht immer einfach. Das Team beschäftigt sich oftmals nicht mit den unmittelbar relevanten Anforderungen, sondern findet komplexe Lösungsansätze zu in weiter Zukunft liegenden Problemen. Doch durch Änderungen in den Anforderungen müssen sehr oft ohnedies ganz andere und neue Lösungsansätze gewählt werden und somit muss das vorher vermeintlich kluge, langfristige Konzept ohnehin verworfen werden. In Zusammenhang mit dem Faktor Kommunikation bedeutet das: Je einfacher ein System oder eine Lösung ist, desto weniger muss kommuniziert werden, da die Lösung quasi selbsterklärend ist.

Feedback
Konkretes Feedback über den aktuellen Status des Systems ist absolut unabdingbar. In erster Linie liefern Komponententests Feedback zu dem zu entwickelnden System: Arbeitet das System so, wie es die Anforderungen beschreiben, gibt es einige wenige Fehler oder funktionieren nur wenige Teile im System korrekt. Auch nicht funktionale Tests, wie zum Beispiel Performance Tests, liefern Feedback über ein System. Werden neue Anforderungen (z.b. via Storycard) eingebracht, so kann der Programmierer Feedback über die Qualität der Storycard liefern und eine Aufwandschätzung abgeben. Generell gilt: Je mehr Feedback man über ein System hat, desto besser kann man darüber kommunizieren.

Mut
Vielen Entwicklern fehlt der Mut, bereits geleistete Arbeit wieder zu verwerfen und neu zu beginnen. Es fällt ihnen nicht leicht, ihren tagelang gebauten Code einfach zu löschen und durch einfacheren, schlanken Code zu ersetzen. Viele haben dann das Gefühl, Zeit verloren zu haben oder nichts Produktives gearbeitet zu haben und dies womöglich gegenüber ihren Vorgesetzten rechtfertigen zu müssen. Hat man beispielsweise drei Möglichkeiten identifiziert, etwas umzusetzen, gehört Mut dazu, jede der Möglichkeiten einen Tag lang zu entwickeln und erst am dritten Tag zu entscheiden, welche der Möglichkeiten für die Lösung herangezogen wird.

2.2.6.2 Die fünf essentiellen XP-Grundprinzipien

Aus den genannten vier Werten leitet Beck [7] 15 Prinzipien ab und identifiziert folgende fünf als die essentiellen Grundprinzipien:

Unmittelbares Feedback
Je kürzer der Zeitraum zwischen einer Aktion und dem Feedback, desto höher der Lernerfolg.

Einfachheit anstreben
Je einfacher eine Lösung ist, umso schneller lässt sich diese umsetzen und von anderen verstehen. Falls notwendig, kann diese dann auch schnell und leicht geändert werden.

Inkrementelle Veränderung
Softwaresysteme wachsen mit der Zeit. Auch wenn wir Einfachheit anstreben und diese auch umsetzen, können Änderungen unerwartete Seiteneffekte hervorrufen. Wenn wir an unserem System immer nur überschaubare Änderungen vornehmen und das System ausreichend testen, so können diese Seiteneffekte entweder leicht ausfindig gemacht werden oder sogar ausgeschlossen werden. Verändern wir gleichzeitig sehr viel an einem Softwaresystem, können mehrere Seiteneffekte auftreten, auch solche, die sich dann gegenseitig beeinflussen. Eine Fehlerkorrektur wird dadurch wesentlich aufwändiger.

Veränderung wollen
Fortschritt in einem lebenden Softwareprojekt bedeutet auch, dass sich das System verändern muss. Durch diese Veränderungen müssen oft einzelne Dinge neu gebaut werden oder andere Teile komplett gelöscht werden. Vor Fehlern sollen Entwickler keine Angst haben, denn gemachte Fehler bergen das größte Lernpotential. Durch Testen und Qualitätssicherung muss ausgeschlossen werden, dass diese Fehler erst im in Produktion befindlichen System entdeckt werden.

Qualitätsarbeit
Qualität darf in keinem Projekt zur Diskussion stehen, denn kein Entwickler möchte schlampig arbeiten oder schlechte Qualität liefern. Aufwände dürfen in Projekten nicht eingespart werden, um an der Qualität zu sparen. Akzeptable Werte für Qualität sollten ausschließlich „hervorragend“ oder „sehr hervorragend“ sein.

Der Vollständigkeit halber, werden hier die weiteren zehn XP-Prinzipien aufgelistet, ohne näher darauf einzugehen[10]:

- Lernen lehren
- Geringe Anfangsinvestition
- Auf Sieg spielen

[10] Kurze Erläuterungen zu den zehn XP-Prinzipien sind im Werk von Balzert u. a. [5] zu finden.

- Gezielte Experimente
- Offene, aufrichtige Kommunikation
- Die Instinkte des Teams nutzen, nicht dageben arbeiten
- Verantwortung übernehmen
- An örtliche Gegebenheiten anpassen
- Mit leichtem Gepäck reisen
- Ehrliches Messen

2.2.6.3 XP-Praktiken

Wie eingangs schon erwähnt, handelt es sich bei XP nicht um ein Vorgehensmodell oder ein Prozessmodell sondern um Praktiken. Die XP-Praktiken leiten sich aus den vorher genannten Prinzipien ab, da sie dabei unterstützen, diese Prinzipien einzuhalten.

Die Praktiken behandeln ganz unterschiedliche Aspekte der Entwicklung und stehen daher oft nicht in direktem Zusammenhang miteinander. Es können individuell XP-Praktiken miteinander kombiniert werden, um das Optimum für das jeweilige Team zu finden. Für Hruschka, Rupp und Starke sind folgende Praktiken die wesentlichen, wie in [43] beschrieben ist:

Iteratives planen Es wird ständig geplant. Kosten und Risiken werden von den Entwicklern geschätzt.

Akzeptanztests Zusätzlich zu den Funktionalitäten beschreibt der Kunde auch, wie automatische Akzeptanztests auszusehen haben.

Einfaches Design Das System wird so einfach wie möglich gehalten. Man beschäftigt sich möglichst mit den unmittelbaren Anforderungen.

Paarprogrammierung (Pair Programming) Durch das paarweise Programmieren ist ein kontinuierliches Review gewährleistet.

Testgetriebene Entwicklung Die Testfälle werden immer vor dem Programmcode geschrieben. Details zur testgetriebenen Entwicklung finden sich in Abschnitt 2.2.9.

Refactoring Durch regelmäßiges „Aufräumen“ wird der Code sauber gehalten und kontinuierlich verbessert.

Kontinuierliche Integration Mindestens einmal täglich soll der geschriebene Programmcode aller Entwickler kompiliert werden. Nur Code, der funktioniert und durch Unit Tests verifiziert wird, kann übernommen werden.

Kollektives Eigentum Der Programmcode darf immer und überall geändert, verbessert und erweitert werden.

Programmierstandard Das gesamte Team muss sich an den zuvor vereinbarten Programmierstil und die Code-Konventionen halten.

Metapher Das Team hat ein gemeinsames Verständnis davon, wie das System funktionieren wird.

Überstunden Wenn Überstunden die Ausnahme sind, kann die Arbeitsleistung des Teams über einen beliebig langen Zeitraum gehalten werden.

Eine weitere XP-Praktik, die Hruschka, Rupp und Starke gänzlich ausgelassen haben, die nach Meinung des Autors jedoch unbedingt zu beachten ist [7]:

Kurze Releasezyklen Dadurch bekommt der Kunde schneller einen Einblick in das Produkt und kann früher Änderungen bekanntgeben.

2.2.6.4 Test in XP

Wolf, Roock und Lippert beschreiben in [88] folgende zwei Typen von Tests in XP: Komponententests und Akzeptanztests. Komponententests sollen so früh wie möglich erstellt werden. Am besten werden die Tests noch vor dem eigentlichen Code, der durch den Test geprüft wird, geschrieben. Diese Tests können dann jederzeit automatisiert ausgeführt werden und zum Beispiel in einen Deploymentprozess eingebunden werden. Somit benötigt die Durchführung der Testläufe keine wertvollen menschlichen Ressourcen. Werden zu einem späteren Zeitpunkt Fehler gefunden, die nicht durch die Komponententests abgedeckt sind – unter Umständen kann auch der Komponententest fehlerhaft sein –, so wird dieser Test zuerst korrigiert und danach erst der Fehler im Quellcode beseitigt.

Beck schildert in [7], dass Tests generell wichtig sind. Vor allem die Komponententests, auf welche dort eingegangen wird, stellen sicher, dass die Software besser entwickelt wird. Genauere Vorgaben, wie die Akzeptanztests auszusehen haben, werden jedoch nicht beschrieben. Wolf, Roock und Lippert bieten in [88] zwei mögliche Herangehensweisen für die Akzeptanztests. Einerseits ist es möglich, die Akzeptanztests direkt an der Ebene der Fachlogik durchzuführen. Dies ist jedoch mit dem Risiko verbunden, dass Fehler, die durch die Benutzeroberfläche maskiert oder hervorgerufen werden, mit diesen Tests nicht aufgezeigt werden. Andererseits können die Akzeptanztests an der Oberfläche durchgeführt werden, was wiederum den Nachteil hat, dass während der Entwicklung die Benutzeroberfläche sehr vielen Änderungen unterliegt. Dies erschwert die Automatisierung dieser Tests ungemein.

Sinnvoll erscheint eine Kombination aus den beiden Ansätzen. Hierbei ist das Testmanagement gefordert, um unnötige Redundanzen bei den Tests zu vermeiden.

2.2.7 Die Crystal Methodenfamilie

Bunse und Knethen [12] beschreibt Crystal als eine Familie agiler Methoden, die auf einem inkrementellen Vorgehen aufbaut. Es gibt keine konkreten Vorgaben an Werkzeugen oder Vorgehensweisen, vielmehr sind Regeln definiert, die zu beachten sind. Alistair Cockburn ist in [15] der Auffassung, dass jeder Mensch anders arbeitet und es aus diesem

Grund keine Methoden gibt, die für alle gleich gut funktionieren. Die Grundphilosophie von Crystal ist, dass die Softwareentwicklung ein Zusammenspiel von Erfindungsgeist und Kommunikation ist, mit dem primären Ziel, benutzbare Software auszuliefern, und dem sekundären Ziel, Erfahrungen für das nächste Softwareprojekt zu sammeln.

Weiters beschreibt Jens Coldewey in seinem Artikel [18] die grundlegende Idee von Alistair Cockburn, dass die Gegebenheiten in dem jeweiligen Projekt beachtet werden müssen. Die Crystal Methodenfamilie orientiert sich in erster Linie an der Größe und der Kritikalität des Projektes. Je höher die Anzahl der Mitarbeiter, desto umfangreicher muss der Projektrahmen sein. Ebenso gilt: je höher die Kritikalität, desto formaler muss dieser Rahmen sein. In Abbildung 2.7 sind die Mitarbeiter im Projekt auf der x-Achse verzeichnet und die Kritikalität auf der y-Achse. Die Z-Achse gibt schließlich die Priorität an, in der das Projekt abgewickelt werden soll (Kosten, Geschwindigkeit, etc.).

In der Crystal Methodenfamilie kennzeichnet ein Farbton die verschiedenen Ausprägungen der Methoden und Prozesse. Je dunkler die Farbe, desto spezifischer sind die Ausführungen und desto mehr Methoden und Prozesse sind durch Crystal vorgegeben. Die Crystal Methodenfamilie umfasst:

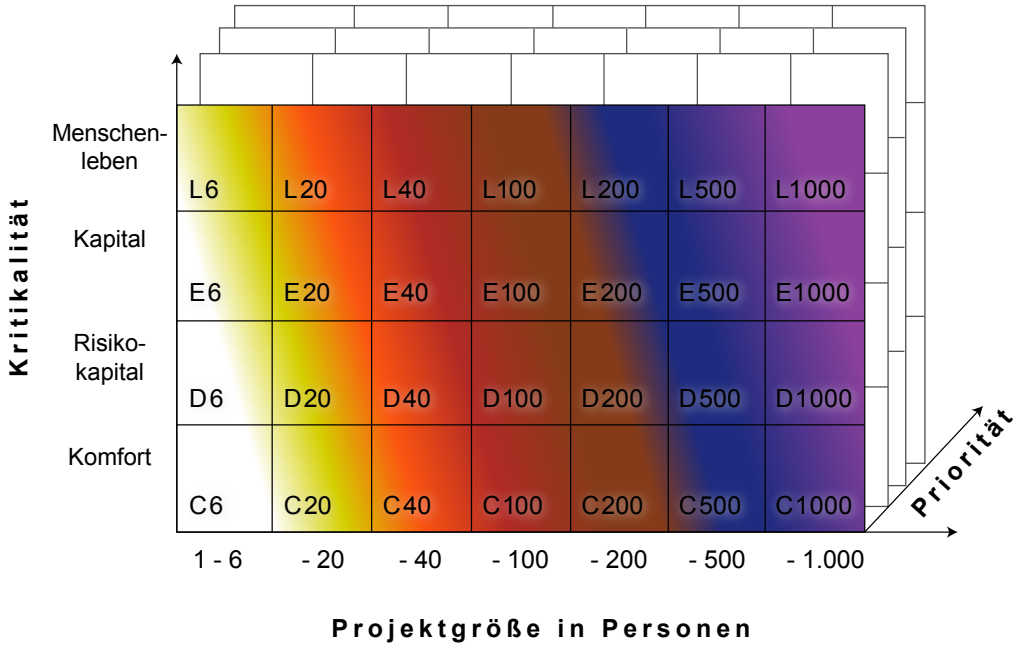

Abbildung 2.7: Crystal Projektkategorien

- Crystal Clear
- Crystal Yellow
- Crystal Orange
- Crystal Orange Web

- Crystal Red
- Crystal Maroon
- Crystal Diamond (Blue)
- Crystal Sapphire (Violet)

Die verschiedenen Methoden befassen sich vorrangig damit, wie die Projektmitglieder miteinander kommunizieren und wie man diese koordiniert. Crystal ist für Cockburn ausschließlich für Teams einsetzbar, die an einem gemeinsamen Ort arbeiten [15].

Die Prinzipien, auf denen alle Crystal-Methoden basieren, sind für die unterschiedlichen Ausprägungen ident. Coldewey hat diese zusammengefasst [18] und macht klar, dass in jedem Fall die persönliche und direkte Kommunikation zwischen zwei Menschen zu präferieren ist. Denn der Dialog ist sowohl effizient als auch effektiv. Weiters hält Coldewey fest, dass mit der Größe des Teams auch die Anzahl der Regeln steigt. Sie sind essentiell, um größere Teams zu managen, zu koordinieren und ein sinnvolles Maß an Einheitlichkeit beizubehalten. Gleichzeitig verdeutlicht Coldewey, dass unnötige Regeln keinem helfen, sondern nur viel Geld kosten. Somit sollte für jede Teamgröße ein passendes Ausmaß an Reglementierung gefunden werden. Die Notwendigkeit an klaren und nach außen transparenten Regelungen, Interaktionen und Entscheidungen hängt aber auch direkt mit der Kritikalität eines Projektes zusammen. Es gilt: je kritischer, desto nachvollziehbarer für Außenstehende.

Bei all diesem Regelwerk unterstreicht Coldewey jedoch, dass mit Regeln alleine noch nichts erreicht ist: *„Formalismus bedeutet nicht Disziplin, Prozess bedeutet nicht Fähigkeit, Dokumentation bedeutet nicht Verständnis.“* [18] Somit ist hier wieder Kommunikation der Schlüsselfaktor, um Disziplin im Projekt zu erreichen, Fähigkeiten zu fördern und Verständnis zu erreichen. Coldewey verdeutlicht, wie wichtig Kommunikation im Allgemeinen und Feedback im Speziellen sind, denn sie reduzieren die Anzahl an Zwischenprodukten oder machen sie gar überflüssig. Das laut Coldewey letzte Prinzip, welches allen Crystal-Methoden zu Grunde liegt, besagt, dass Effizienz in einem Projekt nicht immer gleich wichtig ist. Befindet man sich in einem Projekt am kritischen Pfad, ist Effizienz das oberste Gebot. Arbeitet man jedoch gerade in einer weniger kritischen Phase, kann man mehr Wert auf Qualität und Stabilität legen oder gar Kollegen unterstützen, die sich selbst gerade auf dem kritischen Pfad befinden. Im Grunde verdeutlichen diese Prinzipien die Wichtigkeit der Kommunikation, um Kollaboration in einem Crystal Projekt überhaupt möglich zu machen.

2.2.7.1 Crystal Clear

Crystal Clear ist die kleinste Methodensammlung der Crystal-Familie und wird deshalb hier zur Illustration etwas genauer betrachtet. Alistar Cockburn definiert vier Rollen: Auftraggeber, Senior Softwareentwickler, Softwareentwickler und Benutzer. Die Aufgaben des Projektkoordinators kann eine der vorher genannten Rollen zusätzlich übernehmen. Die Aufgaben des Anforderungsanalysten werden im Team aufgeteilt. Eine Person im Team übernimmt außerdem die Rolle des Fachbereichsexperten. Die Schlüsselrolle in dem Team hat aber der Seniorentwickler.

Vorgesehen sind Entwicklungsiterationen, die zwei bis drei Monate andauern und in denen Meilensteine definiert sind. Der Benutzer ist direkt in die Entwicklung eingebunden und Releases werden immer im Vieraugenprinzip begutachtet. Zumindest Teile der Regressionstests werden automatisiert. In jeder Iteration werden zu Beginn und in der Mitte Workshops zum Methodenreview und für Produktverbesserungen abgehalten.

Es liegt dabei in der Teamverantwortung, ob Standards für Programmierung (Coding) und die Benutzeroberfläche sowie Details und Standards für die Regressionstests festgelegt werden. Weiters setzt Alistar Cockburn voraus, dass ein Versions- und Konfigurationstool sowie ein digitales Whiteboard eingesetzt werden. In schnelllebigen Projekten werden wichtige Informationen oder Gedanken auf Whiteboards festgehalten, jedoch gehen diese verloren, da diese selten in Dokumente überführt werden. Der Nutzen, solche Informationen und Ideen speichern zu können, ist jedoch um ein Vielfaches größer als die Kosten für ein digitales Whiteboard. Ein solches Werkzeug sollte generell in agilen Projekten zur Regel werden.

Es ist ausdrücklich erlaubt, Methoden von anderen (agilen) Modellen, wie beispielsweise ein tägliches Scrum Meeting, Pair Programming aus XP oder Ähnliches, in Crystal Clear einzusetzen.

Helmut Balzert [5] kritisiert an Crystal Clear, dass die extreme Mitarbeiterorientierung zu Problemen führen kann, wenn die sozialen Kompetenzen der Mitarbeiter, wie Teamfähigkeit und Kommunikationsfähigkeit, nicht so ausgeprägt sind, wie es für Crystal notwendig wäre.

Da Crystal, wie XP, kein Vorgehensmodell ist, sondern eine Methodensammlung, ist es leicht möglich, von XP zu Crystal zu wechseln, wie Cockburn in [15] schildert.

2.2.7.2 Test in Crystal

Cockburn definiert keinen genauen Testprozess für Crystal. Er hebt aber hervor, dass Entwickler niemals ihre eigenen Codezeilen testen und überprüfen sollen. Stattdessen schlägt er eine ständige Rotation vor oder – bei größeren Testaufgaben – sogar externe Teams. Grundlage für die Tests in Crystal sind Unit-Test-Frameworks, um die Entwickler in die Testaufgaben einzubinden. Wenn das Projekt im Entwicklungsstadium bereits fortgeschritten ist und Testfälle über die Benutzeroberfläche durchgeführt werden müssen, nimmt Cockburn speziell die GUI-Programmierer in die Pflicht. In erster Linie sollte versucht werden, die Akzeptanztests, die über die Benutzeroberfläche arbeiten, zu automatisieren, jedoch erwähnt er auch die damit verbundenen Schwierigkeiten mit der Automatisierung. Weiters legt er fest, dass die Akzeptanztests so gestaltet und beschrieben werden müssen, dass diese sowohl von Entwicklern und Testern als auch vom Fachbereich durchgeführt werden können.

2.2.8 Feature Driven Development

Feature Driven Development (deutsch: Funktionsgetriebene Entwicklung) (FDD) wurde 1997 von Coad, Lefebvre und De Luca ins Leben gerufen. Er erwähnt FDD erstmalig

1999 in dem Werk [14]. Die Entwicklung in diesem Modell basiert auf einem Feature-Plan.

FDD besteht aus fünf aufeinanderfolgenden Teilprozessen, wobei nur die letzten beiden iterativ durchlaufen werden. Da die ersten drei Phasen klassisch organisiert sind und zuerst vollständig durchlaufen werden müssen, sieht Bunse [12] FDD als semi-agil.

Die Phasen des FDD sind:

1. Entwicklung des globalen Modells
2. Erstellung der Featureliste
3. Planung für jedes Feature
4. Entwurf für jedes Feature
5. Entwicklung für jedes Feature

In den ersten beiden Phasen wird das System global definiert und meist mittels UML Diagramm eine Abstraktion des zu programmierenden Systems erstellt, wie Wolf in [87] beschreibt. Danach wird das System in Features unterteilt. In der dritten Phase werden dann die Features priorisiert und eine Reihenfolge für die Realisierung festgelegt, dies geschieht im Konsens mit dem Projektleiter und dem Entwicklungsleiter. Dabei müssen natürlich sämtliche Abhängigkeiten berücksichtigt werden. Daraus lassen sich dann die Fertigstellungstermine für die einzelnen Geschäftsprozesse ableiten, die aus einer Gruppe von Features bestehen.

Im FDD Modell finden sich viele agile Elemente, auch wenn die ersten drei Phasen einem herkömmlichen Modell sehr ähnlich sind. Agile Methoden wie Kommunikation, Einfachheit, laufende Software, etc. werden jedoch vorausgesetzt, um FDD einzusetzen.

2.2.8.1 Test im Feature Driven Development

Im Feature Driven Development wird der Softwaretest nicht explizit als eigenständige Disziplin erwähnt (siehe dazu [14]). Coad, Lefebvre und De Luca haben jedoch folgende Testtätigkeit festgehalten: Bei der Entwicklung eines Features ist eine Programmcodeinspektion durch das Entwicklungsteam vorgesehen. Es wird eine formale Codeinspektion mit dem Team durchgeführt. Falls der Entwicklungsleiter es für notwendig erachtet, kann hier das Team auch einen externen Experten hinzuziehen.

Verpflichtend vorgesehen sind auch Komponententests, auch diese werden mittels Codeinspektion geprüft.

2.2.9 Test Driven Development

Im Test Driven Development (deutsch: Testgetriebene Entwicklung) (TDD), werden Testfälle nicht nach dem Programmcode geschrieben, sondern bereits bevor der zu testende Programmcode erstellt wird. Das heißt, es wird zuerst der Testfall erstellt und danach wird solange programmiert, bis der Testfall nicht mehr fehlschlägt.

Beck gliedert in [8] den Prozess in folgende fünf Schritte:

1. Erstelle Tests.
2. Starte die Tests und lass diese fehlschlagen.
3. Mach kleine Änderungen am Programmcode.
4. Starte die Tests erneut, bis diese erfolgreich sind.
5. Überarbeite (= refactoring), um doppelten Code zu entfernen.

