Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alex Hense

Evaluierung agiler Vorgehensmodelle in der Softwareentwicklung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alex Hense

Evaluierung agiler Vorgehensmodelle in der Softwareentwicklung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Abgegeben am 13.02.2012

Alex Hense

Thema der Bachelorarbeit

Evaluierung agiler Vorgehensmodelle in der Softwareentwicklung

Stichworte

Agiles Vorgehensmodell, agile Methoden, Best Practices, Evaluierung, Extreme Programming (XP), Scrum, Crystal-Familie, Kanban, Feature Driven Development (FDD), Einarbeitungszeit, Transparenz, Skalierung, verteiltes Team, Dokumentation, Qualitätssicherung, Flexibilität, Kundeninvolvierung

Kurzzusammenfassung

Die agilen Vorgehensmodelle Extreme Programming, Scrum, Crystal-Familie, Kanban und Feature Driven Development werden im Rahmen dieser Arbeit auf die Kriterien Einarbeitungszeit, Transparenz, Skalierbarkeit und verteiltes Arbeiten, Dokumentationsaufwand, Qualitätssicherungsmaßnahmen, Flexibilität bei Anforderungsänderungen und Kundeninvolvierung untersucht und bewertet. Die Bewertung erfolgt relativ zueinander anhand der aufgeführten Argumente für oder gegen das untersuchte Kriterium.

Alex Hense

Title of the paper

Evaluation of agile process models in software development

Keywords

Agile process model, agile methods, best practices, evaluation, Extreme Programming (XP), Scrum, Crystal-Family, Kanban, Feature Driven Development (FDD), implementation period, transparency, scalability, distributed team, documentation, quality assurance, flexibility, customer involvement

Abstract

This bachelor thesis evaluates the agile process models Extreme Programming, Scrum, Crystal-Family, Kanban and Feature Driven Development to the criteria of implementation period, transparency, scalability and distributed work, documentation efforts, quality assurance activities, flexibility for changes in requirements and customer involvement. They are measured relative to each other based on the arguments for or against the investigated criterion.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Für die Entwicklung einer Software steht den Projektbeteiligten eine große Auswahl an Vorgehensmodellen zur Verfügung (Vgl. [Wiki11]). Die Entscheidung, für ein gegebenes Problem das richtige Vorgehen zu wählen, kann sehr schwer fallen. Es lassen sich zwei unterschiedliche Ansätze erkennen. Der eine, „klassische“ Ansatz, gilt als statisch und unflexibel. Die Abfolge der Entwicklungsschritte ist strikt vorgegeben, wonach sich die Menschen und die Prozesse ausrichten. Jegliche Abweichung vom Plan kann sehr kostspielig werden. Der Kunde bekommt sein Produkt erst am Ende der Entwicklung als Ganzes zu Gesicht. Der andere, „agile“ Ansatz, versucht diesen umstrittenen Eigenschaften entgegenzuwirken. So sind in den letzten Jahren zahlreiche agile Vorgehensmodelle entstanden, die für nahezu jede Projektart mit ihren speziellen Herausforderungen eine adäquate Methode anbietet, um diese zu beherrschen und zu meistern (Vgl. [Wiki12]; [Comp11]).

Die agilen Ansätze stellen den Menschen und die zwischenmenschliche Kommunikation in den Vordergrund. Sie versuchen die Abläufe leicht anpassungsfähig zu halten, sodass auch in der Endphase eines Projekts auf Veränderungen der Rahmenbedingungen reagiert werden kann (Vgl. [Stey10], S. 9). Das Entwicklungsteam liefert dem Kunden in regelmäßigen Abständen funktionierende Teile der Software, damit er den Fortschritt sehen und Verbesserungsvorschläge und Ergänzungen machen kann. Auf diese Weise ist der Kunde in den Entwicklungsprozess einbezogen. Das Risiko für das Team eine ungenügende oder nicht gewollte Software auszuliefern wird minimiert.

Eine wichtige Aufgabe bei der Entwicklung der Software besteht somit darin, sich zu Beginn für ein Vorgehensmodell zu entscheiden. Dieses muss sich für die gegebene

Projektart eignet oder sich sogar darauf maßschneidern lassen. Die Entscheidung ist essentiell, denn ist sie falsch gefällt, können viele ungünstige Faktoren das Scheitern des Projekts bewirken. Die Beteiligten könnten sich bewusst oder unbewusst gegen den Prozess wehren, weil sie ihn nicht verstehen, keinen Fortschritt sehen und unmotiviert sind. Das Vorgehen ist eine wichtige Orientierungshilfe für das Entwicklungsteam und eine Art Bauplan für das Vorhaben. Dieser Bauplan gibt die Möglichkeit, zu erledigenden Aufgaben richtig zu koordinieren, den Ablauf zu optimieren, die Komplexität übersichtlich und beherrschbar zu halten, das Projekt erfolgreich in dem vorgegeben Rahmen durchzuführen.