Durch diesen Entwicklungsprozess wird viel Programmcode erzeugt, der auf den ersten Blick keineswegs ein Produkt liefert, sondern lediglich eine Lösung für ein ganz spezielles Problem. Wenn beispielsweise ein Wert multipliziert werden soll, so wird dieser Wert im Programmcode einfach multipliziert und wieder ausgegeben. Wird der Wert geändert, so stimmt in Folge das Ergebnis nicht mehr. Diese Vorgehensweise wird auch als „hardcodiert“ bezeichnet. Erst der letzte Schritt im TDD macht aus dem hardcodierten Code einen parametrisierten, der für alle Eingabewerte funktioniert.

Somit wird bei TDD viel Programmcode geschrieben und wieder verworfen, bis der eigentliche Programmcode entsteht, der schlussendlich die Lösung darstellt.

2.2.10 Behaviour Driven Development

Wynne und Aslak beschreiben Behaviour Driven Development (deutsch: Verhaltensgetriebene Softwareentwicklung) (BDD) als eine Erweiterung von Testgetriebener Entwicklung [90], denn auch hier wird im Vorhinein festgelegt, welche Funktionalität erwünscht ist. Bei BDD wird jedoch gezielt eine Lösung für potentielle Missverständnisse zwischen Projektmitgliedern geboten. Denn BDD bedient sich allgemein verständlicher Ausdrücke und einer geläufigen Alltagssprache, sodass jede an dem Projekt beteiligte Person versteht, wie einzelne Funktionalitäten umgesetzt werden sollen. Durch diese Einfachheit werden Missinterpretationen so weit wie möglich ausgeschlossen.

Die ursprüngliche Idee, die North in [59] mit BDD verfolgt hat, war, dass die Namen der Testmethoden für die Komponententests als ganze Sätze formuliert werden. Was davor unter Umständen „Test_Login“ geheißen hat, wird nun allgemein verständlich wie folgt formuliert: „Login_mit_gültigen_Benutzernamen“. Dadurch wird auf einfache Weise eine Ausgabe erzeugt, die den Testfall genau beschreibt. Diese Ausgabe wird dabei von Analysten, Entwicklern und Testern gleichermaßen verstanden. North hat das erste Framework für die Umsetzung von BDD erstellt: JBehave.[11] Es existieren aber noch zahlreiche andere Umsetzungen in verschiedensten Programmiersprachen: RBehave, NBehave, Cucumber, radish, Specfolw, Behat, SubSpec, und weitere.

Wie auch in vielen anderen agilen Vorgehensmodellen werden im BDD die Anforderungen in User Stories formuliert. Für das BDD werden nun aus diesen User Stories verschiedene Szenarios abgeleitet, welche wesentlich detaillierter sind. Während User Sto-

[11] http://jbehave.org/

ries eher ein Ziel beschreiben, wird in Szenarios im Detail beschrieben, wie dieses Ziel erreicht werden soll.

Eine User Story hat – wie in Kapitel 2.2.4 beschrieben – folgende Form:

Als [Rolle] möchte ich [Ziel], damit ich [Begründung].

Um Szenarien zu erhalten, werden die User Stories umformuliert. Im folgenden Beispiel wird dafür die Beschreibungssprache Gherkin [30] verwendet.[12] Der Aufbau eines Szenarios sieht wie folgt aus:

Given Im ersten Schritt werden Vorbedingungen und Ausgangssituation festgelegt.

When Der zweite Schritt beschreibt ein Ereignis, das eintrifft oder ausgeführt werden soll.

Then Zum Schluss wird das zu prüfende Ergebnis beschrieben.

Die Schlüsselwörter (**Given, When, Then**) sind in vielen Sprachen verfügbar. In Deutsch lauten diese beispielsweise (**Gegeben, Wenn, Dann**). Für komplexere Szenarios wird auch noch das Schlüsselwort **And (Und)** verwendet, um Szenarios vollständig beschreiben zu können.

Um die Ableitung zu illustrieren, sei folgendes Beispiel genannt. Die User Story lautet hier:

Als [Administrator eines Nationalverbandes] möchte ich [Mitgliederdaten editieren], um [Änderungen einzupflegen].

Ein daraus abgeleitetes Szenario wird nun lauten:

Gegeben ich bin als Administrator eines Nationalverbandes eingeloggt.

Wenn ich die Daten eines Mitglieds editiere,

Dann werden die Daten gespeichert und korrekt angezeigt.

Damit ist jedoch noch nicht unbedingt die User Story komplett abgebildet: In vielen Fällen sind mehrere Szenarios notwendig, um eine User Story vollständig zu beschreiben.

Für zahlreiche Programmiersprachen gibt es unterschiedliche Implementierungen des BDD, wie beispielsweise Cucumber, JBehave, NBehave, Behat, und andere. Jedes Team muss die bevorzugte Implementierung und Umgebung selbst wählen. Dabei spielen Parameter wie die eingesetzte Programmiersprache im Projekt oder persönliches Know-how eine

[12] Gherkin kommt ebenfalls im Praxisteil in Kapitel 4.4.1 dieser Arbeit zur Anwendung.

große Rolle.[13] Um die Umsetzung von BDD im Projekt zu erleichtern, gibt es außerdem für die bekannten Entwicklungsumgebungen IntelliJ IDEA[14] und Eclipse[15] Plug-ins.

2.2.11 Zusammenfassung

In allen Softwareentwicklungsmodellen ist der wichtigste Faktor: Disziplin zu kommunizieren. Denn in einem agilen Umfeld ist es essentiell für den Ablauf, dass alle Personen, wie Auftraggeber, Entwickler, Tester, etc., miteinander kommunizieren. Das Loslösen von starren Vorgehensmodellen hin zu agilen, kreativen Methoden erfordert daher umso mehr Disziplin von allen Teammitgliedern, um diese Kommunikation stets aufrechtzuerhalten.

Trotz allem gilt in agilen ebenso wie in traditionellen Modellen, dass Entwickler sich an vereinbarte Programmierrichtlinien halten müssen und sie selbstverständlich ihr Bestmögliches tun, um die Qualität im Projekt aufrechtzuerhalten beziehungsweise zu verbessern.

Dafür werden in agilen Methoden Redundanzen, Managementaufwände und aufgeblasene Workflows weitgehend reduziert oder gar vermieden.

Gernert unterstreicht außerdem in [28], dass der Teammanager möglicher Weise die wichtigste Rolle bei agilen Projekten ist. Denn um ein agiles Team gut zu führen, muss der Teammanager unnötigen Ballast vom Team fern halten, das Team in den Mittelpunkt des Handelns bringen und natürlich das Team stets motivieren. Diverse agile Modelle sind sich dabei einig, dass die Motivation durch eine klare Ausrichtung auf das gemeinsame Ziel und durch die Abschaffung von komplizierten Kommunikationswegen gesteigert werden kann. Die Basis für diese agile Projektkultur bilden dabei eigenverantwortliches, selbstständiges Handeln, ehrliches, rasches Feedback sowie grundlegendes Vertrauen der Teammitglieder untereinander.

2.3 Near Field Communication

NXP Semiconductors und Sony haben 2006 Near Field Communication (deutsch: Nahfeldkommunikation) (NFC) ins Leben gerufen. Das NFC-Protokoll soll es erleichtern, Daten zwischen zwei Geräten auszutauschen. Da diese Technologie nur über sehr kurze Distanz funktioniert, kann mit relativ einfachen Sicherheitsmechanismen gearbeitet werden (vgl. dazu [66]). Die aktuelle Spezifikation von NFC ist auch unter [58] zu finden.

NFC basiert auf der Radio Frequency Identification (deutsch: Identifizierung mittels elektromagnetischer Wellen) (RFID) Technologie und hat einen gesonderten Frequenzbereich der nach ISO 18000-2, ISO 1800-3 und ISO 22536 genormt ist. Somit ist NFC ein internationaler Übertragungsstandard für den Datenaustausch über Strecken bis zu 10 cm. Die NFC-Technologie ermöglicht es, dass die Karte oder ein anderes NFC- Medium[16] mit

[13] Die Umsetzung mit Cucumber ist detaillierter in Kapitel 4.4.1 zu finden.
[14] http://www.jetbrains.com/idea/
[15] http://www.eclipse.org
[16] Es gibt Karten, Sticker, Schlüsselanhänger und mittlerweile sogar Kugelschreiber, die NFC-Tags eingebaut haben.

dem Kartenleser kommuniziert, ohne dass zwischen diesen beiden ein direkter Kontakt besteht. Dies ist möglich, da NFC-Karten eine Antenne eingebaut haben, die mittels Induktion vom Kartenleser aufgeladen wird und so genügend Strom erzeugt, dass die Karte Programmcode ausführen kann, wie in Abbildung 2.8 dargestellt. Rankl und Effing unterscheiden folgende vier Kartenarten nach Art des verbauten Chips:

- Speicherchip
  - ohne Sicherheitslogik
  - mit Sicherheitslogik
- Prozessorchip
  - ohne Koprozessor
  - mit Koprozessor

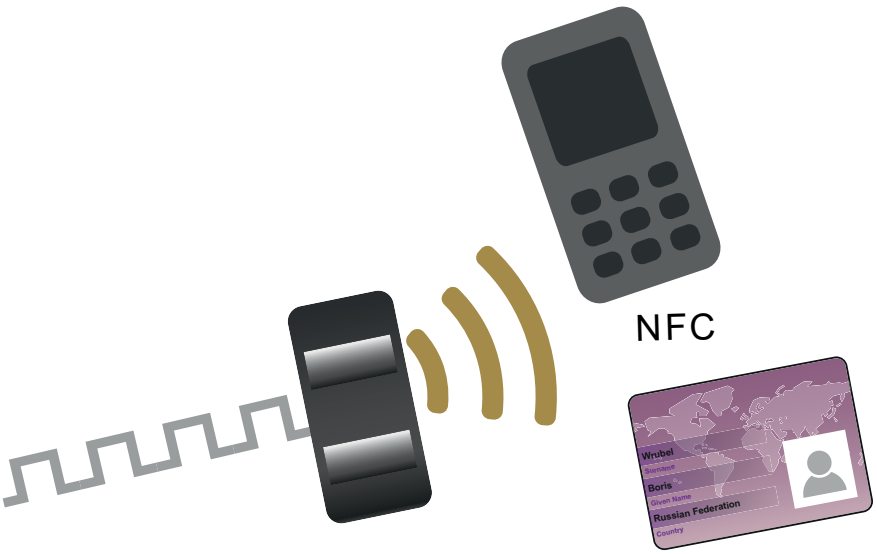

Abbildung 2.8: Schematische Darstellung eines NFC-Lesevorganges

Rankl und Effing beschreiben, dass Speicherkarten eine eingeschränkte Funktionalität haben, da Daten ausschließlich geschrieben und gelesen werden können. Diese Karten werden hauptsächlich dort eingesetzt, wo man einen Wert im Voraus bezahlen muss und dieser dann bei Konsumation geschmälert wird, wie es beispielsweise bei Telefonwertkarten der Fall ist. Um es Angreifern dieser Technik nicht zu ermöglichen, den Wert der Karte zu speichern und zu einem späteren Zeitpunkt wieder auf die Karte zurückzuschreiben, wurde eine Sicherheitslogik in die Speicherkarten eingebaut: Der Chip auf der Karte verhindert, dass eine einmal beschriebene Speicherzelle wieder gelöscht werden kann. Somit ist die Reduktion des Wertes auf der Karte nicht mehr rückgängig zu machen.

Prozessorkarten haben neben dem Prozessor, dem optionalen Koprozessor und einem flüchtigen Arbeitsspeicher noch einen rein lesbaren Speicherbereich (= ROM) und einen nicht flüchtigen Speicherbereich, der sogenannte Electrically Erasable Programmable Read-Only Memory (deutsch: elektrisch löschbarer programmierbarer Nur-Lese-Speicher) (EEPROM). Die Varianten, der am Markt erhältlichen Karten, sind sehr groß: Viele neuere

Karten haben statt dem EEPROM und dem ROM einen Flashspeicher verbaut. Somit kann die Karte flexibler benutzt werden, da es bei diesen Flash-Modellen nicht notwendig ist, dass der Hersteller vorab alle notwendigen Daten und Programme in den ROM einspeist.

In dem hier behandelten Projekt (siehe Kapitel 3) werden ausschließlich kontaktlose Prozessorkarten verwendet. Daher werden Karten mit Kontakten und Magnetstreifen hier nicht näher erläutert. Jedoch können ausführliche Informationen zu sämtlichen Karten und deren Funktionsweisen in [66] nachgelesen werden.

NFC-Karten können auch als vierte Generation von Ausweisen angesehen werden: Nach visuellen Ausweisen, Magnetkarten und Kontakt-Chipkarten sind NFC-Karten mittlerweile zum Standard für kontaktlose Karten geworden. Schmeh liefert dazu in [68] einen interessanten Überblick über amtliche NFC-Ausweisdokumente, die weltweit im Einsatz sind, und welche technischen Möglichkeiten diese nutzen.

Mittlerweile gibt es bereits eine breite Akzeptanz der NFC-Technologie und eine daraus resultierende Bandbreite kommerzieller Einsatzgebiete: beim Wintersport als Liftkarte, als Mitarbeiterausweis für die Zeiterfassung und Zutrittskontrolle, in der Identifikation von Tieren, bei Sportwettkämpfen zur Zeitnehmung, in Hotels als Zimmerschlüssel, in öffentlichen Verkehrsmitteln und seit Kurzem werden auch kontaktlose Zahlungssysteme dem Endkunden angeboten. Die Möglichkeiten der Anwendung und der Erfindungsreichtum der Produktenwickler sind derzeit scheinbar unbegrenzt.

Laut einer Studie von TNS Infratest [79] sind mehr als 25% der Befragten überzeugt, dass bis 2015 eine mobile Zahlmethode für sie persönlich relevant sein wird. Mastercard und Visa haben bereits Zahlungssysteme im Einsatz ([52] und [80]) die auf der NFC Technologie aufbauen.

Jedoch sind Befürchtungen der Endbenutzer, dass kontaktlose Verfahren nicht 100% sicher sind, noch nicht nachhaltig ausgeräumt. Durch den kontaktlosen Bezahlvorgang wird befürchtet, es könne durch einen manipulierten Türstock oder durch ein manipuliertes mobiles Lesegerät Geld abgebucht werden, ohne dass dies der Endbenutzer bemerkt. Ein passives Mithören einer RFID Kommunikation aus drei Metern zwischen Lesegerät und Karte konnten Finke und Kelter bereits in [24] beweisen. Daher ist es notwendig, Sicherheitsmaßnahmen zu implementieren, damit das passive Mithören von NFC-Datenverbindungen für den Angreifer keine nützlichen Informationen liefert, auch wenn dieser über längere Zeit diese Daten sammelt.

Eine Möglichkeit, die Sicherheit zu erhöhen, ist, mit aktiven NFC-Modulen zu arbeiten. Diese aktiven NFC-Module speichern temporär Energie oder werden direkt mit Strom versorgt. Das Mobiltelefon hat übrigens das größte Potential, das am weitesten verbreitete aktive NFC-Modul zu werden. Es wird regelmäßig mit Strom versorgt und einige Hersteller haben deswegen bereits den Trend erkannt und die neuen Modelle so ausgestattet, dass sie NFC verarbeiten und auf ein Secure Access Module (SAM) zugreifen können. Mit Hilfe des SAM ist es möglich, dass Lesegerät und Karte sich gegenseitig authentifizieren, um so Bezahlvorgänge von nicht autorisierten Gegenstellen auszuschließen. Da die meisten Mobiltelefone eine Internetverbindung herstellen können, ergeben sich darüber hinaus ganz neue Möglichkeiten für die Authentifizierung, da eine Prüfung des Bezahlvorgangs vorgenommen werden kann. Ein österreichisches Unternehmen [71] hat beispielsweise ein Bezahlsystem entwickelt, welches die Authentifizierung gänzlich via Internetverbindung durchführt.

Die Bedrohungen auf Kartensysteme sind jedoch ein nicht zu unterschätzender Faktor, welcher in der Entwicklung solcher Systeme immer berücksichtigt werden muss. Die möglichen Angriffspunkte auf Kartensysteme haben Rankl und Effing in [66] wie folgt unterteilt:

Angriff auf sozialer Ebene
Dies sind jene Angriffe, die sich hauptsächlich gegen die Menschen richten, die im Besitz der Chipkarte sind. Beispielsweise fallen das Ausspähen des PIN-Codes oder der Diebstahl der Karte unter einen Angriff auf sozialer Ebene.

Angriff auf physischer Ebene
Für diese Art von Angriff benötigt man die entsprechenden technischen Geräte sowie das Wissen, wie man den Mikrocontroller der Chipkarte attackieren kann, um an die gewünschten Daten zu kommen.

Angriff auf logischer Ebene
Viele bisher bekannt gewordene erfolgreiche Angriffe sind auf der logischen Ebene passiert. Die Angreifer setzen sich mit den verwendeten logischen Komponenten der Chipkarte auseinander und versuchen, durch Berechnungen oder erkannte Fehler im Chipkartenbetriebssystem, Zugriff auf die Daten der Chipkarten zu bekommen.

Unabhängig von der Art des Angriffes gilt es immer, die auf der Karte enthaltenen Daten zu schützen. In [67] werden diese, Schutzziele genannten, Vorgaben im Zusammenhang mit NFC-Karten wie folgt erläutert:

Datengeheimhaltung
Es muss sichergestellt sein, dass sowohl die Daten auf der Karte als auch die Daten während der Datenübertragung zwischen dem Lesegerät und der NFC-Karte von Unbefugten nicht ausgelesen werden können. In weiterer Folge muss die Datenübertragung vom Lesegerät zu dem jeweiligen System, das die Daten weiterverarbeitet, geschützt sein. Ziel ist es, dass die Daten auch nicht in verschlüsselter Form bei einem Unbefugten landen, denn dieser könnte viele andere verschlüsselte Daten sammeln und so versuchen, ein Muster zu erkennen.

Datenintegrität
In vielen Anwendungen ist die Datenintegrität das wichtigste Schutzziel. Für den Fall, dass zum Beispiel auf einer Karte keine persönlichen Daten gespeichert sind, sondern lediglich ein Geldbetrag, ist eine Geheimhaltung des Betrages nicht kritisch für den Konsumenten. Falls es jedoch die Möglichkeit gibt, dass dieser Betrag durch den Konsumenten selbst manipuliert wird, wäre das für den Betreiber des Systems fatal, da dieser dadurch zumindest einen finanziellen Schaden erleiden würde. Somit müssen die übertragenen Daten zwischen NFC-Karte und dem System von beiden Seiten authentifiziert werden, um sicherzustellen, dass diese Daten auf dem Übertragungsweg nicht manipuliert werden.

Schutz der Privatsphäre
Immer wenn personenrelevante Daten verarbeitet werden, muss der Schutz der Pri-

vatsphäre sichergestellt werden. Bei NFC werden die Daten kontaktlos, also sozusagen über den Luftweg, übertragen. Laut Spezifikation ist dies zwar nur über einen sehr geringen Abstand zwischen Kartenleser und NFC-Karte möglich, wie jedoch zuvor beschrieben, wäre hier aber zumindest ein Mithören möglich. Deshalb sollte diese Datenübertragung immer verschlüsselt stattfinden.

NFC-Karten haben ein weiteres Merkmal, das geschützt werden muss: eine eindeutige Identifikationsnummer (UID). Diese wird bei der Produktion der Karte vergeben und ist eindeutig. Damit keine Verhaltensmuster über Benutzer einer Karte erstellt werden können, darf auch diese einfache UID nicht unverschlüsselt abgefragt werden.

Um die drei genannten Schutzziele zu garantieren, betreiben vier Kreditkartenunternehmen (American Express, JCB, Mastercard und Visa) gemeinsam die EMVCo[17] und haben einen Standard für Kreditkarten mit Chip eingeführt. Dieser Standard, EMV [21] genannt, liegt aktuell in der Version 4.3 vor und definiert eine Spezifikation für Kreditkarten mit einem Prozessor. Weiters hat EMVCo auch einen Standard für kontaktloses Zahlen erstellt. Im Abschnitt Security and Key Management der umfangreichen EMV Spezifikation werden sämtliche Sicherheitsmechanismen sowie das Schlüsselmanagement beschrieben. EMV setzt auf asymmetrische Verschlüsselung[18], sämtliche Transaktionen werden darüber hinaus sowohl mit dem Schlüssel der Karte als auch mit jenem des Terminals verschlüsselt und erst danach an den Zahlungsprovider gesendet. Nur dem Zahlungsprovider bekannte und authentifizierte Terminals werden in weiterer Folge auch mit dem Händler abgerechnet. Damit ist sichergestellt, dass keine gefälschten Zahlungsterminals kontaktlos Abbuchungen vornehmen können.

In Österreich sind bereits Kreditkarten mit NFC im Umlauf, 2013 wird außerdem ein Großteil der Bankomartkarten mit NFC Funktionalität ausgerüstet sein [86].

Weitere ausführliche Informationen zu RFID Systemen, sowie über deren technische Funktionsweisen, sind zu finden in [25] von Finkenzeller. Das Werk von Henrici befasst sich ausführlich mit den Themen Sicherheit und Privatsphäre im Zusammenhang mit RFID [40]. Als weiterführende Literatur zum Thema Multi Application Smart Cards empfiehlt der Autor [39].

[17] EMV steht dabei für Europay, Mastercard und Visa, die drei Gründungsmitglieder der EMVCo.
[18] Siehe dazu [1] und [64]

3 Projektbeschreibung und Zielsetzung

Im Folgenden wird näher auf ein Projekt eingegangen, das im Bereich der Sport-Wettkampforganisation angesiedelt ist. Der Sport-Weltverband beauftragte ein Unternehmen mit der Entwicklung eines Mitgliederverwaltungssystems, sowie der dazugehörigen Infrastruktur und dem Vertrieb dieses neuen Systems.

Bisher war es bei Wettbewerben und Veranstaltungen mit sehr viel Aufwand verbunden, Mitglieder zu authentifizieren und festzustellen, ob diese berechtigt sind, an der jeweiligen Veranstaltung teilzunehmen. Das neue auf NFC-Cards optimierte Mitgliederverwaltungssystem soll diesen Prozess nun erheblich beschleunigen und den administrativen Aufwand deutlich reduzieren.

3.1 Projektbeschreibung

Das zu entwickelnde System zur Mitgliederverwaltung muss weltweit jederzeit verfügbar sein, denn der Sport-Weltverband hat rund um den Globus mehr als zehn Millionen Mitglieder. Untergliedert wird der Sport-Weltverband in Kontinentalverbände, welche wiederum aus den einzelnen Verbänden der Mitgliedstaaten bestehen, wie in Abbildung 3.1 zu sehen ist.