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht darin, die Werte, Praktiken, Abläufe und andere relevanten Aspekte der ausgewählten Vorgehensmodelle unter verschiedenen Gesichtspunkten zu beleuchten. Für die Untersuchung wird ein Katalog mit Kriterien erstellt. Anhand der Eigenschaften, die entweder positive oder negative Auswirkungen auf das Kriterium aufweisen, erfolgt eine in Relation zueinander gestellte Bewertung, dieser, nach den Modellen vorgesehen Eigenschaften. Die Auswahl der Vorgehensmodelle ist nach ihrer *Popularität*, *Aktualität* oder der *herausstechenden Vorgehensweise* getroffen. Demnach ist *Extreme Programming* wegen der *extremen* Arbeitsweise wie das Pair Programming ausgewählt, *Scrum* wegen der hohen Beliebtheit und der Methodiken zur Beherrschung der Komplexität, *Crystal-Familie* wegen des Ansatzes, für jede Projektart eine passende Methode anbieten zu können, *Kanban* für die Softwareentwicklung, weil das eine relativ neue Veränderungsmanagement-Methode ist, die in Unternehmen eine Veränderung besonders unkompliziert vollziehen soll, und das *Feature Driven Development* wegen der Planbarkeit und der interessanten Ansätze für die Skalierung.

1.3 Aufbau

Kapitel 1 beschreibt die Motivation, die Zielsetzung, den Aufbau und bestimmt die Bedeutung relevanter Begriffe. Kapitel 2 vermittelt die Grundsätze der agilen Softwareentwicklung. Es wird auf die geschichtliche Entstehung der „Agilen Bewegung“ eingegangen, der Inhalt des „Agilen Manifests“ zusammenfassend wiedergegeben und der Unterschied zu den klassischen Vorgehensmodellen aufgezeigt. Kapitel 3 stellt die Bewertungskriterien vor. Danach folgt im Kapitel 4 die Vorstellung und die Untersuchung der ausgesuchten agilen Vorgehensmodelle. Kapitel 5 rundet die Arbeit mit einer Schlussbetrachtung und einer Zusammenfassung der Bewertungsergebnisse ab.

1.4 Begriffsbestimmung

Einige Begriffe, die in dieser Arbeit verwendet werden, findet man in der Literatur oft unterschiedlich definiert. Um Missverständnisse zu vermeiden, werden die benutzten und gemeinten Definitionen in diesem Abschnitt aufgeführt.

Vorgehensmodell

Synonyme: Vorgehen, Methodik, Methode

Das Wort wird mit der Definition verwendet, die den gesamten geordneten Ablauf eines Softwareentwicklungsprozesses umfasst. Das beinhaltet die anfänglichen Phasen, wie das Sammeln von Anforderungen und Benutzerwünschen, bis hin zu der tatsächlichen Programmierung und die Übergabe des Endprodukts an den Kunden.

Projektart

Die Art eines Projekts kann unterschiedliche Aufgaben für die Beteiligten bedeuten und somit auch speziellen Rahmenbedingungen beschreiben. Diese können die Größe (Anzahl beteiligter Personen), der Typ (z.B. Wartung, Neuentwicklung), die Projektzugehörigkeit (internes oder externes Projekt) und die Art der Abrechnung (Festpreis oder Bezahlung nach Aufwand) sein.

Leichtgewichtig

Vorgehensmodelle sind leichtgewichtig, wenn sie einen flexiblen und unbürokratischen Ablauf beschreiben. Nicht alle Rahmenbedingungen für das Endprodukt werden von Anfang an klar formuliert und festgehalten. In Laufe des Projekts können beispielsweise neue Funktionalitäten gefordert und manche vorhandene verworfen werden. Das macht die meistens aufwendige Erstellung der Dokumente überflüssig. Besonders zu empfehlen ist diese Art von Vorgehensmodellen für kleine Teams. Der Informationsaustausch im Team und auch mit dem Kunden kann somit schneller und unkomplizierter fließen, wodurch sich das Team stärker auf die Erstellung höherwertiger Software konzentrieren kann (Vgl. [Hans10], S. 3).

Schwergewichtig

Schwergewichtig sind Vorgehensmodelle, die stark formell und klar strukturiert sind. Jeder Schritt ist vordefiniert und wird auch entsprechend ausführlich dokumentiert.

Wechselnde Anforderungen an das Produkt stellen eher einen Störfaktor des starren Ablaufs dar. Modelle dieser Art eignen sich besonders gut für die Entwicklung von Systemen, bei denen ein Ausfall gravierende Folgen zufolge hätte. Dazu zählen extrem hohe Kosten oder Gefährdung von Menschenleben (Vgl. [Hans10], S. 2).