Die an alle Mitglieder neu ausgegebenen Mitgliedskarten werden mit NFC-Funktionalität versehen, um bei Veranstaltungen eine Authentifizierung vornehmen zu können. Die Gültigkeit der jeweils einjährigen Mitgliedschaft ist im Zentralsystem hinterlegt und kann nach Bezahlung der Mitgliedschaft immer wieder verlängert werden. Hierbei muss zwischen zwei Mitgliedschaften unterschieden werden: Einerseits kann eine Person eine Mitgliedschaft direkt mit dem Sport-Weltverband abschließen, andererseits kann eine Person einem nationalen Verband beitreten, welcher wiederum mit allen seinen Mitgliedern dem Sport-Weltverband beitritt. Vereinfacht gesagt gibt es eine „direkte“ Mitgliedschaft zwischen Person und dem Sport-Weltverband und eine „indirekte“ Mitgliedschaft über einen Zwischenverband. Die Verbände einzelner Staaten sind darüber hinaus meist auch noch in weitere Landesverbände unterteilt. Im Falle von Österreich sind das beispielsweise neun Landesverbände - einer für jedes Bundesland.

Durch diese komplexe Organisation ergeben sich unterschiedliche Datenstrukturen, welche in der Entwicklung zu berücksichtigen sind. Auch die Internationalität bringt einige implizite Vorgaben mit sich, wie die Verarbeitung von Zeichensätzen aus verschiedenen Alphabeten – von Kyrillisch über Griechisch bis Lateinisch.

Alle Neuanmeldungen werden im Zentralsystem vorgenommen und es erfolgt umgehend die Produktion der individuellen Mitgliedskarte für das neue Mitglied. Die neue Karte ist sowohl die Mitgliedskarte im Sport-Weltverband als auch im nationalen Verband. Dafür

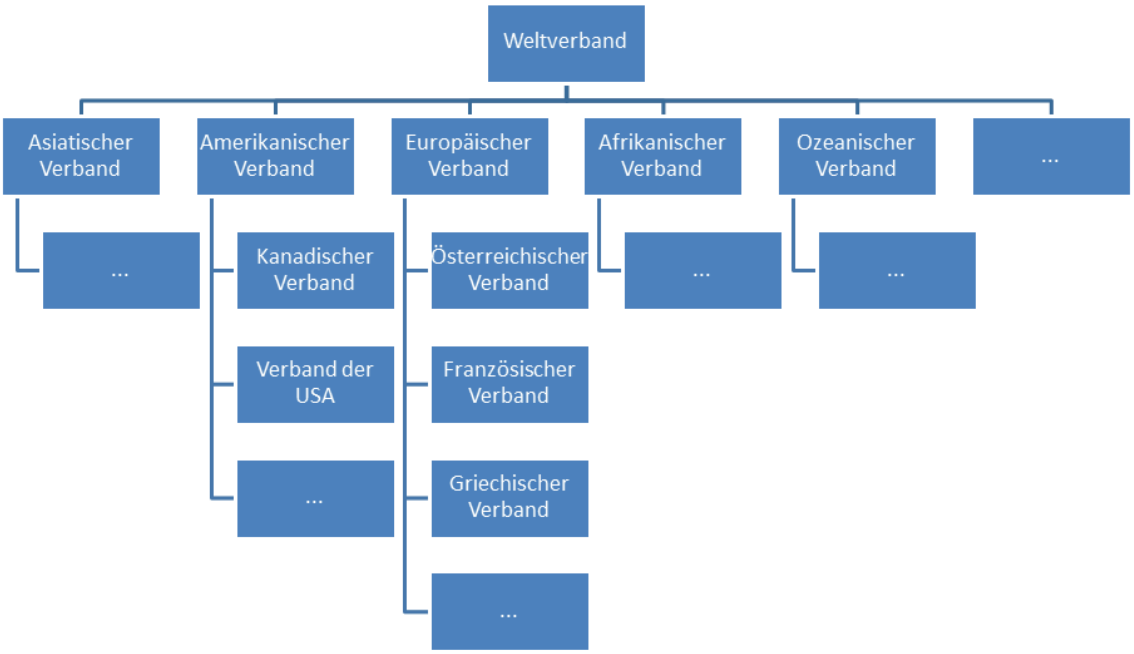

Abbildung 3.1: Gliederung des Sport-Weltverbandes

werden die Vorder- und Rückseite der einzelnen Karten individuell bedruckt. Nicht nur Namen und Land sind in verschiedenen Sprachen abgebildet, auch Logo und Verbandszugehörigkeit werden je nach Anmeldung individualisiert aufgedruckt.

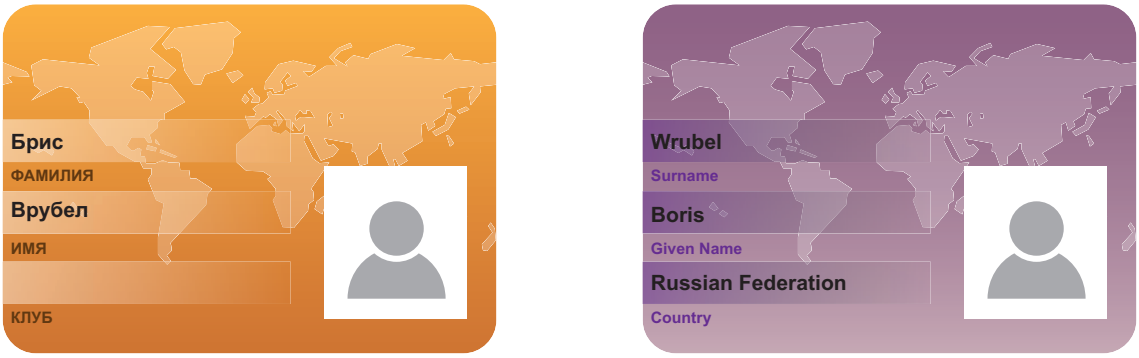

Abbildung 3.2: Schematische Darstellung der Vorder- und Rückseite einer ID Karte

Auf der Karte selbst sind keine persönlichen Daten elektronisch gespeichert, lediglich die Mitglieder ID und der dazugehörige Sicherheitsmechanismus sind auf dem Chip abgelegt. Die Karte kann, solange diese nicht beschädigt wird, über viele Jahre verwendet werden, da laut Hersteller über 100.000 Schreibzyklen pro Karte möglich sind. Wie der Kartenlebenszyklus aussieht, ist in Abbildung 3.3 dargestellt.

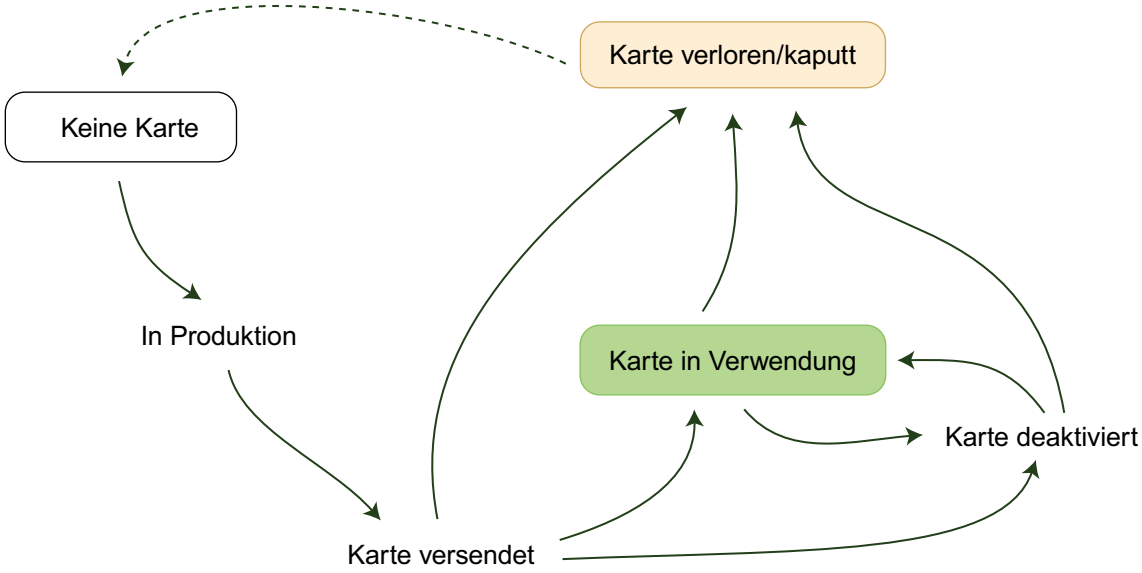

Abbildung 3.3: Kartenlebenszyklus im Projekt

Das gesamte Mitgliederverwaltungssystem besteht aus folgenden Teilsystemen und Komponenten. Diese sind in Abbildung 3.4 ebenfalls schematisch dargestellt.

Bestell- und Zahlsystem
Dieser Webshop bietet Personen die Möglichkeit, eine Mitgliedschaft zu erwerben.

Kartenproduktion
Nach einer Bestellung wird die Kartenproduktion durch den Export der entsprechenden Daten automatisch angestoßen.

Zentralsystem
Im zentralen System ist die komplette Businesslogik hinterlegt und die Daten werden in der Datenbank gespeichert.

NFC-Karte
Die Software, die auf der Java Smart Card läuft, stellt sicher, dass kein unberechtigter Zugriff stattfindet und enthält die Verschlüsselungstechnik. Es ist möglich, bei Bedarf weitere Java Applikationen auf einer sogenannten Multi Application Smart Card[1] zu installieren, um zusätzliche Funktionalitäten für zukünftige Anwendungen anbieten zu können.

NFC-Kartenleser (Hardware und embedded Software)
Auf dem eigens produzierten NFC Kartenleser-Terminal läuft eine an das Projekt angepasste Firmware.

Schnittstelle für Veranstaltungsmanagement
Es wird eine JSON-Schnittstelle angeboten, über die notwendige Daten vom Zentralsystem an die Veranstaltungssoftware weitergeleitet werden können, damit die Mitglieder bei einer Veranstaltung authentifiziert werden können.

[1] Die Karte ist mit dem Betriebssystem MULTOS (Mulit-Application Smart Card Operation System) ausgestattet. Nähere Informationen dazu auf der Website des MULTOS Konsortiums [56].

Browser Plug-in für Authentifizierung
Das Browser Plug-in ermöglicht, in Verbindung mit dem Kartenleser, eine Authentifizierung der Mitgliedskarten.

Torwächter
Ein Webserver mit integriertem Sicherheitssystem ist als Torwächter eingerichtet, um das Zentralsystem gegen nicht autorisierte Zugriffe zu schützen.

Schlüsselmeister
Hier werden alle Schlüssel verwaltet, um Karten authentifizieren zu können.

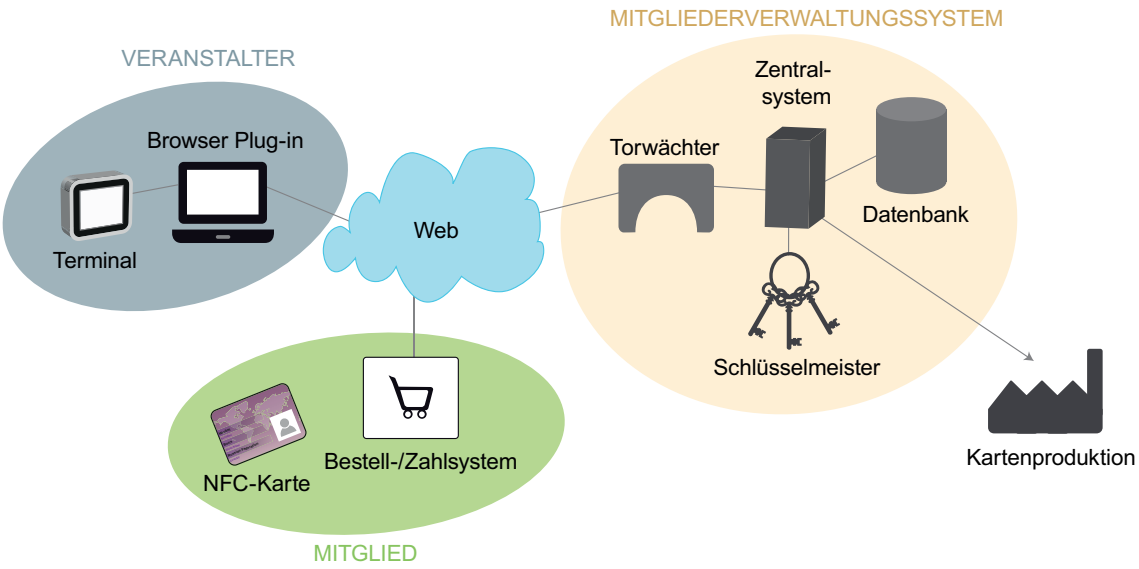

Abbildung 3.4: Systemlandschaft im Projekt

Einige weitere Anforderungen an das System stellen die komplexe Verbandsstruktur, sowie internationale Rahmenbedingungen deutlich klar. So sollen zum Beispiel für unterschiedliche Verbandsebenen verschiedene umfangreiche Auswertungsmöglichkeiten möglich sein.

Für die auf das Zentralsystem zugreifenden Personen soll weiters ein Rollenkonzept und ein Berechtigungssystem implementiert werden. Dabei stellt der Datenschutz eine besondere Herausforderung dar, da in den verschiedenen Nationalverbänden gänzlich unterschiedliche Richtlinien und Gesetze gelten. Beispielsweise dürfen Daten von Mitgliedern in einigen Ländern nur zeitlich begrenzt gelesen werden.

Für alle Datenbearbeitungen im System gilt, dass diese lückenlos historisiert werden. Es muss zu jedem Zeitpunkt festgestellt werden können, welche Daten von welchem Benutzer zu welcher Uhrzeit verändert wurden.

Generell spielen in dem Projekt viele unterschiedliche Komponenten zusammen, was eine Herausforderung vor allem für die Integrationsphase darstellt. Die Komplexität und Internationalität des Auftraggebers stellen dabei sowohl auf technischer Ebene als auch auf kultureller Ebene weitere Herausforderungen dar.

3.2 Technisches Projektumfeld

Das Projekt wurde von Anfang an als agiles Scrumprojekt geplant. Ein erfahrener Scrummaster sowie ein kompetenter Projektleiter sind die zentralen Personen dieses Projektes. Das Entwicklungsteam besteht aus vier Entwicklern, die bereits Erfahrungen in agilen Projekten haben und mit dem agilen Projektablauf vertraut sind. Die Sprintdauer beträgt zwei Wochen. Im Scrumteam arbeitet außerdem ein kleines Testteam, dieses hat jedoch im agilen Projektumfeld keine Erfahrung.

Die Spezifikationen werden mit Hilfe von User Stories umgesetzt. Diese werden in Kunagi (Kurzbeschriebung siehe Abschnitt 4.4.2) verwaltet. Die Priorisierung erfolgt in jedem Sprint vom Product Owner, der größtenteils vor Ort zur Verfügung steht.

Die Systemkomponenten „Torwächter“ und „Schlüsselmeister“ (siehe Abbildung 3.4) werden durch andere Teams entwickelt. Beide Komponenten sind teilweise aus Vorprojekten vorhanden und müssen dementsprechend angepasst werden. Das Teilsystem Torwächter ist für die Sicherheit des Zentralsystems zuständig und stellt sicher, dass nur qualifizierte Anfragen vom Schlüsselmeister oder dem Zentralsystem verarbeitet werden. Die Komponente Schlüsselmeister ist für sämtliche Verschlüsselungs- und Entschlüsselungstätigkeiten zuständig. Die Kartenproduktion wird von einer externen Firma vorgenommen, die in diesem Bereich weltweit anerkannter Spezialist ist.

Das Projekt wird im Abstimmung mit dem Sport-Weltverband durchgeführt, jedoch ist der Sport-Welt-verband weder der Product Owner noch wird das Projekt vom Sport-Weltverband finanziert. Das Finanzierungsmodell sieht vor, dass die Finanzierung ausschließlich über den Verkauf von Mitgliedschaften (Verbandsmitgliedschaften und Einzelmitgliedschaften) erfolgen soll. Daher müssen die Projektkosten auf die zu erwartenden Mitgliedschaften kalkuliert werden. Das volle Entwicklungsrisiko trägt der Product Owner, welcher Teil der beauftragten Firma ist.

Im Jänner 2013 wurde das Produkt als Prototyp bei einer Mitgliederveranstaltung vorgestellt und wohlwollend angenommen. In Folge ist eine schrittweise Einführung der Funktionalitäten geplant. So soll ein weiteres Release im April 2013 ausgerollt werden, bei dem ausgewählte User Stories bereits umgesetzt sein müssen. Im Juli 2013 soll die volle Funktionalität im Rahmen einer Veranstaltung in Betrieb gehen.

Diese schrittweise Einführung ist eine optimale Vorgehensweise für ein Scrumprojekt, da bei jedem Release die vorhandene Funktionalität geprüft wird und eventuell die User Stories adaptiert werden. Zudem werden die Funktionalitäten im Echteinsatz erprobt.

3.3 Problemanalyse

Die bisher beschriebene komplexe Verbandsstruktur und die internationalen Anforderungen sind zwar herausfordernde Parameter dieses Projektes, jedoch überraschender Weise keine wirklichen Problemverursacher. Die wahren Herausforderungen aus Testsicht liegen an ganz anderer Stelle.

Nach den ersten drei Sprints ist das Team mit der Performance nicht hundertprozentig zufrieden. So sind beispielsweise einige User Stories am Sprintende fertig programmiert,

jedoch werden diese von den Testern nicht freigegeben, da die User Stories nicht ausreichend getestet wurden. Sprintende ist alle zwei Wochen, immer Freitagnachmittag. Die Tester beklagen sich jedoch, dass an diesen Freitagen immer noch bis zu Mittag energisch neuer Programmcode geschrieben wird und sie im Anschluss unmöglich genug Zeit haben, um bis zum Ende des Sprints ausreichend zu testen.

Auf Grund der ausgesprochen kurzen Dauer für den Test, wurden lediglich einige wenige Testfälle durchgeführt, jedoch wurden diese nicht vollständig dokumentiert. Somit wurden die entsprechenden Testschritte vom Testteam nicht erfasst, wodurch es anderen Teammitgliedern schwer fiel, Testdurchführungen nachzuvollziehen oder ein gewisses Verständnis für die Testabläufe zu entwickeln. Dies führte nicht nur zu Qualitätseinbußen, sondern auch zu Spannungen im Team.

Gleichzeitig leiden die Tester gerade zu Sprintbeginn massiv an Unterbeschäftigung. Zu diesem Zeitpunkt werden neue User Stories umgesetzt und es ist keine lauffähige Version vorhanden, um Akzeptanztests mit den neu umgesetzten User Stories durchzuführen. So können lediglich User Stories und Fehlernachtests durchgeführt werden, jedoch betreffen diese hauptsächlich User Stories aus dem vorhergehenden Sprint.

Auf den Punkt gebracht ist das Problem, dass Entwicklungsmodell und Testmodell nicht effektiv zusammenarbeiten, einander im Weg stehen und somit auch nicht die geforderte Softwarequalität liefern können.

3.4 Motivation und Zielsetzung

Mit dem agilen Ansatz in der Softwareentwicklung verändert sich gleichzeitig die Rolle den Testers. Das stringente Abarbeiten von Testfällen und Prüfungen gegen die Spezifikation kann im agilen Umfeld nicht mehr jene Ergebnisse liefern, wie im klassischen Umfeld. Das Tätigkeitsfeld des Testers erweitert sich somit um einige Aufgaben. Und auch das Ableiten von Testfällen aus User Stories oder Spezifikationen ist im agilen Umfeld eine aufwändigere und kreativere Arbeit als in konservativen Vorgehensmodellen.

Klares Ziel im Projekt war es, ein zum agilen Entwicklungsmodell passendes Testmodell zu finden, damit die beiden Teile optimal zusammenspielen. Im Folgenden werden die verschiedenen damit zusammenhängenden Strategien erläutert, die Tester in den agilen Entwicklungsprozess einzugliedern. Darüber hinaus wird ein beispielhafter agiler Testprozess herausgearbeitet und definiert.

4 Implementierung von agilen Testmethoden und Testautomatisierungsstrategien

Das Projekt zeigt einige typische Probleme, die auftreten, wenn klassischer Test in einem agilen Projekt eingesetzt wird. Anhand dieses Projektes wird nun ausgearbeitet, wie die Implementierung von agilen Testmethoden in einem agilen Projekt den entscheidenden Vorteil bringen kann. Es scheint zwar auf der Hand zu liegen, in einem agilen Entwicklungsprojekt ebenfalls auf agilen Test zurück zu greifen, Tatsache ist jedoch, dass sich der agile Test derzeit noch nicht vollständig als Standard in der Praxis durchgesetzt hat.

Im Folgenden werden nun einige essentielle Maßnahmen beschrieben, die sich in der Praxis als wichtige Elemente des agilen Tests erwiesen haben. Es wird ein beispielhafter Testprozess vorgestellt und die Wichtigkeit der Continuous Integration aufgezeigt. Des Weiteren wird auf Testautomatisierung im agilen Umfeld eingegangen und die praktische Umsetzung von Akzeptanztests gezeigt. Außerdem werden einige Werkzeuge vorgestellt, die agile Modelle unterstützen und auch für den agilen Test sinnvoll nutzbar sind. Der wichtigste Faktor im agilen Umfeld ist jedoch immer noch der Mensch, daher wird abschließend auf das gewandelte Rollenbild des Testers in agilen Projekten eingegangen.

4.1 Agiler Test

Wie in der Problembeschreibung erwähnt, sind im beschriebenen Projekt die Entwicklung und der Test nicht im selben Rhythmus. Während die Programmierer agil entwickeln, wird im Test nach klassischen Methoden vorgegangen. Dies führte jedoch nicht zu den gewünschten Ergebnissen.

Als erste Maßnahme, um den Testbereich im Projekt zu verbessern, wurde einen Tag vor Sprintende ein sogenanntes „Feature Freeze“ eingeführt. Ab diesem Zeitpunkt war es nicht mehr gestattet, Code mit neuer Funktionalität einzuchecken. Einchecken ist dann nur noch für die Fehlerbehebung und zu Testzwecken gestattet. Mit dieser Maßnahme wurde am letzten Tag des Sprints die volle Konzentration auf Test, Testautomatisierung, Review und Bugfixing gerichtet. Somit wurde im gesamten Team das Bewusstsein geschaffen, dass ohne Test keine User Story erfolgreich abgeschlossen werden kann.

4.1.1 Klassisches und agiles Testvorgehen im Vergleich

Im Projekt war das grundlegende Problem, dass das Testteam stark im Vorgehensmodell des klassischen Testprozesses (siehe Kapitel 2.1.1) verhaftet war. Denn in einem agilen Entwicklungsumfeld führt diese klassische Herangehensweise automatisch zu Proble-

men. Welche relevanten Unterschiede zwischen klassischem und agilem Vorgehen existieren, zeigt Tabelle 4.1 und illustriert damit gleichzeitig die konträren Welten, die hier aufeinander treffen.