Artefakte
Synonyme: Dokumentation

Das sind Dokumente und Unterlagen, die während eines Softwareentwicklungsprozesses entstehen. Dazu zählen unter anderem das Spezifikationsdokument, die Testfälle, das Benutzerhandbuch oder auch der Quellcode der Software.

Vorhaben
Synonyme: Projekt

Ein Vorhaben hat ein klares Ziel, zeitliche, finanzielle und sachliche Vorgaben und eine Definition des Endzustands.

Klassisch *(bezogen auf Vorgehensmodelle)*
Synonyme: traditionell

Mit klassischen oder traditionellen Vorgehensmodellen sind im Wesentlichen das Wasserfall- bzw. das V-Modell gemeint. Unter einer klassischen Organisation wird die unflexible und sequenzielle Ablauforganisation dieser Vorgehensmodelle verstanden.

Der Kunde
Synonyme: Anwender, Benutzer, Auftraggeber

Der Kunde gehört zu der Seite des Arbeitgebers. Er vertritt seine Interessen und versorgt die Entwickler mit benötigten Informationen, hauptsächlich in Form von Anforderungen an das Softwaresystem. Der Kunde kann sowohl nur ein Projektverantwortliche, als auch der spätere Anwender der Software sein.

Kapitel 2

Agilität in der Softwareentwicklung

2.1 Entstehungsgeschichte

Die agilen Vorgehensmodelle beruhen auf den Vorgaben des agilen Manifests [Mani01]. Ausgearbeitet wurde das Manifest im Jahre 2001 von den 17 Vertretern der Ansätze, die sich mit iterativ-inkrementellen Entwicklung beschäftigten haben. Die Ansätze versuchen, im Gegensatz zu den klassischen Modellen, den Entwicklungsprozess besonders flexibel und schlank zu halten. Der Prozess wird durch Werte, Prinzipien und Methoden beschrieben, die stärker die menschlichen und weniger die bürokratischen Aspekte berücksichtigen (Vgl. [Stey10], S. 17). Um diese Eigenschaften zu beschreiben, entschieden sich die 17 Verfasser des Manifests für das Wort „Agil“. Ein Prozess sei laut H. Wolf erst dann agil, wenn die Beteiligten über diesen ständig reflektieren und versuchen würden, diesen immer weiter zu verbessern. Blindes Einsetzen der bekannten Elemente agiler Vorgehensmodelle reiche bei Weitem nicht aus ([Wolf11], S. 146).

Um Agilität zu beschreiben stellt J. Rasmusson die Frage auf was nötig sei, um jede Woche etwas „von Wert“ liefern zu können. Aus der Sicht des Kunden stünden, seiner Meinung nach, keine Dokumente oder Pläne, sondern ein Teil des bestellten und funktionierenden Produkts im Vordergrund ([Rasm10], S. 3-4). Demzufolge kann die Zufriedenheit des Kunden durch regelmäßige Lieferung von qualitativ hochwertiger Software positiv beeinflusst werden.

Als Antwort auf die Frage gibt er auf Seiten 4-5 seines Buches sechs Leitsätze an:

1 *“You break big problems down into smaller ones“*

Um ein großes Problem zu lösen, zerteile es in kleinere beherrschbare und löse sie zuerst. Anschließend füge die Teillösungen zusammen und löse somit das

ursprüngliche Problem (Grundsatz „Teile und herrsche“, eng. “Divide and conquer“) (Vgl. [Logo08], S. 145).

2 “You focus on the really important stuff and forget everything else”

Damit der Kunde jede Woche etwas vom Wert bekommt, halte das zu präsentierende Ergebnis schlank und leicht. Konzentriere dich auf das Wesentliche. Der Kunde soll nur das bekommen, was er auch sehen muss und will - nämlich eine funktionierende Software.

3 “You make sure that what you are delivering works”

Das gelieferte Teilprodukt sollte auch wirklich funktionieren. Deswegen sollte es gründlich getestet sein. Teste oft, viel und täglich, damit das zur Lebensweise[1] wird.

4 “You go looking for feedback”

Hole von dem Kunden regelmäßig Feedback ein. So verliert man das Ziel nicht aus den Augen und wird, falls nötig, wieder auf den richtigen Weg gebracht.