Klassisches Vorgehen	Agiles Vorgehen
Planung für gesamte Projektdauer	Planung für nächsten Sprint
Projektplan ändert sich nur in Ausnahmefällen oder bei Problemen	Projektplan ändert sich ständig auf Grund der aktuellen Gegebenheiten im Sprint und des Feedbacks
Lange Entwicklungszyklen	Kurze Entwicklungszyklen
Fokus auf Funktionsumfang	Fokus auf Zeit
Keine phasenübergreifenden Änderungen	Ständige Veränderung des Gesamtsystems
Anforderungen dürfen nicht mehr verändert werden nach Ende der Analyse- und Designphase	Laufende Änderungen und Anpassungen der Anforderungen
Fokus auf spezifizierte Anforderungen	Fokus auf Bedarf des Kunden
Testfälle werden auf Basis von Anforderungen erstellt	Tests basieren auf User Stories und Erfahrungsaustausch mit dem Product Owner
Tests auf Basis von Anforderungen	Tests auf Basis des vorhandenen Produktes / User Story
Alle Tests werden auf Testversion durchgeführt	Laufender Test (auch von Zwischenversionen)
Sequentielle Abfolge der Teststufen	Alle Teststufen pro Sprint vorhanden

Tabelle 4.1: Vergleich von klassischem und agilem Testvorgehen

Resultierend aus den verschiedenen Vorgehen ergeben sich für den Tester unterschiedliche Abläufe und Herangehensweisen für das Erarbeiten der Testfälle. Abbildung 4.1 und Abbildung 4.2 stellen die unterschiedlichen Testabläufe im klassischen und agilen Umfeld dar.

In klassischen Vorgehensmodellen werden die Testfälle ausschließlich aus den bereits abgenommenen Spezifikationen abgeleitet. Da mit dem Test frühzeitig angefangen wird, werden die Testfälle erstellt, bevor noch eine erste ausführbare Version des SUT vorliegt. Die Entwicklung leitet ebenfalls sämtliche Funktionalitäten aus den Spezifikationen ab, die sie dann in Programmcode umsetzen. Das Problem liegt auf der Hand: Bei diesem Vorgehen läuft man Gefahr, dass Entwickler und Tester ein anderes Verständnis der Spezifikationen haben oder die Spezifikationen unterschiedlich interpretieren. Mitunter kommt es auch vor, dass sich die Auslegungen der Spezifikationen nicht nur voneinander unterscheiden, sondern auch völlig anders sind, als jene des Business Analysten und des Kunden. Diese Diversifikation wird jedoch erst zum Schluss bei der Durchführung der Testfälle aufgezeigt. Dabei wäre es wesentlich effizienter, wenn sämtliche Unklarheiten bereits ausgeräumt werden, bevor Tester und Entwickler mit ihrer entsprechenden Arbeit fortfahren.

Bei agilen Vorgehensmodellen werden die Testfälle anhand der User Stories erarbeitet. Diese werden sowohl mit Entwicklern als auch mit dem Product Owner weiter begutachtet, um die korrekten Akzeptanztests definieren zu können. Somit werden Unklarhei-

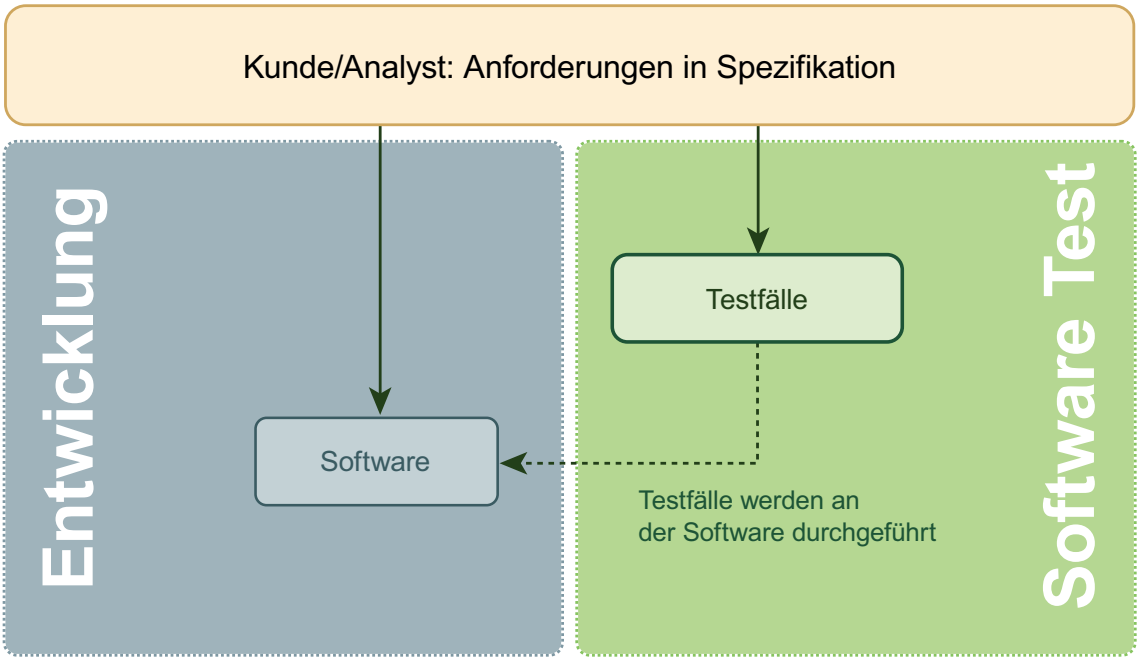

Abbildung 4.1: Testablauf in klassischen Modellen

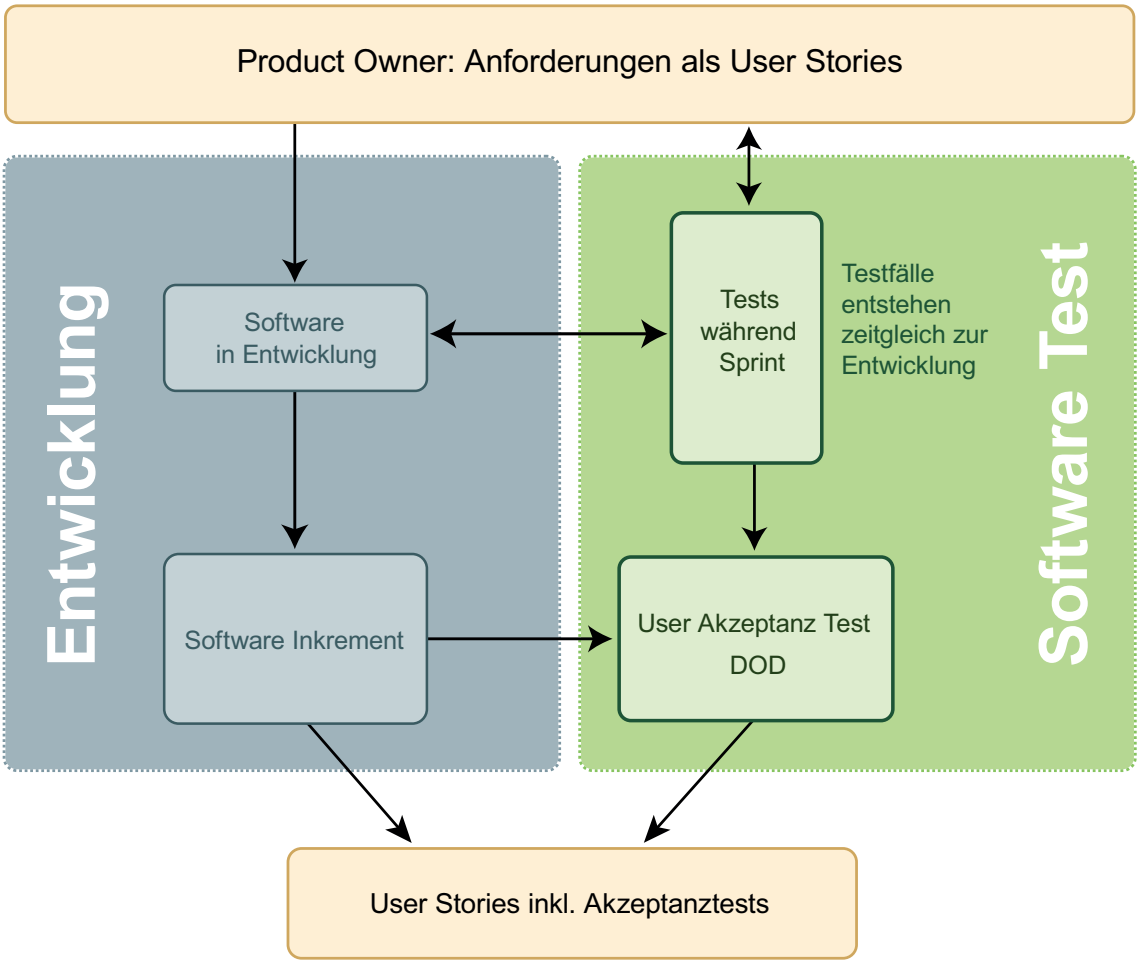

Abbildung 4.2: Testablauf in agilen Modellen

ten und mögliche falsche Interpretationen ausgeschlossen und alle Sichtweisen auf einen Nenner gebracht.

4.1.2 Agiler Testprozess

Ausgehend vom Projektumfeld lässt sich nun ein beispielhafter Testprozess herausarbeiten, der optimal auf dieses agile Vorgehensmodell abgestimmt ist und auch in anderen agilen Projekten eingesetzt werden kann. Abbildung 4.3 stellt diesen beispielhaften agilen Testprozess für Scrum dar. Nachdem der Sprint Backlog mit den ausgewählten User Stories befüllt worden ist (siehe dazu Abschnitt 2.2.5.5), hat der Tester die Aufgabe, für jede dieser Stories Akzeptanzkriterien herauszuarbeiten und logische Testfälle zu definieren. Diese Akzeptanzkriterien werden vom Tester mit dem Product Owner abgestimmt, die Erkenntnisse daraus tauscht er wiederum mit den Entwicklern aus, um Fehlinterpretationen auszuschließen. Gleichzeitig werden hier die Testfälle mit bereits bekannten Testmethoden erstellt (siehe dazu 2.1.4). Diese müssen ebenfalls mit dem Product Owner reviewed werden. Weiters wird die DOD (siehe 2.2.5.4) festgehalten, die eine erfolgreiche Umsetzung einer User Story festlegt. In weiterer Folge müssen gegebenenfalls die Testumgebung vorbereitet und konkrete Testdaten für eine Testdurchführung vorbereitet werden.[1]

In der Praxis kommt es der Erfahrung nach immer wieder vor, dass sich Änderungen an bereits umgesetzten und abgenommenen User Stories ergeben. Im Projekt haben sich zwei Möglichkeiten herauskristallisiert, wie man diese behandelt. Erstens, wenn die Änderung mit sehr geringem Aufwand umgesetzt und getestet werden kann, ist es erlaubt, die User Story abzuändern. Sind, zweitens, aber größere Änderungen notwendig, soll eine neue User Story angelegt werden. Erkenntnisse aus der ursprünglichen User Story werden jedoch dokumentiert, um die neu geschaffene Story effizienter umzusetzen. Die Dokumentation von Testdurchläufen ist sehr wichtig, um zu einem späteren Zeitpunkt feststellen zu können, was getestet und was nicht getestet wurde, denn so können Fehlerursachen eingegrenzt werden.

An dieser Stelle sei noch einmal explizit darauf hingewiesen, dass die Tests natürlich auch angepasst werden müssen, wenn sich die dazugehörige User Story ändert. Das scheint zwar völlig logisch, die Praxis hat jedoch immer wieder gezeigt, dass dieser Punkt übersehen wird. Auch hier ist der Schlüsselfaktor die Kommunikation: Wenn eine User Story geändert wird, müssen alle Involvierten informiert werden, damit sie die entsprechenden Anpassungen vornehmen können.

Daher lassen sich folgende Phasen des agilen Testprozesses identifizieren:

Analyse & Design
: In dieser Phase müssen für jede User Story die Akzeptanzkriterien definiert werden und mit Product Owner und Entwicklern abgestimmt werden, um alle Interpretationen einzuarbeiten und eine gemeinsame Sichtweise zu entwickeln.

[1] Eine Anleitung zum Überführen von logischen Testfällen in konkrete Testfälle ist zu finden in [77].

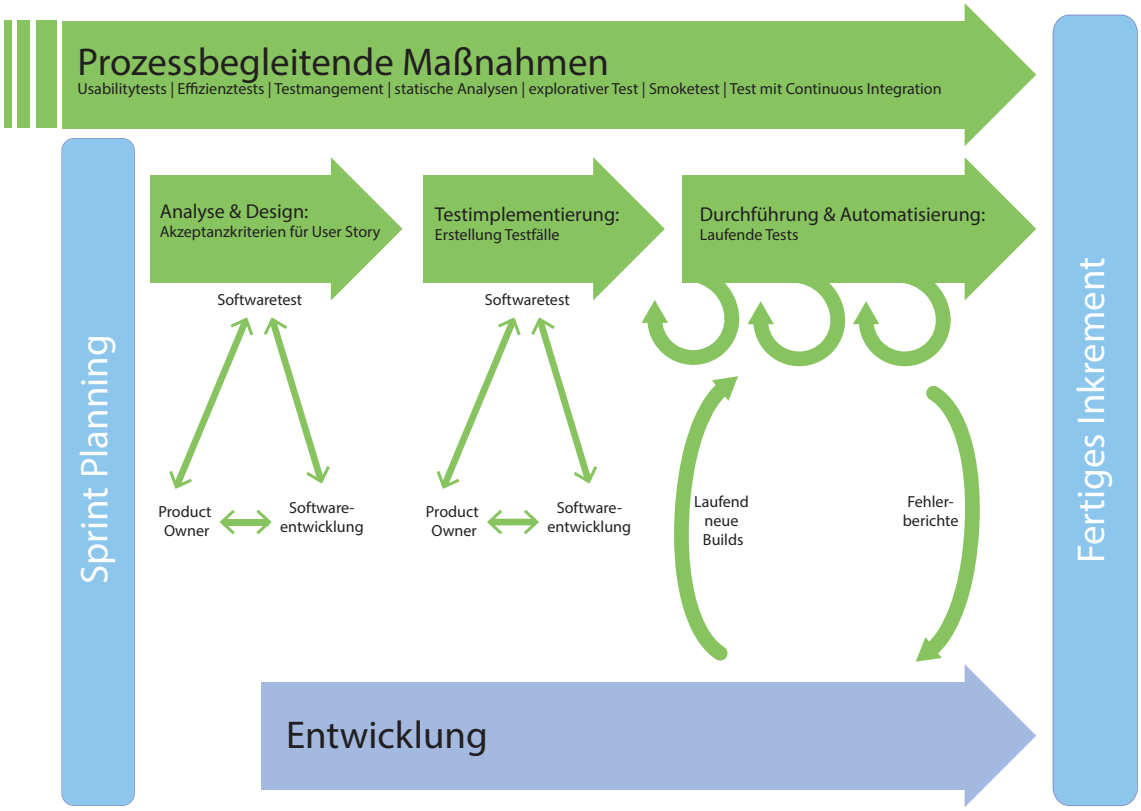

Abbildung 4.3: Beispielhafter agiler Testprozess

Testimplementierung

Parallel zur Programmierung der User Story, entwirft der Tester die entsprechenden Testfälle. Hier werden die bereits bekannten und bewährten Testmethoden angewandt (siehe dazu 2.1.4).

Durchführung & Automatisierung

Während der Testdurchführung werden Fehler aufgezeigt und die Fehlerberichte werden den Entwicklern kommuniziert, damit die Fehler analysiert und beseitigt werden können. Testfälle, die für eine Automatisierung geeignet sind, werden identifiziert und wenn möglich gleich in automatisiert ausführbare Testscripts überführt. Die Testdurchführung muss dokumentiert werden, um Fehlerquellen leichter identifizieren zu können.

Parallel zu den genannten Phasen führt der Tester laufend explorative Tests, Smoketests und die automatisierte Version der Smoketests mit Hilfe des Continuous Integrationservers (siehe dazu Abschnitt 4.3) durch. Als Smoketests werden eine Auswahl an Testfällen bezeichnet, die einen besonders hohen Stellenwert haben. Oft werden diese Testfälle zuerst durchgeführt und nur bei positiver Durchführung dieser Testfälle wird mit den restlichen fortgefahren. Andernfalls müssen zuerst Fehler beseitigt werden bevor man die weiteren Testfälle behandelt.

Weiters müssen die nicht funktionalen Tests ebenfalls parallel zum beschriebenen Testprozess durchgeführt werden, wie dies auch in Abbildung 4.3 dargestellt ist. Es gilt einerseits laufend Effizienztests durchzuführen, um zu prüfen, ob vereinbarte SLAs einge-

halten werden, sowie andererseits, die Benutzeroberfläche durch Usabilitytests auf deren Bedienbarkeit zu prüfen. In dem beschriebenen Projekt wurden für diese Tätigkeiten externe Experten hinzugezogen, da die internen Ressourcen nicht ausreichten, um auf die Spezifika der weltweit einzusetzenden Software und die damit verbundenen vielfältigen Anforderungen an die Benutzbarkeit einzugehen. Die Durchführung von Effizienztests wurde ebenfalls extern vergeben, da die erwartenden Zugriffe in Spitzenzeiten von mehr als 10.000 gleichzeitigen Benutzern aufwändig zu simulieren sind.

4.1.3 Testmanagement in agilen Modellen

Klar ist: Duch die Rolle des Testmanagers ändert sich stark in einem agilen Umfeld. In dem hier betrachteten Projekt wurde die Rolle des Testmanagers gar nicht gelebt. Entscheidungen wurden im Team beschlossen, jedoch waren diese nicht immer eindeutig. Auch wurden diese Entscheidungen nicht nach außen kommentiert, was immer wieder zu Verwirrung, Unklarheiten oder Missverständnissen führte. Linz beschreibt in [51], dass in einem Scrum Team kein dedizierter Testleiter vorgesehen ist. Das Testmanagement liegt in dieser Definition im Verantwortungsbereich des Scrum Masters: Sämtliche Testaufgaben werden im Sprint explizit geplant oder implizit als Teilaufgabe der dazugehörigen User Story definiert. Ein Continuous Build (siehe Abschnitt 4.3) ergänzt oftmals ein klassisches Reporting, da ohnehin der tagesaktuelle Teststand für die bereits automatisierten Testfälle zur Verfügung steht.

Linz ist in [51] der Meinung, dass sämtliche Testmanagementaufgaben durch einen gut ausgebildeten und erfahrenen Tester im Scrum Team – in Abstimmung mit dem Scrum Master – übernommen werden können und sollen. Auch wenn Scrum keine dedizierten Tester vorsieht, ist der Autor dieser Arbeit der Meinung, dass ausgebildete Tester, entsprechende Testmanagementaufgaben effizienter durchführen können, sie die Arbeit der Entwickler besser unterstützen und sich die Arbeit des Testmanagers positiv auf das Team auswirkt.

Betrachtet man nun den agilen Testprozess, die stark veränderten Aufgabenfelder und das damit gewandelte Rollenbild des Testers, ist festzustellen, dass der Tester nun während des gesamten Projektes vollständig in die Abläufe integriert ist. Er kann dadurch von Anfang an das Projekt unterstützen und inhaltlich vorantreiben. Stehzeiten werden reduziert und Spitzenbelastungen ebenfalls abgefedert. Der Test ist nun im Ablauf nun der Entwicklung nachgelagert sondern ein integrativer Bestandteil des agilen Projektes.

4.2 Testautomatisierung im agilen Umfeld

Für die Akzeptanztests, also jene Tests, die feststellen, ob eine User Story komplett und korrekt umgesetzt wurde, wurde im hier analysierten Projekt Cucumber verwendet. Cucumber ist ein Softwarewerkzeug, das ermöglicht, funktionale Beschreibungen in textbasierter Form zu erstellen und diese dann automatisch auszuführen (näheres dazu im Abschnitt 4.4.1). Im Projekt wurde also von Anfang an mit diesem Werkzeug Testautomatisierung durchgeführt, jedoch wurde die Automatisierung nicht konsequent implementiert und effektiv eingesetzt.

Diese Herangehensweise war zwar grundsätzlich sinnvoll, es stellte sich aber heraus, dass der Wartungsaufwand der Automatisierungsscripts enorm war. Der Grund dafür liegt in der Natur eines agilen Projektes: Die Software ändert sich – gerade zu Beginn – stark und grundlegend. Solche fundamentalen Änderungen der Software sind für die Testautomatisierung ein Problem, da es schwierig wird, stabile Testscripts zu erstellen, die fehlerfrei durchgeführt werden können. Wenn sich Funktionalitäten aber häufig ändern, müssen auch die Automatisierungsscripts ständig angepasst werden und das lässt wiederum die Aufwände drastisch steigen. Das Argument, dass man in diesen Bereichen doch einfach auf Testautomatisierung verzichten und stattdessen auf manuellem Wege testen solle, ist in diesem Fall jedoch nicht zulässig. Die Testfälle sind einerseits zu umfangreich und andererseits werden die Testfälle in den Sprints so häufig durchgeführt, dass man mit manuellem Test nicht nachkäme.

Modularisierung heißt der Ansatz, der Testautomatisierung auch in einem agilen Umfeld attraktiv macht, da sie den Aufwand möglichst gering hält. Hierbei wird ein Testfall in einzelne Aktionen oder Module zerlegt, wie von Seidl, Baumgartner und Bucsics in [72] erläutert. Wenn beispielsweise ein Testfall lautet *„Mitglied für eine Veranstaltung anmelden“*, wurde bisher das Testscript für den gesamten Testfall angelegt. Beim modularen Ansatz wird der Testfall aber in kleinere Module zerlegt, hier zum Beispiel vereinfacht in: *„Einloggen“*, *„Veranstaltung suchen“* und *„Anmelden“*. Wenn sich nun Änderungen im Modul *„Veranstaltung suchen“* ergeben, welches übrigens auch noch in einigen anderen Testfällen verwendet wird, muss das Testscript nur für dieses Modul geändert werden und alle davon abhängigen Testfälle können weiterhin automatisiert laufen. In Kapitel 4.4.1 wird anhand eines Beispiels weiterführend erklärt, wie ein solches Modul noch in weitere, praktikable Schritte zerlegt wird.

Es ist Aufgabe des Testteams, festzustellen, welche Teile des Systems stabiler sind als andere, um genau diese für eine Automatisierung auszuwählen. In weiterer Folge entscheidet auch das Testteam, welche Module für die Automatisierung realisiert werden. Das Vorgehen dazu beschreiben Seidl, Baumgartner und Bucsics in [72]. Die Modularisierung wurde vom Team umgehend umgesetzt und setzt die Automatisierung nun deutlich effizienter um.

Wie ein konkreter Ablauf über mehrere Iterationen aussehen könnte, hat Singh in [73] beschrieben. In Abbildung 4.4 wird dieser Ablauf beispielhaft dargestellt, zudem sind hier zu wartende Testfälle hinzugefügt worden, was in der Praxis durch die Änderung einer Funktionalität notwendig geworden sein könnte. Dieses Vorgehen wird auch durch Crispin und Gregory in [19] bestätigt.