5 “You change course when necessary”

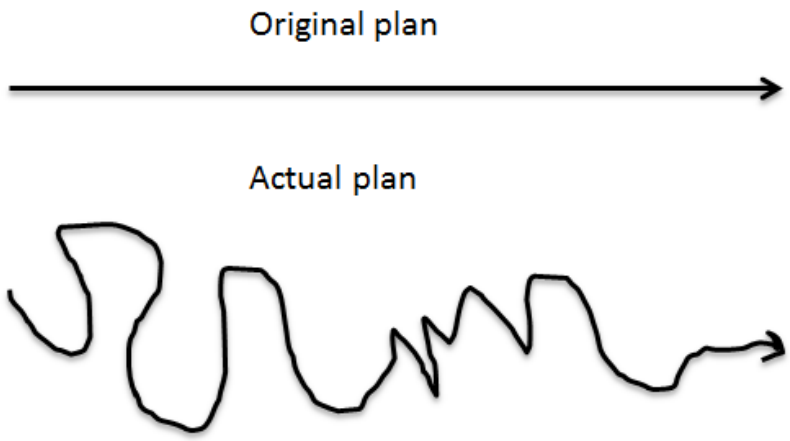

Abbildung 1: Möglicher Verlauf eines Projekts ([Rasm10], S. 5)

Vieles kann sich schnell ändern, deswegen folge nicht blind dem ursprünglichen Plan. So messe deinen Plan mit der Wirklichkeit. Verändere deinen Plan und nicht die Wirklichkeit.

6 “You become accountable”

[1] Frei übersetzt aus dem Englischen: “Daily testing becomes a way of life“ (Vgl. [Rasm10], S. 5).

> Mit der Übernahme eines Auftrags schuldet man dem Kunden gegenüber Rechenschaft über das ausgegebene Geld. Gib das Geld so aus, als wäre es dein eigenes.

2.2 Die klassische Vorgehensweise

Die klassischen Vorgehensmodelle, wie das Wasserfallmodell oder das V-Modell, bilden einen direkten Wiederspruch zu den agilen Methoden. Diese unterteilen den Entwicklungsprozess in klar gegliederte, voneinander abgegrenzte, sequenziell ablaufende und ausführlich dokumentierte Phasen. Jede dieser Phasen hat ein Fertigstellungsdatum, ein festes Ergebnis und darf erst anfangen, wenn die vorherige abgeschlossen ist. Die typischen Phasen des Wasserfallmodells sind Planung, Definition, Entwurf, Implementierung, Test, Einsatz und Wartung (siehe Abbildung 2) (Vgl. [Glog11], S. 6). Eine Rückkopplung zwischen den Phasen war anfangs nicht vorgesehen und wurde erst später in das Wasserfallmodell eingefügt, allerdings nur zu der vorherigen Phase. Das V-Modell erweitert die wasserfallartigen Phasen auf dem absteigenden, um die korrespondierende qualitätssichernde Phasen auf dem aufsteigenden Ast des Buchstaben „V“ (siehe Abbildung 3). Während z.B. die Anforderungen definiert werden, werden parallel dazu auch die Abnahmetests definiert, die die Software erst in der letzten Phase bestehen muss. Die Rollenverteilung ist ebenfalls fest definiert. Jedes Team muss für bestimmte Aufgabenbereiche Spezialisten beinhalten, die sich ausschließlich mit diesen beschäftigen. Der Kunde ist bei der Erfassung von Anforderungen in Form eines Lastenhefts und bei der Übernahme des vollständigen Softwareprodukts in das Projekt involviert.

Die Vorteile der klassischen Vorgehensweise sind:

- erleichterte Planung und Kontrolle
- strikt voneinander abgegrenzte Phasen
- Effektivität bei festen Anforderungen und Kosten
- Einfacher Ablauf
- Kundenbeteiligung ist nur am Anfang und Ende des Projekts erforderlich

Die Nachteile sind:

- Flexibilität bei Änderungen der Rahmenbedingungen ist nicht gegeben
- Fehlererkennung erfolgt sehr spät und die Beseitigung ist kostspielig
- Big Bang Test: Getestet wird erst, wenn alles fertig ist
- Umfangreiche und detaillierte Dokumentation ist notwendig
- Ergebnisse sind für den Kunden erst am Projektende sichtbar
- Rückkopplung ist nur zwischen den benachbarten Phasen möglich