Die sorgfältige Auswahl der Automatisierungswerkzeuge ist besonders wichtig, da jedes Projektumfeld unterschiedliche Anforderungen mit sich bringt. Es gibt nicht ein für die agile Testautomatierung optimal geeignetes Testwerkzeug, sondern eine Vielzahl an Optionen. Kommerzielle Werkzeuge ermöglichen einen schnellen Einstieg, jedoch sind die Investitionskosten sehr hoch. Bei Open Source Werkzeugen muss Aufwand in die Anpassung und Weiterentwicklung investiert werden, um die Werkzeuge an das zu testende System anzupassen. Die Erfahrung hat gezeigt, dass fast ausnahmslos auch bei kommerziellen Werkzeugen Anpassungen vorgenommen werden müssen, auch wenn die Hersteller dieser Tools dies so selbstverständlich nicht kommunizieren. Dieses Problem tritt vor allem dann auf, wenn es sich um Individualsoftware handelt, in der eigens entwickelte oder angepasste Steuerelemente verwendet werden.

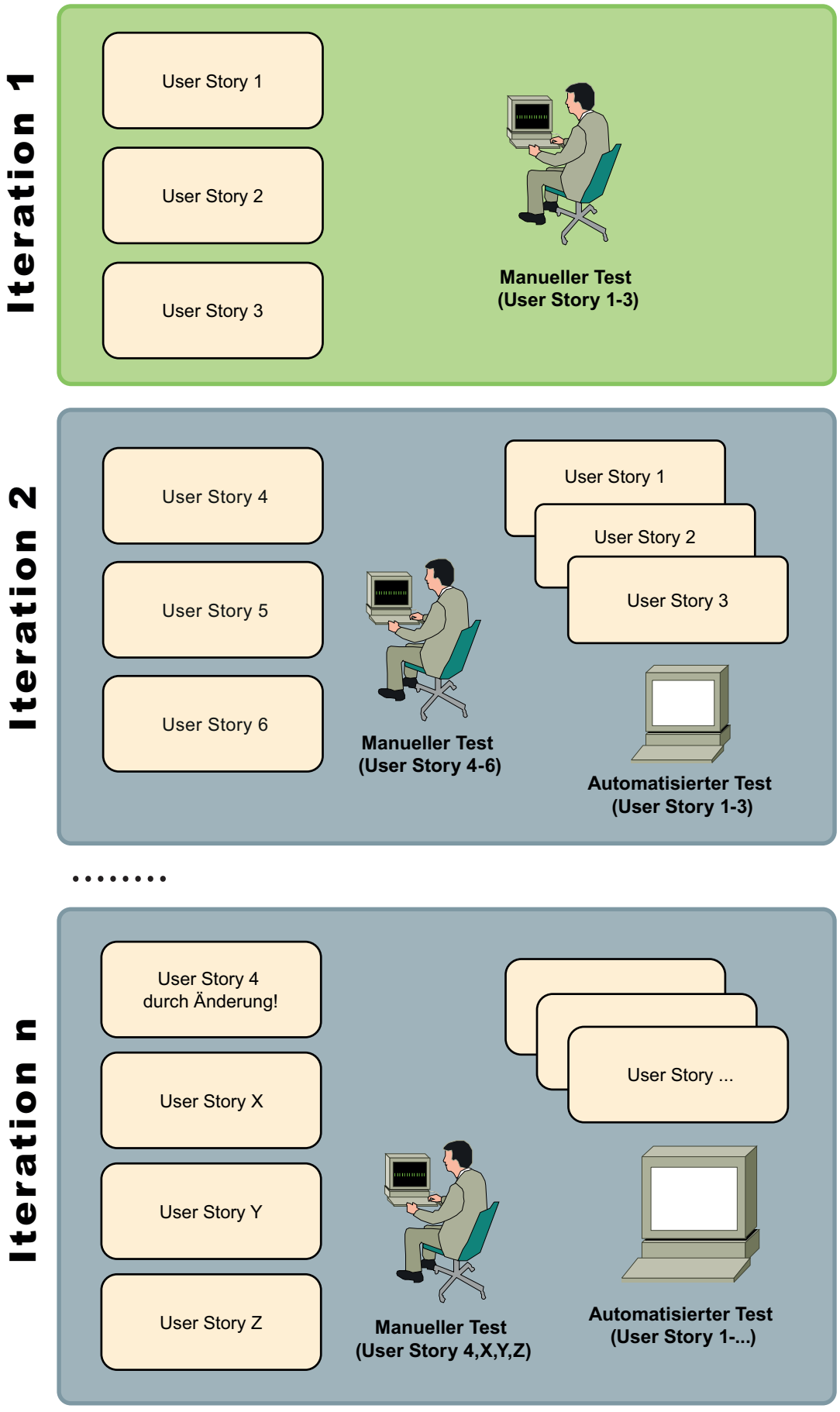

Abbildung 4.4: Testautomatisierung in iterativen Vorgehensmodellen

Im hier betrachteten industriellen Projekt wird mit einer Weboberfläche gearbeitet, hier sind meist keine zusätzlichen Adaptierungen notwendig. Daher hat man sich für Cucumber, ein Open Source Werkzeug entschieden. Mehr dazu folgt in Kapitel 4.4.1.

In Zusammenhang mit der Auswahl der Werkzeuge sei hier auch auf das Paper [2] verwiesen. Bach beschäftigt sich darin mit agiler Testautomatisierung für mittelgroße und große Projekte. Unter anderem weist er auf folgende Erkenntnis in Bezug auf Testwerkzeuge hin: Testwerkzeuge sollten nicht nur sorgfältig ausgewählt werden, man sollte auch bedenken, dass nicht zwangsläufig sämtliche Testautomatisierungstätigkeiten mit einer Software durchgeführt werden müssen – auch wenn dies gern von den Herstellern postuliert wird. Oft ist es aufwändiger, mit einem Werkzeug alles zu testen, als für beispielsweise einen Tabellenkalkulationsvergleich ein weiteres Werkzeug zu verwenden. Jedoch sollten die Ergebnisse der Tests an einer gemeinsamen Stelle zusammenlaufen. Somit ist sichergestellt, dass nur eine Informationsquelle für die Ergebnisse existiert.

Testautomatisierung kann im agilen Test, wie auch im klassischen Test, in verschiedenen Teststufen umgesetzt werden. Der Automatisierungsgrad sollte je nach Teststufe variieren. Crispin und Gregory beschreiben in [19] die Testautomatisierungspyramide von [17] in Relation zum klassischen V-Modell, einem Vorgehensmodell für Softwareentwicklung. Um die Zusammenhänge zu verdeutlichen, wird hier zur Ergänzung des V-Modells, wie in Abbildung 4.5 gezeigt, die Testautomatisierungspyramide hinzugefügt. Die Automatisierungspyramide ist eine besonders plakative Ergänzung zum V-Modell, weil sie die Gewichtung und Anzahl der empfohlenen Tests aufzeigt.

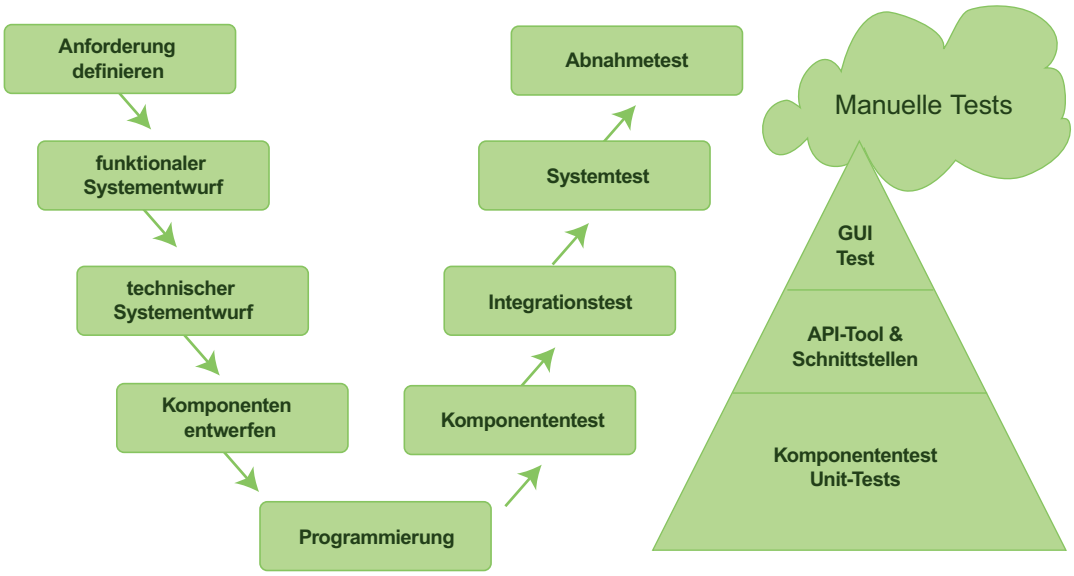

Abbildung 4.5: Teststufen und Testautomatisierungspyramide

Die Basis der Testautomatisierung, laut Testautomatisierungspyramide, bilden die Komponententests, die auch zahlenmäßig deutlich mehr sein sollten als die Integrationstests. Die Integrationstests des V-Modells werden mit automatisierten Schnittstellentests ergänzt. An der Spitze der Pyramide sind die GUI-Tests, die mit dem Systemtest gleichzusetzen sind, da hier alle Komponenten des Systems getestet werden müssen. Ergänzt wird die Pyramide durch die „undefinierbare“ Wolke der manuellen Tests. Undefinierbar ist diese Wolke deshalb, weil es hierfür keine Regel gibt, wie viele Tests in Relation zu den anderen Teststufen notwendig sein werden. Fest steht aber, dass die manuellen Tests immer notwendig sind, da ein vollständiges Automatisieren in der Regel weder wirtschaftlich noch technisch machbar ist.

Ein praktisches Beispiel dafür, dass sich der Bedarf an manuellen und automatisierten Testfällen durchaus ändern kann, liefert das zuvor beschriebene Projekt. Das Terminal und die Software für die NFC-Karte werden von anderen Teams entwickelt und getestet, daher sind für diese Komponenten ausschließlich Systemtests durchzuführen. Die Qualität der Komponenten selbst ist bereits durch Vorprojekte sichergestellt.

Diese Systemtests wurden anfänglich ausschließlich manuell durchgeführt, weil es diese Testfälle erfordern, dass eine Karte im richtigen Moment an das Terminal gehalten wird. Da dieser Testfall pro Sprint nur einmal durchgeführt wurde, waren die manuellen Testfälle für dieses Szenario mit geringem Aufwand durchführbar. Im Laufe des Projektes stellte sich jedoch heraus, dass ein Lasttest für diese Art von Testfällen notwendig wird. Es mussten nun mehrere tausend Authentifizierungsvorgänge mit einer Karte und einem Terminal durchgeführt werden. Somit wurde der Wunsch nach einer möglichen Automatisierung immer dringlicher.

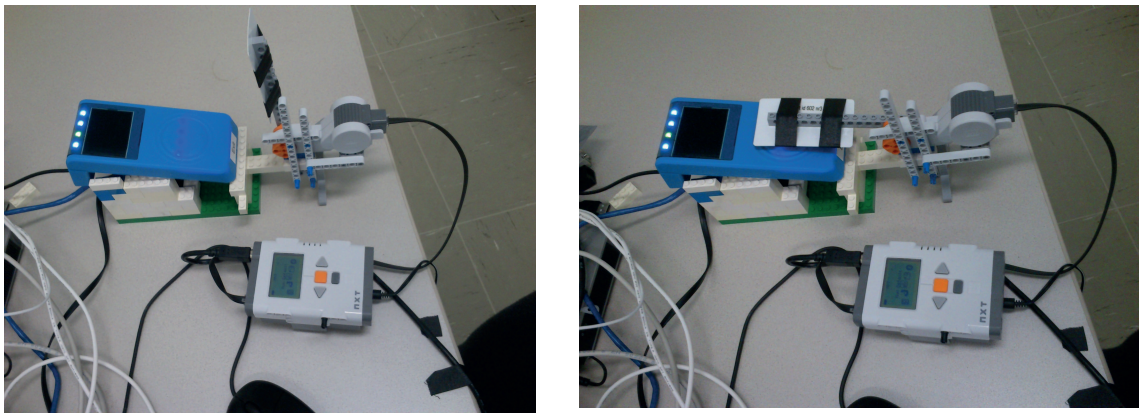

Abbildung 4.6: Testautomatisierung für Hardwarekomponenten

Durch einen kreativen Lösungsansatz wurde dieser Testfall, der lange Zeit manuell durchführt wurde, automatisiert. Mit dem Einsatz eines Kinderspielzeugs und der Möglichkeit, dieses mit Hilfe von Java-Code steuern zu können, wurde dieser interaktiver Testfall automatisiert. Abbildung 4.6 zeigt das Terminal und eine NFC-Karte, die von einem Arm gehalten und automatisiert an das Terminal herangeführt wird. Der Arm wurde mit Lego Mindstorms[2] konstruiert, auf dem jedoch eine alternative Firmware namens lejos[3] instal-

[2] http://mindstorms.lego.com/
[3] http://lejos.sourceforge.net/

liert wurde. Erst diese Firmware macht es möglich, die Steuerung des Roboterarms in die Testautomatisierung einzubinden, denn sie steuert das Verhalten des Arms abhängig von Status- und Fehlermeldungen des Terminals. Siehe dazu auch [23].

Die Grenzen der Automatisierung hängen nicht nur von der Machbarkeit ab, sondern auch stark von der Wirtschaftlichkeit. Während in dem hier beschriebenen Fall eine Automatisierung anfänglich weder notwendig noch wirtschaftlich gewesen ist, hat sich im Laufe des Projektes Gegenteiliges hervorgetan. Es gilt, die richtige Kombination aus manuellen Tests und Testautomatisierung für das eigene Projekt zu finden; die Grenzen können sich durchaus während des Projektes verschieben.

Die Vorteile der Einbindung von automatisierten Tests in den Sprintablauf sind leicht nachzuvollziehen:

Schnelles Feedback
: Bei Verwendung eines Continuous Integrationservers (siehe dazu Abschnitt 4.3) wird jeder Programmcode, der von den Programmierern freigegeben wird, sofort mit vorhandenen Testfällen geprüft. So liefert auch die Testautomatisierung schnelles Feedback, wie das bei agilen Modellen gewünscht ist (siehe dazu 2.2.6.1).

Kontrolle
: Wenn die automatisierten Testfälle fehlschlagen, heißt das für das Testteam, dass diese Teile des Systems erneut getestet werden müssen oder dass an diesen Stellen Änderungen gemacht wurden.

Ausschluss von Seiteneffekten
: Es wird sichergestellt, dass bereits abgenommene Funktionen weiterhin funktionieren und nicht durch Seiteneffekte Fehlerwirkungen hervorrufen.

Weiters können durch die Anwendung von strukturierten Testfallentwurfsverfahren wie zum Beispiel der Klassifikationsbaummethode [35], der Grenzwertanalyse [77] und anderen, sehr viele Testfälle erzeugt werden. Bei manueller Durchführung werden diese Testfälle üblicher Weise reduziert oder zum Beispiel nach Risiko ausgewählt. Die Testautomatisierung ermöglicht es, auf diese Reduktion teilweise zu verzichten und oft kann mit wenig Aufwand eine deutlich höhere Testabdeckung erreicht werden. Allerdings muss trotzdem beachtet werden, dass weitere Aufwände notwendig sind, um diese Testscripts zu erstellen. Oft werden die später anfallenden Wartungsaufwände für diese Testscripts übersehen, die bei Änderungen in dem SUTs entstehen, selbst wenn die Modularisierung angewendet wird.

Seidl, Baumgartner und Bucsics schildern in [72], wie ein logisches Konzept für eine Testautomatisierung aussehen soll. Gute Testautomatisierung ermöglicht es, Testfälle mit unterschiedlichen Daten wiederholt durchzuführen. Dazu werden die Testdaten und die eigentlichen Testfälle getrennt. Die Wertzuweisung der Parameter in den Testfällen erfolgt erst bei der tatsächlichen Testdurchführung. Diese Art der Testautomatisierung wird auch datengetriebener Test genannt.

Zusammenfassend wurden folgende Schritte im Projekt umgesetzt, um die Testautomatisierung effizient zu realisieren:

Modularer Aufbau
Der modulare Aufbau der Testautomatisierung erleichtert es, in darauffolgenden Sprints weitere Testscripts schnell zu erstellen (siehe auch 4.2). Die konkrete Vorgehensweise und ein einführendes Beispiel mit sogenannten Feature Files sind in Abschnitt 4.4.1 zu finden.

Continuous Integrationserver
Durch die Einbindung der Testläufe in den Continuous Integrationserver ist sichergestellt, dass die Testläufe regelmäßig laufen und diese Testläufe erzeugen somit schnelles Feedback.

Testautomatisierungspyramide
Bei der Menge der automatisierten Tests wurde, wie oben beschrieben, auf die Testautomatisierungspyramide (Abbildung 4.5) geachtet.

Teameinbindung
Die Durchführung der Akzeptanztests wurde auf das gesamte Team verteilt, dadurch haben die automatisierten Testläufe einen deutlich höheren Stellenwert im Team bekommen und wurden von allen vorangetrieben. Dieses Vorgehen ist auch in [33] beschrieben, mit ähnlich positiven Ergebnissen.

Punktueller manueller Test
Aufwändige Testfälle wurden vorerst von der Automatisierung ausgenommen, da in diesen Fällen regelmäßiger manueller Test wirtschaftlicher ist.

Statusbericht
Die täglichen automatisierten Testläufe liefern nun eine gute Basis für den regelmäßigen Statusbericht an Projektleiter und Product Owner und können von beiden direkt am Integrationserver eingesehen werden. Dieser Vorschlag wurde aus [33] übernommen.

4.3 Continuous Integration

Da im Projekt agil entwickelt wurde, war Continuous Integration von Beginn an ein fixer Teil des Projektes. Jedoch wurden lediglich die Komponententests damit durchgeführt, wie es in der agilen Entwicklung weit verbreitet ist. Das Testteam hat es aber verabsäumt, den Continuous Integration Prozess auch für die Testdurchführungen zu nutzen, was ihre Arbeit und vor allem die Zusammenarbeit mit dem Entwicklerteam vereinfacht hätte.

Aus diesem Grund hat man sich umgehend nach der Problemanalyse dazu entschieden, automatisierte Testfälle in die Continuous Integration einzubinden. Somit wurde sichergestellt, dass das komplette Testset beim Kompiliervorgang in den frühen Morgenstunden zumindest einmal täglich durchlaufen wurde.

Wie wichtig diese Praxis für die Testdurchführung ist, zeigt folgendes Beispiel und etabliert damit Continuous Integration als essentiellen Bestandteil des Tests.

Zu Beginn wurden die einfachen User Stories realisiert, die die Eingabe von Mitgliedern und deren Daten ermöglichen. Diese Stories wurden erfolgreich getestet und vom Product Owner abgenommen. Durch die weitere Implementierung von User Stories, wurde jedoch ein Fehler, der alle Datumsfelder betraf, in den Programmcode eingeschleust. Das Testteam erstellte daraufhin weitere Testfälle, um sicherzustellen, dass dieser Fehler in Zukunft nicht mehr auftritt. Das Testscript hatte vereinfacht folgende Testschritte:

- Einloggen in die Applikation
- Anlegen eines Mitglieds mit entsprechenden Daten (inkl. Geburtstag)
- Speichern der Daten
- Mitglied in der Übersichtsliste auswählen
- Geburtsdatum vergleichen

Im darauffolgenden Sprint schlägt dieser Testfall aber auf einmal fehl. Das Mitglied konnte in der Übersichtsliste nicht mehr aufgefunden werden. Was war passiert? Durch eine weitere Änderung im Quellcode, wurde der geänderte oder neu hinzugefügte Datensatz gar nicht mehr in der Übersichtsliste angezeigt. Dies war natürlich so nicht erwünscht und musste beseitigt werden. Jedoch wurde der Fehler bereits am nächsten Tag nach dem Einchecken des entsprechenden Codes aufgedeckt und der Entwickler hat diesen auch schnell finden können, da der Fehler am Tag zuvor eingebaut wurde. Dank der Continuous Integration wurde schnelles Feedback geliefert und so konnte der Fehler schnell identifiziert und ausgebessert werden.

In klassichen Projekten kommt es vor, dass viele Programmkomponenten erstellt werden und sobald diese fertig sind, werden diese zusammengefügt und die Funktionalität geprüft. Der agile Gedanke ist jedoch ein völlig anderer. Denn erst beim Zusammenfügen aller Programmkomponenten würden die Entwickler Feedback bekommen. Dies ist jedoch ein Zeitpunkt, der vom Erstellungstag des Codes schon weit entfernt ist. Durch das späte Feedback könnten bereits Folgefehler im Code passiert sein, eine Korrektur kostet zu diesem Zeitpunkt bereits viel Zeit und Geld.

Aus diesem Grund setzen agile Methoden auf Continuous Integration. Continuous Integration bedeutet, dass fertiger Programmcode sofort in die Integrationsumgebung überspielt wird. Dadurch erhält man unmittelbar eine Meldung, ob eine Kompilierung überhaupt möglich ist. Ist der Programmcode syntaktisch korrekt, können im Anschluss automatisierte Testfälle durchgeführt werden. Dadurch ist sichergestellt, dass sich der neue Programmcode nicht durch Nebeneffekte auf bisherige Funktionen auswirkt. Durch diesen Vorgang wird einer der wichtigsten agilen Grundsätze realisiert: Schnelles Feedback. In diesen Fall meldet der Compiler unmittelbar nach dem Einchecken, ob neuer Code integriert werden kann oder ob es Probleme gibt.

Auf einem Integrationsserver werden verschiedene Stufen von Tests durchgeführt, wie auch Linz in [51] beschreibt. Neben dem Compilerlauf können hier statische Codeanalysen durchgeführt werden, die die vom Team vereinbarten Kodierungsrichtlinien überprüfen.

Für den Tester interessant sind die automatischen Durchführungen von Komponenten-, Integrations- und Systemtests, oder auch Akzeptanztests. Komponententests dürfen in der Regel keine Fehler aufzeigen, da diese von den Entwicklern vorab in deren Entwicklungsumgebungen durchgeführt werden. Integrationstests prüfen das Zusammenspiel von mehreren Komponenten und deren Schnittstellen, hierfür werden entsprechende Werkzeuge benötigt, die sich von Projekt zu Projekt unterscheiden. Systemtests sollen das Verhalten und die Anwendungsfälle des Endbenutzers simulieren, um festzustellen, ob das System nach den Anforderungen funktioniert. Eine mögliche Herangehensweise für solche Tests ist im nächsten Abschnitt 4.4.1 beschrieben.

Weiters ist bei agilen Modellen unumgänglich, dass stets ein lauffähiges Programm zur Verfügung steht. Dadurch kann der Product Owner jederzeit sehen, wie User Stories umgesetzt sind. Für den Tester erleichtert es ebenfalls die Arbeit, da auch er jederzeit den aktuellen Stand testen, Fehlernachtests durchführen und Testscripts für automatisierte Testfälle erstellen kann.

Durch das Ausführen von automatisierten Akzeptanztests auf Benutzerebene, sieht der Entwickler sofort, wenn neu eingecheckter Programmcode unter Umständen andere Fehler aufdeckt, dadurch kann die Ursache schneller gefunden werden.