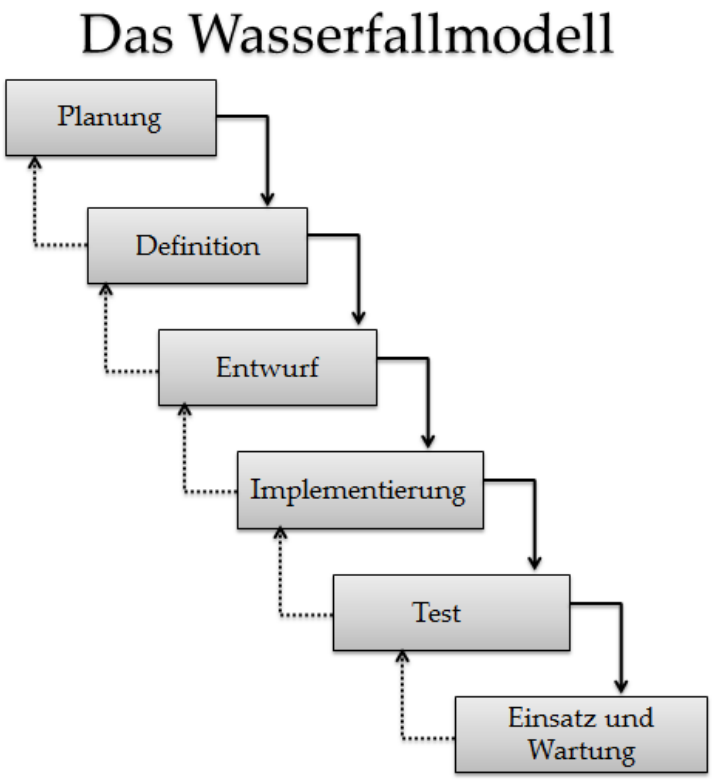

Abbildung 2: Das Wasserfallmodell

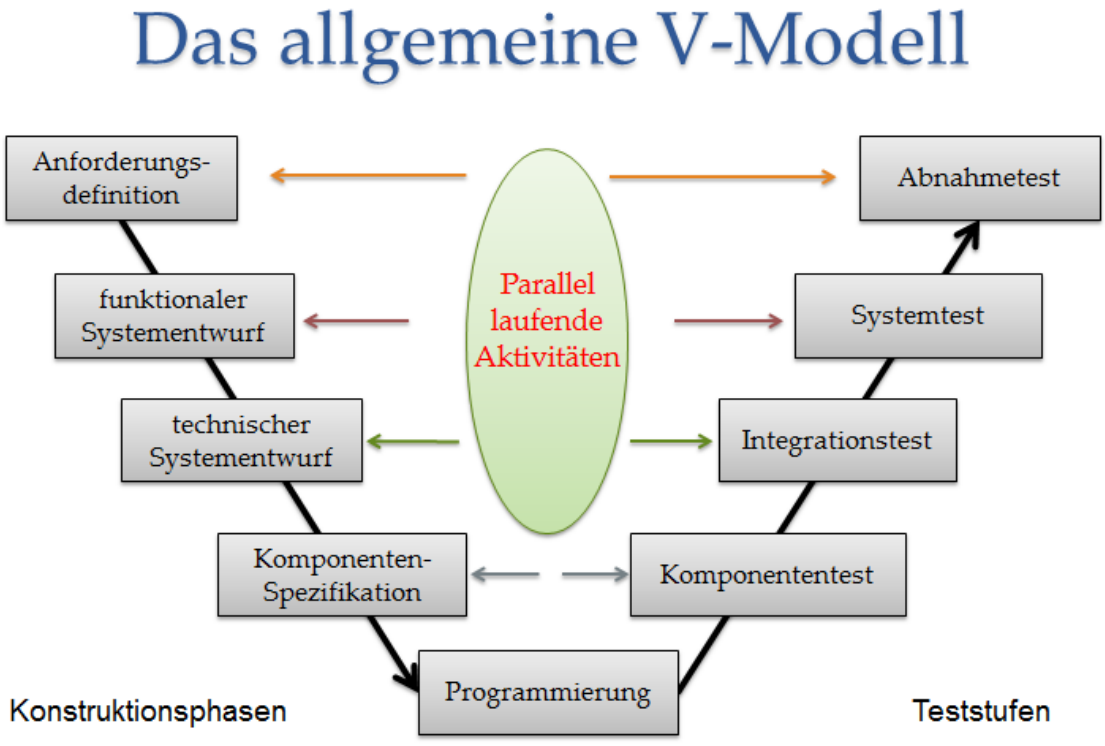

Abbildung 3: Das allgemeine V-Modell

Kapitel 3

Bewertungskriterien

Im Folgenden werden die Bewertungskriterien mit einer Erläuterung vorgestellt, nach denen die ausgesuchten agilen Vorgehensmodelle evaluiert werden. Die Kriterien sind aus den Werten des agilen Manifests abgeleitet, ergänzt um einige gemeinsamen Merkmale, die bei der Entscheidungsfindung für oder gegen ein Modell von Interesse sein könnten. Die Bewertung erfolgt relativ zueinander anhand der aufgeführten Argumente, die als positiv oder negativ für das jeweilige Untersuchungskriterium eines Modelles ausgelegt werden.