Wie auch Watkins in [82] bestätigt, konnte durch die Integration der automatisierten Tests in die Nightly Builds[4] signifikant Zeit eingespart werden.

Um Continuous Integration einzusetzen sind Fachwissen und Ressourcen notwendig, die sich jedoch durch die effizienteren Abläufe rasch amortisieren. Es ist unbedingt notwendig, dass diese Scripts auch ständig auf ihre Funktionalität überwacht werden und die täglichen Testausführungen ausgewertet werden.

4.4 Werkzeuge im agilen Test

Wie bereits herausgearbeitet, sind Softwareprojekte höchst unterschiedlich. Somit ist auch klar, dass die den agilen Test unterstützenden Werkzeuge auf das individuelle Projekt, die Anforderungen und das Umfeld abgestimmt werden müssen. Im Folgenden werden anhand des hier betrachteten Projektes einige hilfreiche Werkzeuge für den agilen Test vorgestellt, sowie Kriterien für deren Auswahl diskutiert.

4.4.1 Akzeptanztests mit Cucumber

Akzeptanztests und Testautomatisierung wurden zu Beginn des Projektes unterschiedlich behandelt. Die Akzeptanztests wurden manuell vom Testteam durchgeführt, um zu überprüfen, ob eine User Story vollständig erfüllt wurde. Nach und nach wurden einige wenige hoch priorisierte Akzeptanztests automatisiert, um diese auch in späteren Iterationen effizient durchführen zu können. Jedoch wurde die Automatisierung nicht konsequent voran-

[4] Da unter Umständen die Testläufe sehr lange dauern können, vor allem dann wenn diese über die Benutzeroberfläche realisiert werden, werden oft komplette Testläufe nur einmal pro Nacht durchgeführt (sogenannte Nightly Builds).

getrieben. Ein weiteres Problem war die Dokumentation. Durch die Zeitknappheit im Projekt, wurden einzelne automatisierte Testfälle nicht ausreichend dokumentiert. So konnten Teammitglieder nicht nachvollziehen, wie die Testschritte dieser Testfälle tatsächlich ablaufen. Die fehlende Dokumentation führte auch zu Problemen, wenn die Testscripts geändert werden mussten. Denn wenn die Testscripts geändert waren, fehlte oft die Zeit, diese Änderungen auch in der Dokumentation nachzuziehen. Somit hatten die Testscripts einen anderen Status, als in der Dokumentation abgebildet war. Ein unüberschaubares Chaos, das sich natürlich auf die Qualität des gesamten Tests auswirkte. Um all diese Probleme mit einem Schlag zu lösen, wurde nach einer Herangehensweise und einem dazu passenden Werkzeug gesucht, die sowohl Dokumentation als auch Abänderbarkeit und automatische Durchführung vereinen konnten. Behaviour Driven Development (näheres dazu in Kapitel 2.2.10) mit Cucumber als Werkzeug war diese Lösung.

Cucumber ermöglicht es, Anforderungen funktional in Textform zu beschreiben und diese Anforderungen zu einem späteren Zeitpunkt mittels Testautomaten auszuführen. Diese, durch ihre Einfachheit bestechende, Lösung wird zwar ebenfalls von kommerziellen Tools angeboten, dass die Funktion aber tatsächlich von einem Open Source Werkzeug mit geringem Entwicklungsaufwand realisiert werden kann, ist beachtlich. Alle auf einer Browser-Benutzeroberfläche basierenden Systeme sind damit mit überschaubarem Aufwand testbar.

Die Beschreibung der Anforderungen wird in Textdateien festgehalten, den sogenannten Feature Files. Diese Textdateien werden mit Hilfe der formalen Sprache Gherkin geschrieben. Gherkin gibt den Aufbau der Anforderungen vor, damit diese später von Cucumber verarbeitet und durchgeführt werden können. Zu Gherkin und Feautre Files siehe auch Kapitel 2.2.10.

Mittlerweile unterstützt Gherkin über 40 natürliche Sprachen. Das Team setzte jedoch auf Englisch, da englische Feature Files einfacher zu lesen sind als deutsche.

Im Projekt werden mit der Verwendung von Cucumber für die Testfallerstellung gleich drei Projektanforderungen gleichzeitig erfüllt, denn eines dieser Feature Files erfüllt drei Funktionen. Erstens ist das Feature File die Dokumentation für die Akzeptanztests der einzelnen User Stories. Da in Folge auch die Automatisierung darüber gesteuert wird, zwingt das System dazu, genau zu beschreiben, welche Akzeptanzkriterien erfüllt sein müssen damit ein Feature vollständig umgesetzt ist. Somit ist gleichzeitig die zweite Funktion erfüllt: der Testfall ist erstellt. Drittens können auf Basis dieser Feature Files in weiterer Folge auch die regelmäßigen, automatisierten Regressionstests durchgeführt werden (siehe dazu die beiden vorhergehenden Abschnitte 4.2 und 4.3). Da im Projekt in einem sehr zeitkritischen Projektumfeld gearbeitet wurde, war die Zeitersparnis, die mit Cucumber erzielt wurde, ein wesentlicher Erfolgsfaktor.

Die Tests werden mit Hilfe eines Webdrivers[5] direkt über die Benutzeroberfläche durchgeführt. Somit werden hiermit automatisierte Systemtests durchgeführt, die das komplette System inklusive aller Teilsysteme mittesten. Als zusätzliche Erweiterung wurde außer-

[5] Der Webdriver sorgt dafür, dass die gewünschten Aktionen in dem entsprechenden Browser in Benutzeraktionen wie Mausklicks, Texteingaben, etc. umgesetzt werden.

dem mit Capybara[6] als Middleware gearbeitet. Dadurch kann man jederzeit die Testläufe auf verschiedenen Browsern durchführen, ohne die Testscripts ändern zu müssen.

```
1  Feature: Valid users can log in
2           Invalid username/password combinations fail
3
4    Background:
5      Given an "admin" account "AdminBoris" with password "
       ↪ Password123"
6      Given an "nf\_admin" account "AdminGreek" for the National
       ↪  Federation "Greek Federation" with country code "GRE"
7      Given an "Event\_Registration" account "EventBoris" with
       ↪ password "Password123"
8
9    Scenario Outline: A valid user can log in successfully
10     Given I am on the login page
11     When I log in with "<username>" and "<password>"
       ↪ credentials
12     Then I see the member administration page
13
14     Examples: valid usernames and passwords
15     | username    | password    |
16     | AdminBoris  | Password123 |
17     | AdminGreek  | Password345 |
18
19   Scenario Outline: An invalid user cannot log in successfully
20     Given I am on the login page
21     When I log in with "<username>" and "<password>"
       ↪ credentials
22     Then I see a login error message
23
24     Examples: invalid usernames and passwords
25     | username    | password    |
26     | AdminBoris  | falsePasswd |
27     | AdminGreek  | falsePasswd |
28     | EventBoris  | Password    |
```

Listing 4.1: Feature File für Login, geschrieben in Gherkin

Der Aufbau eines Feature Files hat mehrere Teile, im ersten Teil wird mit dem Schlüsselwort *Feature* die gewünschte Funktionalität kurz beschrieben, dies dient zur Dokumentation. Danach folgen abhängig von der beschriebenen Funktionalität ein oder mehrere Szenarios. Oft werden mehrere dieser Szenarios benötigt, um die Funktionalität vollständig zu prüfen. Jedes Szenario wird mit einer kurzen Funktionsbeschreibung versehen, dies erfolgt mit den Schlüsselwörtern *Scenario Outline*.

[6] Nähere Infos unter https://rubygems.org/gems/capybara.

Die Szenarios bestehen wiederum aus drei Schritten wie in Kapitel 2.2.10 beschrieben.

Im Listing 4.1 ist ein Beispiel eines Feature Files dargestellt, welches den Login-Testfall in Gherkin beschreibt. Nach der Featurebeschreibung folgt das erste Szenario, für gültige Loginversuche: Zeile 10 legt als Vorbedingung fest, dass die Login Seite im Browser angezeigt wird. Im zweiten Schritt, in Zeile 11, wird die Aktion beschrieben, dass Benutzername und Passwort eingegeben werden, wobei die Werte für diese beiden Eingabefelder hier noch nicht festgelegt werden. Der Ergebnisschritt (Zeile 12) legt fest, welche Seite nach dem erfolgreichen Login angezeigt werden soll. In diesem Fall soll die Mitglieder-Administrationsseite (= member administration page) angezeigt werden. Im zweiten Szenario wird ein ungültiger Login-Versuch beschrieben. Zeile 20 und 21 beschreiben dieselbe Ausgangssituation und Aktion wie im vorhergehenden Szenario. Zeile 22 beschreibt als erwartetes Ergebnis eine Fehlermeldung vom System, da Benutzer und Passwort ungültig sind. Die jeweiligen zu verwendenden Testdaten sind jeweils nach der *Scenario Outline* in *Examples* angegeben.

Das Feature File wird durch ein sogenanntes „Step Definition File“ ergänzt, dem Cucumber File. In dieser Datei werden die zuvor beschriebenen Testschritte in Aktionen übersetzt. In Listing 4.2 wurde für die Umsetzung die Programmiersprache Ruby verwendet. Cucumber ist auch in anderen Programmiersprachen verfügbar, jedoch scheint die Verwendung von Ruby hier am einfachsten, da sie sehr unkompliziert zu schreiben und übersichtlich zu lesen ist.

Die Definitionen für die Testschritte des Gherkin Files (Listing 4.1) sind nun im Cucumber File (Listing 4.2) angeführt. Diese „Step Definition Files“ bestehen wieder aus drei Teilen mit den Schlüsselwörtern Given, When und Then; passend zu den Feature Files.

Given

Der erste Teil gibt an, für welche Testschritte die folgenden Anweisungen geschrieben worden sind. Im angeführten Beispiel also für die Situation „I am on the login page“.

When

Der Anweisungsteil gibt an, dass Benutzername und Passwort eingegeben werden müssen. In Listing 4.2 ist angegeben, dass diese aus der Tabelle im Feature File gelesen werden sollen. Danach wird mit *click_button submit* das Einloggen mittels Mausklick abgeschlossen.

Then

Schlussendlich wird noch überprüft, ob auf der neu aufgebauten Website der Text *Member Administration* angezeigt wird, um zu verifizieren, ob der Loginvorgang einwandfrei funktioniert hat. Werden negative Testfälle, also jene mit falschen Benutzernamen- und Passwortkombinationen, durchgeführt, wird eine Fehlermeldung erwartet, die das Wort *Failed* enthält.

Mit Hilfe dieser beiden Files können nun diverse Akzeptanztests, Testfälle und Automatisierungsscripts in einem Arbeitsschritt beschrieben und erstellt werden. Dadurch konnten im Projekt die Testaktivitäten beschleunigt und effizienter gestaltet werden.

```
1  Given /^I am on the login page$/ do
2    visit 'http://hostname/'
3  end
4
5  When /^I log in with "(.*?)" and "(.*?)" credentials$/ do |
   ↪ username, password|
6    clear_session_storage(username)
7    fill_in "input-username", :with=> username
8    fill_in "input-password", :with=> password
9    click_button "submit"
10 end
11
12 Then /^I see the member administration page$/ do
13   page.should have_selector("title", :text => "Member␣
     ↪ Administration")
14 end
15
16 Then /^I see a login error message$/ do
17   page.should have_content("Failed")
18 end
```

Listing 4.2: Step Definition File für Login, geschrieben in Cucumber

4.4.2 Testprozess unterstützende Werkzeuge

Im Projekt muss eine Softwarelösung für die Projektunterstützung eingesetzt werden, da die Flut an Informationen nur schwer bewältigt werden kann. Dies gilt sowohl für die Softwareentwicklung als auch für den Softwaretest. Darüber hinaus sind die Räumlichkeiten ungeeignet, um ein physisches Taskboard auf traditionelle Art und Weise mit Karteikarten zu bekleben. Das Projektteam arbeitet bei diversen Teilsystemen mit anderen Teams zusammen, die jedoch nicht vor Ort sind. Es soll aber eine Möglichkeit geschaffen werden, dass die Mitarbeiter anderer Teams am Prozess teilhaben können und so das kollektive Wissen erhöht und die Zusammenarbeit erleichtert wird. Somit ist eine Softwarelösung unbedingt notwendig. Hierfür wurden sämtliche Werkzeuge analysiert und auf ihre Einsetzbarkeit in diesem Umfeld überprüft.

Scrum kann nur funktionieren, wenn der Prozess durch entsprechende Tools oder Boards zur Verwaltung der Artefakte unterstützt wird. User Stories, Bugs, Burndown Charts und dergleichen sollen gut visualisiert werden, damit sie für alle Teammitglieder transparent und verständlich sind. Der Einsatz elektronischer Tools zur Unterstützung des Scrumprozesses ist jedoch in Fachkreisen sehr umstritten und immer wieder heiß diskutiert. Viele sind von der Effektivität physischer Werkzeuge und deren Flexibilität gegenüber elektronischen Tools überzeugt.

Die Vorteile vom nicht elektronischen Taskboard und den dazugehörigen Karteikarten für User Stories sind schnell erläutert: Die Verwendung von Papier und Stift ist denkbar einfach, denn die User Stories können damit leicht und schnell aktualisiert werden. Diverse potentielle, technische Hürden, wie Login, Usability und Verbindungsprobleme sind aus-

geschaltet. Weiters wird die Kommunikation gefördert, da die Teammitglieder einander beim Taskboard treffen und dabei gleich ein paar wichtige Informationen austauschen. Da Kommunikation in agilen Projekten ein großer Erfolgsfaktor ist, ist dieser Aspekt nicht zu unterschätzen. Zwei der größten Nachteile der physischen Boards sind jedoch einerseits die Wartbarkeit bei verteilten Teams und andererseits die fehlende Verfolgbarkeit und Dokumentation von Änderungen.

Es folgen kurze Beschreibungen der evaluierten Werkzeuge:

Whiteboard mit Webcam

Als einfachste Möglichkeit, diese Probleme zu umgehen, kann es sich durchaus bewähren, ein normales Whiteboard an der Wand zu verwenden und einfach eine Webcam davor aufzustellen. Damit haben auch nicht anwesende Teammitglieder einen Überblick und sind immer über den aktuellen Status des Projektes informiert. Nachteil dabei ist natürlich, dass diese Personen keine Änderungen machen können und eine natürliche Person diese Änderungen für sie einpflegen muss. Jedoch kann diese rasche „read-only“ Lösung durchaus schnelle Verbesserungen im Projektablauf bringen.

Abbildung 4.7: Whiteboard als Taskboard

Um diese beiden Aspekte auszugleichen, gibt es am Markt verschiedene elektronische Werkzeuge, die vom einfachen Taskboard bis zur agilen Komplettlösung reichen. Zu den einfachen Taskboards zählt etwa Scrumy[7], während Trello[8] ausgereifter und funktioneller – aber trotzdem kostenlos – ist.

Scrumy

Scrumy ist ein einfaches Taskboard, das als Software as a Service angeboten wird. Die Software gibt es in zwei unterschiedlichen Versionen *Free* und *Pro*.[9] Das Scrumy Taskboard bietet die klassischen Spalten ToDo, In Progress, Verify und Done. Erfahrungsgemäß benötigen Benutzer jedoch immer wieder weitere, individuelle Spalten, welche jedoch bei diesem Tool nicht hinzugefügt werden können. Scrumy bietet aber die Möglichkeit, Daten über eine API auszutauschen, was dieses Tool zu einer wertvollen Ergänzung der gesamten Entwicklungs- und Testlandschaft werden lässt. Abbildung 4.8 zeigt ein einfaches Taskboard mit drei User Stories und zugehörigen Tätigkeiten.

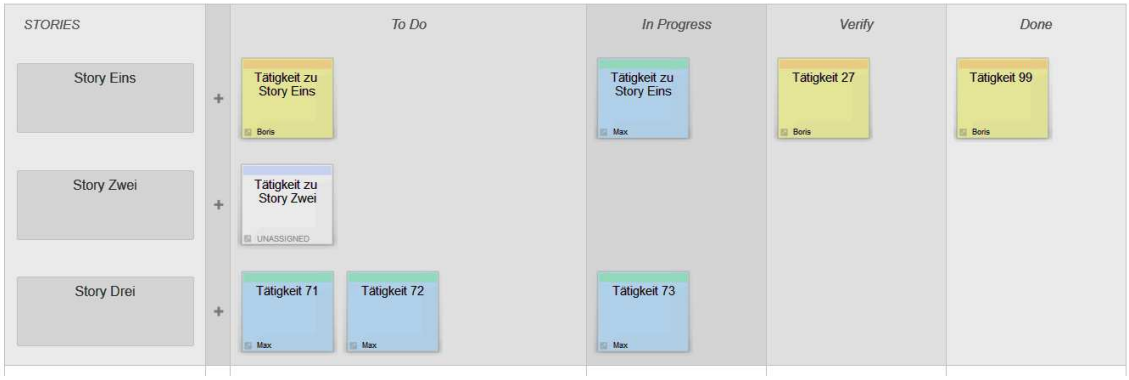

Abbildung 4.8: Scrumy Taskboard

Trello

Trello (siehe Abbildung 4.9) ist eine ausgereiftere Taskboardsoftware, die eine sehr benutzerfreundliche Oberfläche bietet. Die Funktionalität von Trello bietet weit mehr als ein herkömmliches Taskboard. So kann man – neben vielen anderen Features – beispielsweise Filter setzen, individuelle Tags vergeben, Benachrichtungen für einzelne Karten einrichten und Kommentare hinzufügen. Dieses Taskboard wird ebenfalls als Software as a Service angeboten. Trello ist derzeit kostenlos, eventuell kann es in Zukunft weitere Features geben, die kostenpflichtig werden.[10] Zum Umfang von Trello gehören ebenfalls eine Android- und iOS-App. Mit Hilfe der Apps ist es möglich, nahezu den vollen Funktionsumfang unterwegs zu nutzen.

Diese drei Taskboards (Whiteboard, Scrumy und Trello) haben eines gemeinsam: Sie bieten keine explizite Lösung für den Softwaretest an. Selbstverständlich wäre es möglich,

[7] https://scrumy.com/
[8] http://www.trello.com
[9] Informationen zu den zusätzlichen Features der Pro-Version gibt es unter http://scrumy.com/about#scrumy-pro.
[10] https://trello.com/privacy

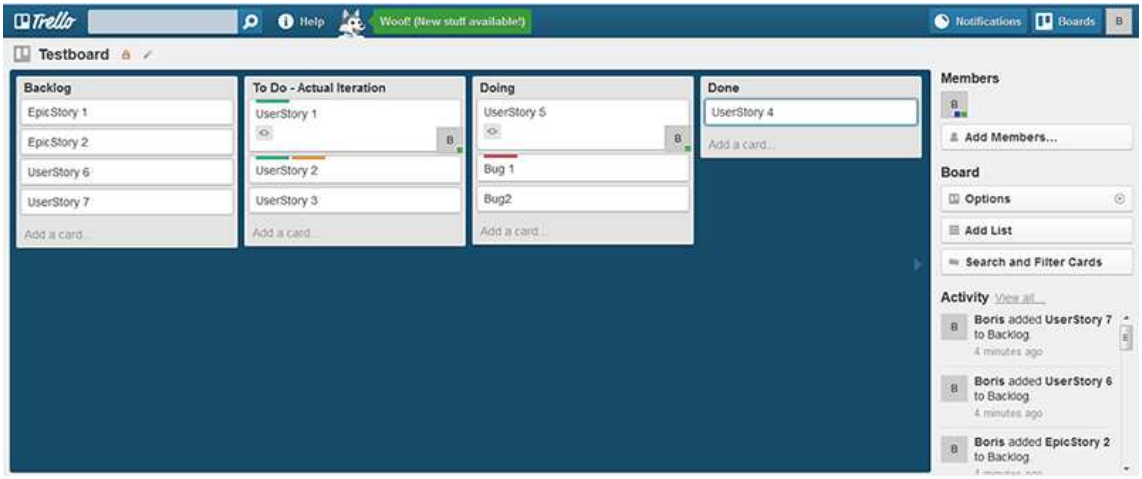

Abbildung 4.9: Trello Taskboardsoftware

für die Tasks des agilen Tests, auch eines dieser Taskboards zu verwenden. Jedoch ist zu bedenken, dass die Tasks der Entwicklung und jene des Tests unterschiedlich sind. Somit müssten Entwicklung und Test jeweils eigene Taskboards verwenden, was jedoch zu einer unerwünschten Trennung dieser beiden Bereiche führen würde. Wesentlich optimaler für beide sind Lösungen, die beides unterstützen.

Komplettlösungen zu Scrum bieten ein sehr umfangreiches Spektrum an Prozessunterstützung an. Es können alle üblichen Rollen eines Scrumteams mit verschiedenen Berechtigungen versehen, sowie alle Artefakte des Scrumprozesses abgebildet werden. Klarer Vorteil ist dabei, dass Artefakte, wie das Breakdown Chart, immer am aktuellsten Stand sind und diverse Auswertungen automatisch ausgeführt werden. Alle Rollen des Scrumteams arbeiten damit in einem gemeinsamen Werkzeug und haben somit auch einen gesamthaften Blick auf das Projekt.

VersionOne [11]

VersionOne ist eine komplette Managementlösung für Softwareentwicklungsprojekte und unterstützt den gesamten Scrum Prozess. Sie hat eine intuitive Oberfläche und ist außerdem für Teams bis zehn Benutzer als Software as a Service kostenlos. In der freien Version sind alle grundlegenden Auswertungen vorhanden. Weiters werden über 40 Konnektoren zu diversen Fehlermanagementwerkzeugen, Entwicklungsumgebungen, Quellcodeverwaltungen und Ähnlichem geboten. VersionOne arbeitet mit drei übersichtlichen Boards: einem Storyboard für die User Stories, einem Taskboard für die einzelnen Tasks und einem Testboard, das ausschließlich für die Tests des aktuellen Sprints vorgesehen ist. VersionOne unterstützt den Test mit einem Modul für das Testmanagement, welches Hilfe bei der für die Testfallerstellung bietet. Ebenso gibt es Unterstützung beim Erstellen und Verwalten von Akzeptanztests und Regressionstests. Jedoch ist Letzteres nur in der umfangreichsten Edition enthalten.