Das Bewertungssystem ist folgendes:

- Zeitraum: ***sehr kurz, kurz, lange, sehr lange***
- Aufwand: ***sehr gering, gering, groß, sehr groß***
- Involvierung: ***sehr stark, stark, ausreichend, schwach***
- Sonstige Eigenschaften: ***sehr gut, gut, ausreichend, mangelhaft***

3.1 Einarbeitungszeit

Einführung einer neuen Arbeitsweise, in diesem Fall eines agilen Vorgehensmodells, könne laut M. Steyer eine sehr große Änderung für die Mitarbeiter bedeuten. Dabei treffe man leicht auf Wiederstand, wenn die neuen Methoden den alten, sich bereits bewehrten widersprechen ([Stey10], S. 106-107).

Bis eine neue Methodik von den Anwendern verstanden wird und von denen eingesetzt werden kann, kann viel Zeit vergehen. Diese Zeitspanne hängt unter anderem von der Komplexität und der Vielfallt der Abläufe ab und wie strickt die

Vorgehensweise vordefiniert ist, bzw. auf welche Art und Weise die Beteiligten diese anpassen können. Dazu kommt noch die Größe des Teams. Je größer es ist, desto länger kann es dauern, bis jedes Mitglied den Ablauf akzeptiert und verinnerlicht hat. Jeder Mensch lernt unterschiedlich schnell und brauch eventuell eine andere Herangehensweise um etwas zu begreifen.

Dieses Kriterium untersucht das Vorgehensmodell nach Eigenschaften, die die Einführung erleichtern bzw. erschweren können.

3.2 Transparenz

Viele Projekte seien in ihrem Umfang und Komplexität zu mächtig um diese realitätsnah mit geringem Risiko kontinuierlich planen zu können. Ein gewisses Maß an Erfahrung für die Anwendung der empirischen Steuerung des Prozesses sei notwendig und es müsse ausreichend Informationen über den aktuellen Zustand verfügbar sein. Die Transparenz des gesamten Prozesses stelle nach K. Schwaber und J. Sutherland einen wichtigen Faktor dar ([ScrG11], S. 4).

Außerdem, damit ein Vorgehensmodell eine hohe Benutzerakzeptanz erfährt, muss das möglichst unkomplizierte und leicht zu verstehende Strukturen aufweisen. Der Kunde als Auftraggeber, wie auch das Entwicklungsteam, muss ständig Zugriff auf Informationen haben, die ihnen möglichst schnell und einfach eine Übersicht über die aktuellen Fortschritte und auch über die anstehenden Aufgaben geben. Der Einsatz von visuellen Mitteln zur Veranschaulichung der Abläufe und die Darstellung des aktuellen Standes des Projekts tragen einen großen Teil dazu bei.

Dieses Kriterium untersucht ein Vorgehensmodell auf die Mittel und Wege, die für die Beteiligten das Verfahren und den aktuellen Entwicklungsstand durchsichtiger gestalten.

3.3 Skalierbarkeit und verteiltes Arbeiten

Die agilen Vorgehensmodelle sind für die Anwendung in kleineren Teams entwickelt worden. Für ein zu großes Team besteht die Gefahr, die agilen Prinzipien nicht mehr einhalten zu können. Für ein größeres Projekt, für das die beteiligten Personen nicht alle in einem Raum untergebracht werden können und auf mehrere Räume oder sogar Gebäude verteilt werden müssen, funktioniert die direkte Kommunikation „von Angesicht zu Angesicht“, eins der wichtigsten agilen

Prinzipien[2], nicht mehr. Bei einem zu kleinen Team kann das Projekt unnötig lange Laufzeit haben. Nach der Fertigstellung der Software können die benutzten Technologien schon lange überholt und nicht mehr effizient sein.

Es war vor nicht so langer Zeit üblich, dass das ganze Team am selben Standort zusammengearbeitet hat. Dieser Zustand ist heute eher die Ausnahme (Vgl. [Cohn], S. 385). Um Entwicklungskosten zu senken verlagern viele Unternehmen ihre Geschäftsprozesse an spezialisierte Drittunternehmen ins Ausland (Outsourcing). Bei Auslagerung dieser Art an eine andere Niederlassung desselben Unternehmens (Offshoring) und bei der Zusammenarbeit mit einem ausländischen Team können Situationen entstehen, wodurch in gemeinsamen Projekten ein verteiltes Team entsteht. Dieses Team kann nicht nur durch geografische Entfernung, sondern auch durch verschiedene Zeitzonen getrennt sein. Für erfolgreiche Anwendung eines Vorgehensmodells müssen Techniken zur Skalierung, sowie für Überwindung der Entfernung vorhanden sein.

Mit diesem Kriterium wird das Vorgehensmodell auf die Möglichkeit zur Skalierung, sowie auf die Eignung für verteiltes Arbeiten untersucht.