[11] http://www.versionone.com/

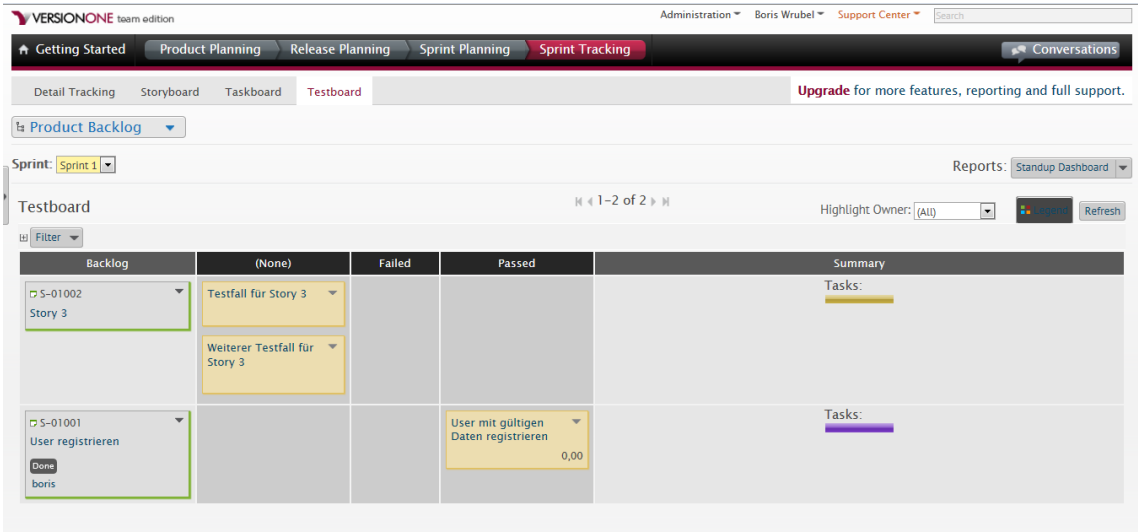

Abbildung 4.10: VersionOne Testboard

GreenHopper [12]

GreenHopper ist eine Erweiterung von JIRA, eines der wohl bekanntesten Ticketsysteme. Wie sämtliche Produkte von Atlassian sind diese als Software as a Service verfügbar, alternativ kann man einen eigenen Server betreiben. Bei der Serviceversion sind Nutzungsgebühren fällig, wobei für die ersten zehn Benutzer insgesamt lediglich $ 20 bezahlt werden müssen (für GreenHopper und JIRA Lizenzen pro Monat oder auch als einmaligen Betrag). Für bis zu 25 Nutzer sind jedoch bereits $ 1800 für den selben Umfang fällig. Die GreenHopper / JIRA Lösung bietet eine solide Projektverwaltung mit agilen Erweiterungen. Es können elektronische Whiteboards erstellt und zu einem hohen Grad an die Bedürfnisse des Projektteams angepasst werden. Weiters stehen zahlreiche Auswertungen zur Verfügung.

JIRA kann mit zahlreichen weiteren Modulen des Herstellers oder von Drittanbietern erweitert werden. Geron vom Forbes Magazin unterstreicht in seinem Artikel [29] die hohe Benutzerfreundlichkeit von Atlassian Werkzeugen, jedoch bietet JIRA keine spezielle Unterstützung für den Softwaretest. Sämtliche Anforderungen an Testprozesse müssen selbstständig in JIRA konfiguriert und angepasst werden. Dies ist zwar grundsätzlich möglich, da JIRA in hohem Grade konfigurierbar ist, jedoch wären diese Anpassungen sehr tiefgreifend und somit auch aufwändig.

Rally [13]

Rally ist ein sehr umfangreiches Managementtool, das nicht nur für agile Projekte geeignet ist. Es bietet zahlreiche Möglichkeiten, um den Arbeitsbereich individuell zu konfigurieren. So kann beispielsweise jedes Projektmitglied seine eigene Benutzeroberfläche konfigurieren. Es können auch sehr leicht und unkompliziert externe Dashboards eingebunden werden. Das Werkzeug liefert eine funktionale Benutzeroberfläche, die hochgradig konfiguriert werden kann. Ähnlich wie bei VersionOne sind Projekte bis zehn Benutzer kostenfrei.

[12] https://www.atlassian.com/software/greenhopper/overview
[13] http://www.rallydev.com/agile-alm-platform-products

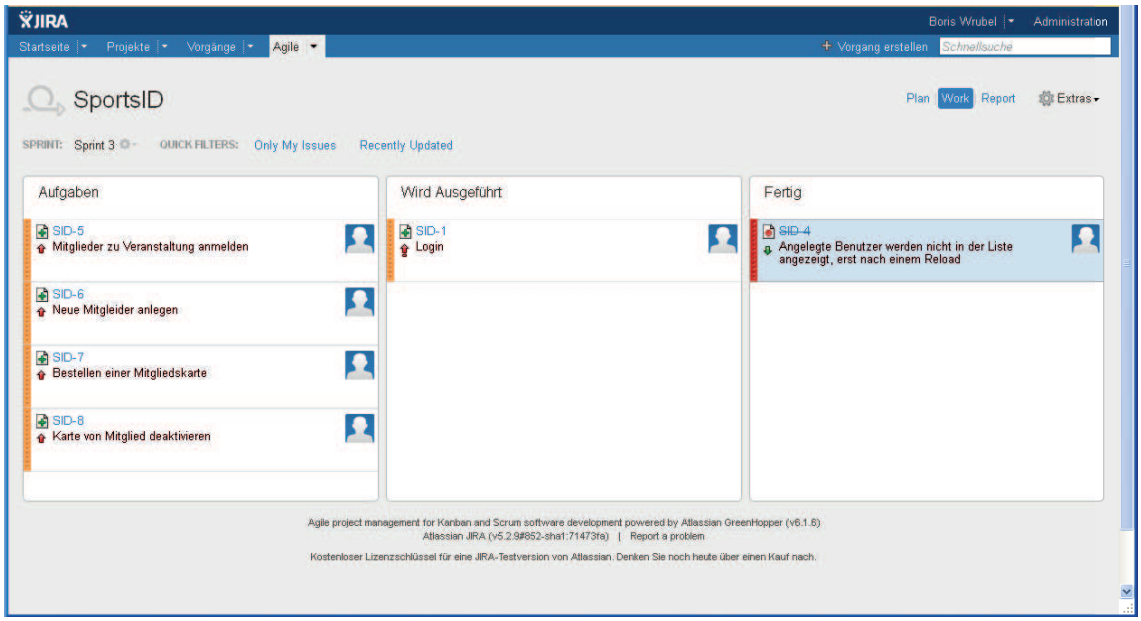

Abbildung 4.11: GreenHopper Taskboard

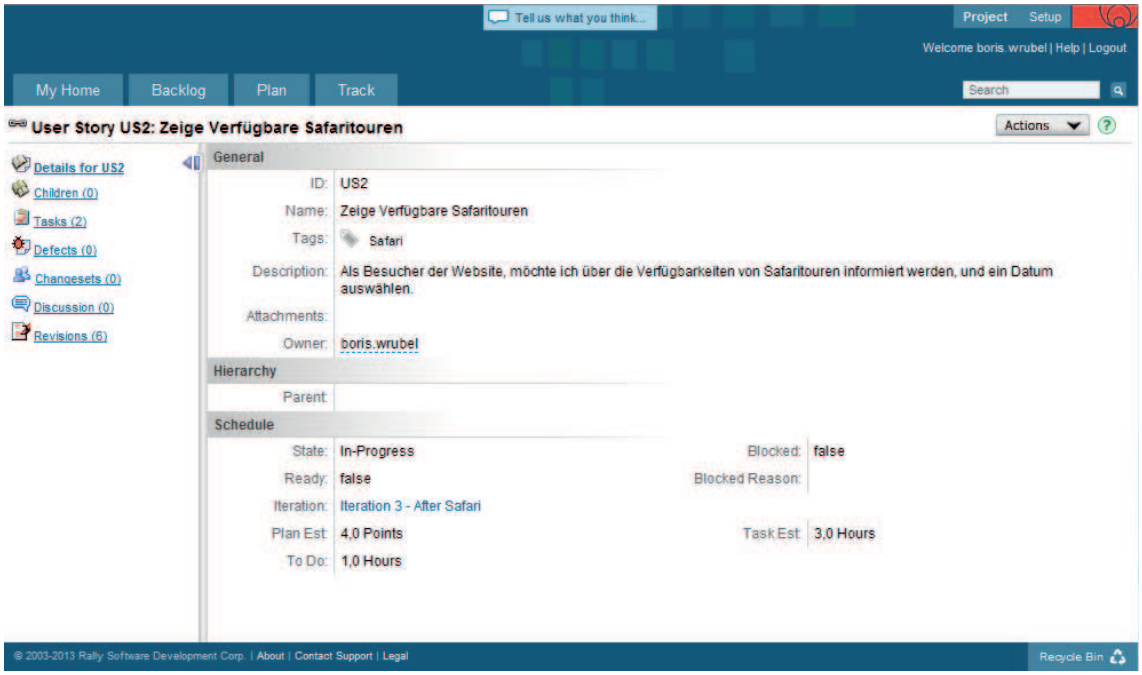

Abbildung 4.12: Rally User Story

Kunagi [14]

Kunagi ist ein Open Source Tool, das als Java Webserver Applikation ausgeliefert wird. Somit ist man für den Betrieb und die Datensicherheit selbst verantwortlich. Im Gegensatz zu VersionOne oder Rally werden die Daten nicht ausgelagert, was für viele Unternehmen ein wichtiges Kriterium ist. Die Software ist sehr gut an den Scrum Prozess angelehnt und erweitert diesen laut eigenen Angaben mit Best Practices. Einige Schritte des Scrum Prozesses, wie das Scrum Poker, können mit Hilfe von Kunagi auch von verteilten Teams wahrgenommen werden. Die explizite Testunterstüzung ist sehr einfach, für jede User Story kann in einem extra Bereich relevante Information für den Test abgelegt und referenziert werden. Für umfangreiche Testaufgaben und Testautomatisierungen ist dies jedoch ungenügend.

Abbildung 4.13: Kunagi Burndown Chart

Bei der Entscheidung für eines dieser Tools sind klarer Weise die Kosten ein wichtiger Faktor. Viele der genannten Tools werden kostenlos für einen kleineren Nutzerkreis angeboten. Sobald die Teamgröße jedoch wächst, können unter Umständen rasch erhebliche Kosten anfallen. Ein weiterer entscheidender Faktor kann die Frage der Wartbarkeit darstellen. Die meisten der genannten Tool werden zwar als Software as a Service angeboten, bei den anderen Lösungen fällt der Wartungsaufwand aber intern an.

Auch im hier vorgestellten Projekt waren die Kosten ein entscheidendes Kriterium für die Anschaffung eines Werkzeugs. In diesem Fall wurde Kunagi als die optimale Lösung für das Projektumfeld ausgewählt. Es ist ein Open Source Werkzeug und somit gibt es keine Anschaffungskosten, lediglich der Betrieb der Software muss gewährleistet werden. Die internen Kosten für den Betrieb und die Betreuung einer Open Source Software, die als Webapplikation läuft, mussten kalkuliert und mit den Kosten von anderen Lösungen verglichen werden.

[14] http://www.kunagi.org

Die Anzahl der Projektmitglieder überschreitet zum Zeitpunkt der Entscheidung nicht zehn Personen. Jedoch werden im Laufe des Projektes mehr als zehn Personen aktiv teilnehmen und dies würde bei einigen der genannten Werkzeuge die Kosten rasant ansteigen lassen.

Ein weiterer ausschlaggebender Faktor für die Entscheidung im Projekt war die Datensicherheit, die bei einem Betrieb im hauseigenen Rechenzentrum mit der höchstmöglichen Sorgfalt gewährleistet ist. Bei Serviceprovidern kann dies, entgegen aller Versicherungen der Software as a Service Anbieter, nicht vollständig sichergestellt sein. Somit hat sich das Feld der Kandidaten rasch auf Kunagi reduziert, welches nach nur einem Tag nach der Entscheidung als Werkzeug für die Prozessunterstützung im Projektteam ausgerollt wurde.

So mächtig sowohl Cucumber als auch Kunagi auf ihren Gebieten sind, beide bieten jedoch kein Testmanagementwerkzeug an. Dieses ist aber von enormer Wichtigkeit, um nicht den Gesamtüberblick zu verlieren. Denn in jedem Projekt gibt es Testfälle, die nicht automatisierbar sind (= Manuelle Tests) beziehungsweise Tests, die nicht unmittelbar mit einer Funktionalität zusammenhängen (= Nicht funktionale Tests, siehe dazu 2.1.3.2). Diese Testaufgaben werden stets parallel zur Umsetzung und dem Test von User Stories durchgeführt[15] und müssen ebenso geplant, überwacht und auf ihre Vollständigkeit geprüft werden.

Ein Testmanagementwerkzeug übernimmt jedoch nicht nur die Überwachung von manuellen und nicht funktionalen Tests. Darüber hinaus können optimaler Weise automatisierte Testfälle direkt aus dem Testmanagementwerkzeug angesteuert und gestartet werden, während Ergebnisse dieser Testläufe vom Testautomaten ebenso rückgemeldet und dokumentiert werden.

Für das zuvor beschriebene Projekt ist es außerdem wichtig, die Zuordnung von Testfällen zu User Stories zu kennen, zu wissen ob diese bereits automatisiert sind, sowie in weiterer Folge den aktuellen Status dieser Testfälle abrufen zu können. In Kunagi ist jedoch nur ein Textfeld für sämtliche Testaktivitäten zu einer User Story vorgesehen. Dies ist für die Testaktivitäten in dem Projekt jedoch nicht ausreichend und außerdem mühsam zu warten.

Um in diesem Projekt alle diese Anforderungen in einem Werkzeug zu vereinen, empfiehlt sich für das Testmanagement SquashTM[16]. Das Werkzeug wurde zu einem früheren Zeitpunkt bereits evaluiert und steht daher im Unternehmen schon zur Verfügung. SquashTM ist ein Open Source Werkzeug für das Anforderungs- und Testmanagement. In Verbindung mit SquashTA[17] ist es möglich, auch automatisierte Scripts einzubinden und die zuvor genannten Anforderungen für die Einbindung der automatisierten Tests werden vollends erfüllt.

[15] Siehe dazu auch Abbildung 4.3.

[16] www.squashtest.org

[17] SquashTM ist ein Testmanagementwerkzeug, während SquashTA (TA steht hierbei für „Test Automation“) ein Testmanagement Controller ist. Es liefert die Schnittstelle zwischen SquashTM und dem zu testenden System. Details zu SquashTA sind zu finden unter http://www.squashtest.org/index.php/en/squash-ta/squash-ta-overview.

In weiterer Folge kann die Application Programming Interface (deutsch: Programmierschnittstelle) (API) von SquashTM genutzt werden, um die Ergebnisse der automatisierten Durchläufe von Cucumber in SquashTM zu erfassen. Somit ist ebenfalls eine vollständige Testdokumentation in SquashTM vorhanden. Die API von SquashTM ist jedoch noch nicht vollständig implementiert, daher wird in der vorliegenden Arbeit nicht näher auf die Lösung eingegangen. Eine vollständige Implementierung und Dokumentation der API wird für Ende 2013 erwartet.

Auch für die Testautomatisierung gibt es zahlreiche Werkzeuge, die sorgfältig evaluiert werden müssen, um sich optimal in den Testprozess einzugliedern. In Kapitel 4.2 werden diese Testautomatisierungswerkzeuge und ihr Einsatz im Detail erläutert.

Die Praxis hat gezeigt, dass die Teammitglieder die Testwerkzeuge schnell annehmen und den Umstieg von einem alten System – wie beispielsweise den traditionellen Klebezetteln – problemlos vollziehen, wenn die neuen Werkzeuge sie in ihrer Arbeit optimal unterstützen und nicht zusätzliche Arbeit schaffen. Wenn dies dann noch mit einer hohen Benutzerfreundlichkeit der Werkzeuge einhergeht, verläuft die Einführung dieser meist auch mit einer begleitenden signifikanten Steigerung der Effizienz und des Wissenstransfers innerhalb des Teams.

4.5 Rollenbild des agilen Testers

Die Abläufe und Anforderungen in agilen Projekten sind andere als in Projekten in denen nach klassischen Vorgehensmodellen getestet wird. Somit ist es eine logische Konsequenz, dass sich in agilen Projekten auch das Rollenbild des Testers verändert.

Geht man nach den Definitionen von Scrum [70] oder XP [7], ist ein dedizierter Tester nicht einmal mehr vorgesehen, weil die Entwickler auch testen. Wenn man jedoch ein wirklich agiles Testprojekt umsetzen und leben will, gibt es die Person des agilen Testers sehr wohl und sie muss weit mehr Fähigkeiten mitbringen, als konventionelle Tester, um erfolgreich arbeiten zu können. Hellerer bestätigt beispielsweise in [38], dass die Aufgaben des Testers im Scrum Umfeld weitaus verantwortungsvoller sind, als in einem klassischen Vorgehensmodell.

Erfahrung ist für jeden guten Tester ein großer Pluspunkt. Für einen agilen Tester ist diese praktische Erfahrung jedoch ein wirklicher Schlüsselfaktor. Dabei ist es weniger wichtig, ob die Projekte agil oder nach klassischen Vorgehensmodellen durchgeführt worden sind. Der entscheidende Punkt ist, dass ein erfahrener Tester unterschiedliche Testmethoden kennt und weiß, wie und wann diese eingesetzt werden sollten. Diese Skills ermöglichen es ihm, sehr flexibel auf Gegebenheiten reagieren zu können und passende Methoden für die jeweilige Situation zu finden. Auch das erworbene Wissen verschiedener Technologien und das sogenannte Gespür für das Auffinden von Fehlern sind weitere Pluspunkte eines erfahrenen Testers, denn dieser hat wesentlich mehr mit der Programmierung zu tun, als es ein klassischer Tester hat.

Jimmink beschreibt in seinem Artikel in [49], dass sich in der Vergangenheit das Recruiting in erster Linie auf die technischen und analytischen Fähigkeiten der Kandidaten konzentriert hat. Im agilen Test sind die Softskills der Kandidaten jedoch entscheidend für den Erfolg des Projektes. Denn effektive Interaktion mit internen und externen Be-

teiligten ist ein allgegenwärtiger Bestandteil des Projektalltags. Für einen kommunikativ begabten Tester ist es außerdem leichter, sich technische Fähigkeiten im Laufe des Projektes anzueignen, als für einen technisch hoch begabten und erfahrenen Entwickler, zu erlernen, wie er effektiv und gut kommuniziert.

Eine dieser kommunikativen Aufgaben des Testers wird von Crispin und Gregory in ihrem Buch [19] beschrieben. Sie weisen darauf hin, dass Tester sehr oft eine Schnittstellenfunktion zwischen Programmierern und dem Fachbereich oder auch dem Kunden sind. Dadurch nimmt der Tester auch Aufgaben des Kundenbetreuers oder Key Account Managers wahr, der dafür natürlich auf seine sozialen und kommunikativen Fähigkeiten zurückgreift.

Hellerer weist in [38] auf einen weiteren Grund hin, warum nicht jeder Tester für ein agiles Umfeld geeignet ist. Denn es gibt durchaus Personen, die das tägliche Scrum Meeting als eine unangenehme Art der Überwachung oder Kontrolle empfinden. Man braucht als Tester – und auch Entwickler – ein gesundes Maß an Selbstbewusstsein und Teamverständnis, um eine solche Art der Abstimmung als wertvolles und produktives Element der täglichen Arbeit anzuerkennen.

Gleichzeitig darf man nicht außer Acht lassen, dass ein agiler Tester ebenso ein hohes technisches Verständnis mitbringen muss. Im klassischen Test definieren das Requirements Engineering und die Analyse die Funktionalität, welche der Tester dann überprüft. Im agilen Test definiert der Product Owner die User Story, welche durch den Tester dann so verfeinert wird, dass Product Owner, Entwickler und Tester ein gemeinsames Verständnis der User Story haben. Dafür ist neben ganzheitlichem Denken selbstverständlich auch technisches Verständnis gefragt.

Beispielsweise gibt es kaum noch isolierte Systeme, also Systeme, die keine Schnittstellen zu anderen Systemen aufweisen. Es ist somit für den Tester immer mehr Integrationsarbeit notwendig, da er die Fehler in den Schnittstellen identifizieren muss. Sneed ist in seiner Keynote zur iqnite 2012 [74] sogar der Meinung, dass die Entwicklungsarbeit der Zukunft hauptsächlich das Kombinieren und Anpassen von Standardkomponenten umfassen wird. Umso mehr fungieren Tester als Integratoren dieser Softwarestandardkomponenten, deren Zusammenspiel sie auch technisch verstehen müssen.

Auch das Rollenbild des Testmanagers ist in agilen Teams vertreten, jedoch wird diese Rolle sowohl in der Literatur, als auch in der Testpraxis, nicht zwingend von einer Person eingenommen. Hellerer ist beispielsweise der Meinung, dass ein expliziter Testmanager nur bedingt notwendig ist, vielmehr müssen agile Tester im Team teilweise Aufgaben des Testmanagers übernehmen. Sind die Testaufgaben jedoch so groß, dass das Team diese nicht eigenständig koordinieren kann, ist es sinnvoll einen Testkoordinator einzusetzen. Vor allem, wenn es mehrere Scrum Teams gibt, auf die Tester verteilt sind, müssen diese koordiniert werden.

Zusammengefasst setzt sich das ideale Rollenbild eines agilen Testers aus Praxis, technischem Know-how und Softskills wie folgt zusammen:

Testmethoden flexibel einsetzen
Er bringt Praxis im Testbereich mit und kann auf Erfahrungen mit diversen Testmethoden zurückgreifen und diese flexibel anwenden.

Testmanagement Skills
Weiters hat er Erfahrung in der Testplanung und -koordination, um gegebenenfalls Testmanageraufgaben übernehmen zu können.

Softwareentwicklung Basics
Der Tester sollte zumindest grundlegende Fähigkeiten in der Softwareentwicklung mitbringen und versteht Scriptsprachen, da er diese immer wieder bei diversen Automatisierungstools zum Einsatz bringen wird.

Blick für das große Ganze
Um den Überblick zu wahren und die User Stories möglichst realitätsnah zu testen, braucht er darüber hinaus ein ausgeprägtes ganzheitliches Denken und Qualitätsbewusstsein.

Teambewusstsein
Der ideale agile Softwaretester ist außerdem ein ausgeprägter Teamdenker, der gemeinsam mit anderen nach einem Ziel strebt und nicht als Einzelkämpfer auftritt. Gleichzeitig übernimmt er gerne Verantwortung und ist sich seiner Wichtigkeit im Team bewusst.

Feedback- und Kritikfähigkeit
Der agile Tester sieht das tägliche Reporting als selbstverständlich an und ist daran interessiert, auch im Dialog Feedback zu geben, während er ebenso kritikfähig ist.

Kommunikationsfähigkeit
Man kann gar nicht genug betonen, dass im agilen Umfeld ein Skill ganz weit oben steht: Kommunikationsfähigkeit. Nur mit funktionierender Kommunikation im Testteam sowie zum restlichen Projektteam und zu den Auftraggebern kann agiler Test überhaupt implementiert werden.

5 Empfehlungen für agiles Testvorgehen

Aus der Analyse des konkreten Fallbeispiels konnten praktische Lösungsansätze erarbeitet werden, welche im Projekt zu mehr Akzeptanz und Effizienz geführt haben. Die Erkenntnisse aus diesem Projekt zusammen mit weiteren Erfahrungen aus der Praxis ergeben folgende Empfehlungen für agiles Testvorgehen.

Agiler Testprozess
: Der agile Testprozess muss sich nahtlos in agile Entwicklungsmodelle eingliedern lassen. Dabei ist wichtig, dass der Test zeitgleich mit der Entwicklung startet und in Symbiose mit der Entwicklung durchgeführt wird. Agiles Testen ist mehr als nur das starre Durchführen von Testfällen, die zuvor aus Anforderungen abgeleitet wurden; vielmehr ist es ein umfassender Qualitätssicherungsprozess. Im agilen Testprozess sind Testautomatisierung und eine Einbindung in Continuous Integration unverzichtbar.

Testautomatisierung im agilen Umfeld
: Testautomatisierung hat im agilen Umfeld eine noch größere Bedeutung bekommen als bisher. Denn die Testfälle müssen öfter wiederholt werden und auch das Feedback muss rascher verfügbar sein. Daher empfiehlt sich ein modularer Aufbau der Testfälle, die Arbeit mit einem Continuous Integrationserver sowie eine Zusammenstellung nach der Testautomatisierungspyramide. Ein Statusbericht sollte ebenso Teil der Automatisierung sein. Dabei sollte man nicht vergessen: Es gibt immer noch Anwendungsfälle für den manuellen Test.

Continuous Integration im agilen Test
: Der agile Softwaretest macht sich den Prozess der Continuous Integration zunutze und integriert Akzeptanztests, die im Bereich des Softwaretests liegen, in die vorhandene Infrastruktur. Continuous Integration ist die Basis für schnelles Feedback und ist daher auch für den Test ein Schlüsselfaktor. Zumindest einmal täglich müssen automatisierte Testläufe auf der aktuellen Programmcodebasis durchgeführt werden, um den Status für alle Beteiligten ersichtlich zu machen und Probleme aufzuzeigen.

Die Wahl der Werkzeuge
: Jedes Projekt hat seine individuellen Anforderungen und Parameter, die nach unterschiedlichen Werkzeugen verlangen. Die Evaluierung der Werkzeuge, die den agilen Testprozess unterstützen sollen, ist ein wesentlicher Teil des Testvorgehens, da die Entscheidung der Umsetzung und Einführung in der Praxis sowohl wirtschaftliche als auch teamrelevante Auswirkungen hat. Denn die Tools sollen immer die agilen Teams bei ihrer Arbeit unterstützten und keine zusätzliche Arbeit durch deren Anwendung schaffen.

Der agile Tester

Die Rolle des Testers ist im agilen Umfeld wesentlich verantwortungsvoller. Er wird Schnittstelle zwischen Entwicklern und Product Owner und hat dabei vor allem kommunikative Fähigkeiten mitzubringen. Außerdem zeichnet er sich sowohl durch praktische Erfahrung als auch durch technisches Know-how aus. Und er ist ein ausgesprochener Teamplayer.

All diese Schlüsselfaktoren haben immer zum Ziel, dass sich das agile Testvorgehen nahtlos in die Abläufe der agilen Entwicklung eingliedert. Der agile Softwaretest ist als Teil des agilen Entwicklungsprozesses zu verstehen und nicht als Tätigkeit, die dem Programmieren nachgelagert ist. Die Zusammenarbeit zwischen diesen beteiligten Teams wird sich dadurch ebenso grundlegend wandeln, wie es der Testprozess selbst getan hat. Denn durch die neuen Aufgaben ergeben sich neue Verantwortungen und neue Synergien zwischen den Teams, welche schlussendlich die Qualität des Softwareergebnisses wesentlich beeinflussen werden.

6 Zusammenfassung

Die vorliegende Arbeit zeigt, dass ein agiles Entwicklungsprojekt ein ebenso agiles Testvorgehen verlangt. Anhand eines konkreten Anwendungsfalles werden die Probleme illustriert, die der falsche Umgang mit dem Softwaretest in einem agilen Projekt mit sich bringt. Gleichzeitig wird ein praktischer Lösungsansatz vorgestellt, wie der agile Test ein solches Projekt optimal unterstützen kann.

Um das Zusammenspiel von Entwicklung und Test in einem teilprojektreichen Technologieintegrationsprojekt zu verdeutlichen, wird in Kapitel 2 zunächst ein kompaktes aber möglichst vollständiges Grundwissen zu Softwaretest, Testautomatisierung und agiler Softwareentwicklung vorgestellt. Es werden der fundamentale Testprozess und die verschiedenen Teststufen erläutert. Weiters wird auf die vier Testarten nach ISTQB eingegangen, genauso wie auf die bereits etablierten Testmethoden. Relevante Vorgehensweisen zur Testautomatisierung werden ebenso vorgestellt. Da es im Praxisteil um ein agiles Softwareprojekt geht, werden in den Grundlagen auch relevante Softwareentwicklungsmodelle behandelt. Basiswissen zur Near Field Communication bildet den Abschluss des Theorieteils, um das technologische Umfeld des Projektes abzustecken.

In Kapitel 3 folgt die Beschreibung des Softwareentwicklungsprojektes. Es handelt sich hierbei um die Implementierung eines weltweit einsetzbaren Systems zu Registrierung von Mitgliedern bei Sportveranstaltungen. In einem komplexen organisatorischen Projektumfeld wird auf agile Softwareentwicklung gesetzt, gleichzeitig werden jedoch im Softwaretest klassische Methoden eingesetzt. Welche Probleme dies für die Ergebnisse und den Fortschritt des Projektes, sowie für die Teamkonstellation mit sich bringt, wird in diesem Kapitel illustriert.

Der Lösungsansatz, welcher in diesem Projekt in Folge implementiert und erfolgreich weitergeführt wurde, ist in Kapitel 4 im Detail beschrieben. Es handelt sich hierbei nicht nur um ein Fallbeispiel, sondern er geht vielmehr um ein in der Praxis erprobtes Grundgerüst des agilen Softwaretests. Die konkreten Lösungsansätze, welche in diesem Praxisbeispiel sorgfältig ausgewählt, eingeführt und bewertet wurden, sind die Basis für einen beispielhaften, und durchaus auch auf ähnliche Projekte übertragbaren agilen Testansatz. Dieser umfasst einen agilen Testprozess, Empfehlungen für die Testautomatisierung sowie Erläuterungen zur Continuous Integration im agilen Test. Weiters sind konkrete Werkzeugempfehlungen für die Akzeptanztests sowie Empfehlungen für die Unterstützung des gesamten agilen Testprozesses und Testmangements Teil dieses Lösungsansatzes. Als einen der wichtigsten Faktoren des hier vorgestellten agilen Testansatzes geht das Kapitel selbstverständlich auf die neuen Aufgaben des agilen Testers und das damit einhergehende gewandelte Rollenbild ein.

Literatur

Wissenschaftliche Literatur

[2] James Bach. “Agile Test Automation”. In: *URL: http://satisfice. com/articles/agileauto-paper. pdf* (2003).

[3] James Bach. *Exploratory Testing Explained*. James Bach Blog. 2003. URL: http://www.satisfice.com/articles/et-article.pdf.

[4] H. Balzert. *Lehrbuch Der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Lehrbücher der Informatik. Spektrum Akademischer Verlag GmbH, 2011. ISBN: 9783827422460. URL: http://books.google.at/books?id=1eVadwTvV_AC.

[5] H. Balzert u. a. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Lehrbücher der Informatik. Spektrum Akademischer Verlag, 2009. ISBN: 9783827417053. URL: http://books.google.at/books?id=vmfIb9R2QikC.

[6] G. Bath und J. McKay. *Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst: Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard*. Dpunkt.Verlag GmbH, 2011. ISBN: 9783898647359. URL: http://books.google.at/books?id=naDKXwAACAAJ.

[7] K. Beck. *Extreme Programming*. Programmer’s choice. Addison-Wesley, 2003. ISBN: 9783827321398. URL: http://books.google.at/books?id=79dSpPDQdLYC.

[8] K. Beck. *Test Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003. ISBN: 9780321146533. URL: http://books.google.at/books?id=gFgnde_vwMAC.

[10] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990. ISBN: 9781850328803. URL: http://books.google.at/books?id=jjXTHQAACAAJ.

[11] Barry W Boehm. “Guidelines for verifying and validating software requirements and design specifications”. In: *Proc. European Conf. Applied Information Technology (IFIP 79)*. 1979, S. 711–719.

[12] C. Bunse und A.V. Knethen. *Vorgehensmodelle kompakt*. It Kompakt. Spektrum Akademischer Verlag, 2008. ISBN: 9783827419507. URL: http://books.google.at/books?id=n9V4LgAACAAJ.

[14] P. Coad, E. Lefebvre und J. De Luca. *Java modeling in color with UML: enterprise components and process*. Java Series. Prentice Hall PTR, 1999. ISBN: 9780130115102. URL: http://books.google.at/books?id=fo0_AQAAIAAJ.

[15] A. Cockburn. *Agile Software Development: The Cooperative Game*. The Agile Software Development Series. Pearson Education, 2006. ISBN: 9780321482754. URL: http://books.google.at/books?id=i39yimbrzh4C.

[16] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 2004. ISBN: 0201699478.

[17] M. Cohn. *User Stories;für die agile Software-Entwicklung mit Scrum, XP u.a.* mitp-Verlag, 2010. ISBN: 9783826658983. URL: http://books.google.at/books?id=tkVSCmAu2RIC.

[18] J. Coldewey. "Multi Kulti Ein Üerblick über die agile Entwicklung". In: *Objekt Spectrum* 1 (2002), S. 69–76.

[19] L. Crispin und J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley signature series. Addison-Wesley, 2009. ISBN: 9780321534460. URL: http://books.google.com/books?id=68_lhPvoKS8C.

[22] M. E. Fagan. "Design and code inspections to reduce errors in program development". In: *IBM Systems Journal* 15.3 (1976), S. 182–211. ISSN: 0018-8670. DOI: 10.1147/sj.153.0182.

[23] G. Ferrari und D. Laverde. *Programming LEGO Mindstorms with Java*. Syngress Media, 2002. ISBN: 9781928994558. URL: http://books.google.at/books?id=sfw0ZRsyf3AC.

[24] Thomas Finke und Harald Kelter. "Radio frequency identification–Abhörmöglichkeiten der Kommunikation zwischen Lesegerät und Transponder am Beispiel eines ISO14443-Systems". In: *Online verfügbar unter http://www. bsi. de/fachthem/rfid/Abh_RFID. pdf* (2007).

[25] K. Finkenzeller. *RFID-handbuch*. Hanser, Carl, 2008. ISBN: 9783446412002. URL: http://books.google.at/books?id=49HTBDrfqFUC.

[27] K. Frühauf, J. Ludewig und H. Sandmayr. *Software-Prüfung: Eine Anleitung zum Test und zur Inspektion*. vdf Lehrbuch. vdf, Hochschulverlag an der ETH Zürich, 2007. ISBN: 9783728130594. URL: http://books.google.at/books?id=YJIMPiSbzZcC.

[28] C. Gernert. *Agiles Projektmanagement: Risikogesteuerte Softwareentwicklung*. Hanser, 2003. ISBN: 9783446219953. URL: http://books.google.at/books?id=283impnkMR4C.

[31] T. Gilb, D. Graham und S. Finzi. *Software inspection*. Addison-Wesley, 1993. ISBN: 9780201631814. URL: http://books.google.at/books?id=GO11btO0-vIC.

[32] Boris Gloger. *Scrum : Produkte zuverlässig und schnell entwickeln*. 3., aktualisierte Aufl. München: Hanser, 2011. ISBN: 978-3-446-42524-8.

[33] Dorothy Graham und Mark Fewster. *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison-Wesley Professional, 2012. ISBN: 0321754069.

[34] Thomas Grechenig u. a. *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium. Pearson Studium, 2010. ISBN: 9783868940077.

[35] Matthias Grochtmann und Klaus Grimm. "Classification trees for partition testing". In: *Software Testing, Verification and Reliability* 3.2 (1993), S. 63–82. URL: http://onlinelibrary.wiley.com/doi/10.1002/stvr.4370030203/abstract.

[36] U. Hehn. "Was bringt und das Neue Certified Tester Advanced Level 2012?" In: *SQ Magazin* 26 (2013), S. 16–18.

[37] L.J. Heinrich, A. Heinzl und F. Roithmayr. *Wirtschaftsinformatik-Lexikon: Mit etwa 3500 Stichwörtern und 2500 Verweisstichwörtern*. Oldenbourg Wissensch.Vlg, 2004. ISBN: 9783486275407. URL: http://books.google.at/books?id=0EOHQJBR3SkC.

[38] Heinz Hellerer. *Soft Skills für Softwaretester und Testmanager*. Dpunkt.Verlag GmbH, 2012. ISBN: 3898648311.

[39] Mike Hendry. *Multi-application Smart Cards*. Cambridge University Press, 2007. ISBN: 9781139464901. URL: http://books.google.at/books?id=ieAkyyRnJrwC.

[40] Dirk Henrici. *RFID Security and Privacy: Concepts, Protocols, and Architectures (Lecture Notes in Electrical Engineering)*. Springer, 2008. ISBN: 3540790756.

[41] M. Howard und D. LeBlanc. *24 Deadly Sins of Software Security (ebook)*. McGraw-Hill Education, 2009. ISBN: 9780071626767. URL: http://books.google.at/books?id=HtPr8kbMknQC.

[42] W.E. Howden. *Functional program testing and analysis*. McGraw-Hill series in software engineering and technology. McGraw-Hill, 1987. ISBN: 9780070305502. URL: http://books.google.at/books?id=HM4mAAAAMAAJ.

[43] P. Hruschka, C. Rupp und G. Starke. *Agility kompakt: Tipps f ür erfolgreiche Systementwicklung*. It Kompakt. Spektrum Akademischer Verlag, 2009. ISBN: 9783827420923. URL: http://books.google.at/books?id=AOUdJ89_F6AC.

[44] ZhanWei Hui u. a. "Software security testing based on typical SSD:A case study". In: *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*. Bd. 2. Aug. 2010, S. V2–312 –V2–316. DOI: 10.1109/ICACTE.2010.5579101.

[45] IEEE. "IEEE Standard for Software Unit Testing". In: *ANSI/IEEE Std 1008-1987* (1987), S. i –22.

[47] ISO/IEC. *DIN ISO 9241 Ergonomie der Mensch-System-Interaktion*. ISO.

[48] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[49] Eric Jimmink. "Testing in an organisation going agile". In: *Agile Record* 1 (Jan. 2010), S. 26–28. URL: www.agilerecord.com.

[50] S. Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, 2002. ISBN: 9780201745702. URL: http://books.google.at/books?id=6Yu7s6XOV8cC.

[51] Tilo Linz. *Testen in Scrum-Projekten - Leitfaden für Softwarequalität in der agilen Welt*. dpunkt, 2013.

[53] Thomas J. McCabe. "A complexity measure". In: *Software Engineering, IEEE Transactions on* 4 (1976), S. 308–320.

[54] S. Medhat. "Agile Testing Between Theory and Practice". In: *Agile Record* (Juli 2010). URL: http://www.agilerecord.com.

[55] Gerard Meszaros, ShaunM. Smith und Jennitta Andrea. "The Test Automation Manifesto". In: *Extreme Programming and Agile Methods - XP/Agile Universe 2003*. Hrsg. von Frank Maurer und Don Wells. Bd. 2753. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 73–81. ISBN: 978-3-540-40662-4. DOI: 10.1007/978-3-540-45122-8_9. URL: http://dx.doi.org/10.1007/978-3-540-45122-8_9.

[57] G.J. Myers. *Methodisches Testen von Programmen*. Oldenbourg, 2001. ISBN: 9783486256345. URL: http://books.google.at/books?id=srqkL75liPkC.

[61] Sachar Paulus. *Basiswissen Sichere Software*. Dpunkt.Verlag GmbH, 2011. ISBN: 3898647269.

[62] Mauro Pezze und Michal Young. *Software testen und analysieren*. Oldenbourg Wissensch.Vlg, 2008. ISBN: 9783486585216. URL: http://books.google.at/books?id=p_3VOQAACAAJ.

[63] R. Pichler. *Scrum- Agiles Projektmanagement erfolgreich einsetzen*. Dpunkt.Verlag GmbH, 2007. ISBN: 9783898644785. URL: http://books.google.at/books?id=KhoxPQAACAAJ.

[64] Klaus Pommerening. "Asymetrische Verschlüsselung". In: (2006). URL: http://www.staff.uni-mainz.de/pommeren/Kryptologie04/Asymmetrisch/Asymm.pdf.

[66] W. Rankl und W. Effing. *Handbuch der Chipkarten: Aufbau - Funktionsweise - Einsatz von Smart Cards*. Hanser Fachbuchverlag, 2008. ISBN: 9783446404021. URL: http://books.google.at/books?id=1hwiBO7IdQEC.

[67] G.H. Schalk und R. Bienert. *RFID: MIFARE und kontaktlose Smartcards angewandt*. Elektor, 2011. ISBN: 9783895762192. URL: http://books.google.at/books?id=Bp5LywAACAAJ.

[68] K. Schmeh. *Elektronische Ausweisdokumente*. Hanser, 2009. ISBN: 9783446419186. URL: http://books.google.at/books?id=GxReYGwrd5UC.

[69] K. Schwaber und M. Beedle. *Agile software development with scrum*. Series in agile software development. Prentice Hall, 2002. ISBN: 9780130676344. URL: http://books.google.at/books?id=BpFYAAAAYAAJ.

[72] R. Seidl, M. Baumgartner und T. Bucsics. *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken*. Dpunkt.Verlag GmbH, 2011. ISBN: 9783898647243. URL: http://books.google.at/books?id=23OkZwEACAAJ.

[73] Mallika Singh. "Test Automation in Agile Environments". In: *Agilerecord* (Juli 2010). URL: www.agilerecord.com.

[74] Sneed. "40 Jahre Software-Qualitätssicherung im Rückspiegel". In: *iqnite Deutschland 2012*. 2012.

[75] H.M. Sneed, M. Baumgartner und R. Seidl. *Der Systemtest: Von den Anforderungen zum Qualitätsnachweis*. Hanser Fachbuchverlag, 2008. ISBN: 9783446417083. URL: http://books.google.at/books?id=AUNtoTgp0E8C.

[77] A. Spillner und T. Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester- Foundation Level nach ISTQB-Standard*. Dpunkt.Verlag GmbH, 2010. ISBN: 9783898646420. URL: http://books.google.at/books?id=93GpQAAACAAJ.

[78] A. Spillner u. a. *Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard*. Dpunkt.Verlag GmbH, 2011. ISBN: 9783898647465.

[82] John Watkins. *Agile testing: how to succeed in an extreme testing environment*. Cambridge University Press, 2009.

[83] J.A. Whittaker und H.H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing*. Pearson/Addison Wesley, 2004. ISBN: 9780321194336. URL: http://books.google.at/books?id=RbZqPgAACAAJ.

[87] H. Wolf und W.G. Bleek. *Agile Softwareentwicklung*. Dpunkt.Verlag GmbH, 2010. ISBN: 9783898647014. URL: http://books.google.at/books?id=SVq5SQAACAAJ.

[88] H. Wolf, S. Roock und M. Lippert. *eXtreme programming*. dpunkt-Verlag, 2005. ISBN: 9783898643399. URL: http://books.google.at/books?id=V0SMtgAACAAJ.

[89] M.R. Woodward, D. Hedley und M.A. Hennell. "Experience with Path Analysis and Testing of Programs". In: *IEEE Transactions on Software Engineering* 6.3 (1980), S. 278–286. ISSN: 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.1980.230473.

[90] Matt Wynne und Hellesoy Aslak. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers (Pragmatic Programmers)*. Pragmatic Bookshelf, 2012. ISBN: 1934356808.

Online Referenzen

[1] *Asymmetrische Verschlüsselung. Das Verfahren sowie die Vor- und Nachteile*. URL: http://www.philipphauer.de/info/info/asymmetrische-verschluesselung/ (besucht am 22. 03. 2012).

[9] Kent Beck u. a. *Agile Manifesto*. 2001. URL: http://agilemanifesto.org/.

[13] *Certified Tester Advanced Level Syllabus*. 2007. URL: http://www.istqb.org/downloads/syllabi/advanced-level-syllabus.html (besucht am 20. 08. 2012).

[20] Xin Software Design. *Seiteneffekte*. 2012. URL: http://www.proggen.org/doku.php?id=theory:bugs:sideeffects.

[21] *EMV Specifications*. 2011. URL: https://www.emvco.com (besucht am 19. 03. 2013).

[26] *Foundation Level Syllabus*. 2011. URL: http://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html (besucht am 03. 07. 2012).

[29] Tomio Geron. *Were All Coders: Atlassian Opens Up The Engineering Sandbox*. 2012. URL: http://www.forbes.com/sites/tomiogeron/2012/02/22/were-all-coders-atlassian-opens-up-the-engineering-sandbox/2/ (besucht am 02. 04. 2013).

[30] *Gherkin*. URL: https://github.com/cucumber/cucumber/wiki/Gherkin (besucht am 23. 04. 2013).

[46] IPL. *Structural Coverage Metrics*. 1997. URL: http://www.ipl.com/pdf/p0823.pdf (besucht am 17. 02. 2013).

[52] *Mastercard paypass*. 2012. URL: http://www.mastercard.com/at/merchant/paypass.html (besucht am 12. 01. 2013).

[56] *MULTOS Consortium*. 2013. URL: http://www.multos.com/ (besucht am 01. 04. 2013).

[58] *NFC Forum Specifications*. 2011. URL: http://www.nfc-forum.org/specs/ (besucht am 19. 03. 2013).

[59] Dan North. *Introducing BDD*. URL: http://dannorth.net/introducing-bdd/ (besucht am 22. 04. 2013).

[60] D. Panchal. *What is Definition of Done?* Sep. 2008. URL: http://www.scrumalliance.org/articles/105-what-is-definition-of-done-dod.

[65] *Programmfehler*. 2013. URL: https://de.wikipedia.org/wiki/Programmfehler#Fehlerfreiheit (besucht am 28. 01. 2013).

[70] Sutherland J. Schwaber K. *The Scrum Guide*. Website. 2011. URL: http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf (besucht am 01. 08. 2012).

[71] *Secure Shopping*. 2012. URL: http://www.secure-shopping.at (besucht am 17. 03. 2013).

[76] Mountaingoat Software. *Scrum Figures*. 2005. URL: http://www.mountaingoatsoftware.com/scrum/figures.

[79] *TNS mPayment Sonar*. 2013. URL: http://www.tns-infratest.com/presse/pdf/Presse/2013_02_08_TNS_Infratest_TNS-mPayment-Sonar.pdf (besucht am 15. 03. 2013).

[80] *Visa paywave*. 2012. URL: http://www.visaeurope.at/at/visa_%E2%80%93_das_unternehmen/innovationen/visa_paywave.aspx (besucht am 03. 01. 2013).

[81] Kelly Waters. *User Stories*. Jan. 2008. URL: http://www.allaboutagile.com/user-stories/ (besucht am 12. 07. 2012).

[84] Wikipedia. *Zeckendorf's theorem*. 2012. URL: https://en.wikipedia.org/wiki/Zeckendorf%27s_theorem (besucht am 27. 12. 2012).

[85] Wikipedida. *Fibonacci Zahlen*. 2012. URL: http://de.wikipedia.org/wiki/Fibonacci-Folge (besucht am 12. 06. 2012).

[86] Barbara Wimmer. *Alle Bankomatkarten bekommen NFC-Funktion*. 2013. URL: http://futurezone.at/produkte/13607-alle-bankomatkarten-bekommen-nfc-funktion.php (besucht am 08. 04. 2013).