3.4 Dokumentationsaufwand

Einer der Kritikpunkte an traditionellen Vorgehensmodellen sei, laut A. Schatten, „ein (scheinbar unnötig) hoher Dokumentationsaufwand“ ([Scha10], S. 62). Jede Softwareentwicklung müsse laut K. Eilebrecht angemessen dokumentiert sein ([Eile10], S. 15-18). Das kann für die Entwickler selbst wichtig werden oder auch für das Team, das später die Software warten bzw. weiterentwickeln soll. Eine gute Dokumentation sei nach Eilbrecht unter anderem redundanzfrei, aus der Sicht des Lesers geschrieben und vollständig[3]. E. Tachtsoglou und R. Arndt empfehlen sogar als Fazit ihrer Arbeit über Anforderungen an Dokumentation die „Einführung einer eigenen Projektphase nur für die Erstellung der Dokumentation“ ([Tach09]). Bei agilen Modellen steht nach dem Wert des agilen Manifests „Funktionierende Software mehr als umfassende Dokumentation“ funktionierender Code im Vordergrund [Mani01]. Trotz dem ist es nicht verboten zu dokumentieren, es wird nur anders gewichtet (Vgl. [Hans10], S. 7).

Dieses Kriterium soll den Aufwand wiedergeben, der für die Erstellung und die Pflege der Artefakte nach dem Vorgehensmodells vorgesehen ist.

[2] Vgl. Die Prinzipien des agilen Manifest: http://www.agilemanifesto.org/iso/de/principles.html (Feb. 2012).
[3] *Vollständig* bedeutet, dass auch jede kleinste selbsterklärende Methode kommentiert sein muss.

3.5 Qualitätssicherungsmaßnahmen

Eine Studie hat sich mit Qualitätssicherung in der Softwareentwicklung beschäftigt. Befragt wurden über 800 Entwickler, Tester und Projektleiter. Die Studie zeigt, dass mehr als ein Drittel der Befragten schon in den anfänglichen Entwicklungsphasen Qualitätssicherungsmaßnahmen durchführen, jedoch 37% setzen in den frühen Projektphasen keine ein. Ein Viertel der an der Studie teilnehmenden arbeiteten nach einem agilen Vorgehensmodell (hauptsächlich nach Scrum). Die Kosten betrügen durchschnittlich ein Fünftel des Gesamtbudgets (Vgl. [Neum11]).

Die Zahlen aus der Studie zeigen, dass die Qualitätssicherung eine wichtige Rolle in der Softwareentwicklung spielt. Nach der Meinung der Studieninitiatoren sei sie aber noch nicht an die agilen Methoden angepasst, weswegen 17% der Befragten keinem Vorgehensmodell folgen würden.

Dieser Untersuchungspunkt beschäftigt sich mit der Frage, in welcher Form und welchem Umfang nach einem Vorgehensmodell Qualitätssicherung betrieben wird.

3.6 Flexibilität bei Anforderungsänderungen

Eines der Werte des agilen Manifests ist das „Reagieren auf Veränderungen mehr als das Befolgen eines Plans“ [Mani01]. Es ist demzufolge entscheidend, wie schnell und flexibel auf neue und sich ändernde Anforderungen reagiert werden kann. Ein agiles Vorgehensmodell muss für Änderungen entsprechende Abläufe anbieten. Falls die neu geforderte Funktionalität die höchste Priorität bekommt, muss die als Nächstes umgesetzt werden. Dient diese z.B. nur der Benutzerfreundlichkeit, dann könnte die Umsetzung auch später erfolgen. Nach der Aussage von A. Schatten stelle die „unter anderem mangelhafte Flexibilität (durch vorgegebene starre Strukturen)“ einen weiteren Kritikpunkt an traditionellen Vorgehensmodellen dar ([Scha10], S. 62). Seine anschließende Erläuterung dazu ist: „Eine enge Zusammenarbeit mit dem Kunden ermöglicht eine flexible Handhabung von Anforderungen (auch deren Änderungen) und eine unmittelbare Rückmeldung des Kunden in Bezug auf gelieferte Software-Komponenten.“

Demzufolge untersucht dieses Kriterium, wie flexibel nach einem Vorgehensmodell auf Änderungen der Anforderungen reagiert werden kann.

3.7 Kundeninvolvierung

A. Schatten nennt „mangelnde Einbeziehung des Kunden in den Entwicklungsprozess“, als einen der Kritikpunkte an den traditionellen Vorgehensmodellen. Des Weiteren erläutert er: „Dabei spielt die Interaktion mit dem Kunden eine zentrale Rolle“ ([Scha10], S. 62). Der Kunde vertritt oder er ist selbst der spätere Anwender der geplanten Software. Er weiß am besten über die Einzelheiten des Systems, bzw. über seine Funktionalitäten Bescheid und ist auch in der Lage, diese in angemessener Art und Weise an die Entwickler weiterzugeben.

Dieser Untersuchungspunkt beschäftigt sich mit dem nach einem Vorgehensmodell vorgesehenen Ausmaß des Kundeneinbezugs in die Entwicklungsaktivitäten.

Kapitel 4

Agile Vorgehensmodelle

4.1 Extreme Programming (XP)

XP ist im Rahmen einer Softwareentwicklung im Bereich der Lohnabrechnung bei der Firma Chrysler im Jahre 1996 entstanden. Ein Projekt hatte keinen Erfolg und Kent Beck bekam die Möglichkeit das gescheiterte Vorhaben wiederzubeleben. Das gab ihm die Gelegenheit seine bis dahin informellen Praktiken als eines der ersten agilen Vorgehensmodelle zu formulieren. Diese bedeuteten einen *extremen* Wandel in der Arbeits- und Denkweise, wodurch die Methodik auch ihren Namen bekam.

4.1.1 Die Werte von XP

K. Beck gibt für XP in „Rules and Practices“ ein strukturiertes iterativ-inkrementelles Vorgehen für die „traditionellen“ Phasen Planung, Design, Kodierung und Testen vor.[4] Die Phasen sind in der Reihenfolge nicht festgelegt, obwohl einige erst als logische Folge der anderen anfangen können (erst die Planung, dann die Codierung). Die strikte Vorgabe der Abläufe unterscheidet ihn von den anderen Vertretern[5] des agilen Manifests. Die fünf zentralen Werte (siehe Abbildung 4) machen zusammen mit den 14 von ihm definierten Prinzipien[6] die bewerten Praktiken erst anwendbar (Vgl. [Hans10], S. 13-17):

[4] Später erfolgte eine Anpassung. Siehe http://www.extremeprogramming.org/rules.html (Feb. 2012).
[5] Liste der 17 Vertreter des agilen Manifests: http://www.agilemanifesto.org/authors.html (Feb. 2012).
[6] Die 14 Prinzipien werden in dieser Arbeit nicht weiter behandelt.

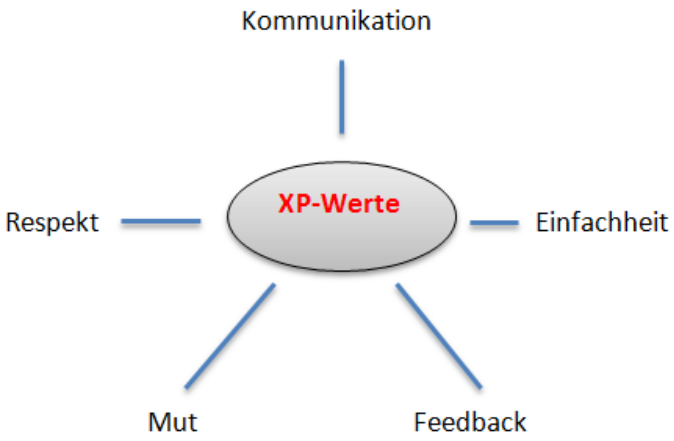

Abbildung 4: Zentrale Werte von XP

Kommunikation

Sie ist, sowohl mit dem Kunden als auch zwischen den anderen Teammitgliedern, für ein agiles Projekt von größter Bedeutung.

Einfachheit:

In jeder Phase des Projekts ist die einfachste Lösung die beste. Es darf beispielsweise kein Algorithmus jetzt schon geschrieben werden, der erst für spätere Funktionalitäten gedacht ist.

Feedback:

Sowohl von der Software selbst durch erfolgreich durchlaufene Testfälle, als auch von dem Kunden nach abgeliefertem Produkt, durch das Feedback erhält das Team wichtige Informationen über den Fortschritt, die Qualität und die Richtigkeit der Software.

Mut:

Durch die Möglichkeit im Verlauf eines laufenden Projekts neue Anforderungen realisieren zu können, muss der Quellcode ständig neu angepasst werden (Refactoring). Das kann so weit gehen, bis es zur Implementierung einer neuen Funktionalität einfacher wäre, das ganze Design neu zu entwerfen und somit von vorne anzufangen. Dies dem Kunden gegenüber zuzugeben erfordert Mut, den man um diese Entscheidung zu treffen aufbringen muss.

Respekt:

Dieser Wert wurde später ergänzt. Respektvoller Umgang miteinander und auch mit dem Kunden sei nach K. Beck eine solide Basis für Umgang miteinander und nach E. Hanser eine „Grundlage für ein erfolgreiches Projekt“ ([Beck04]; [Hans10], S. 17).

4.1.2 Die XP-Praktiken

Abbildung 5 zeigt die „Best Practices“ von Extreme Programming. Sie haben als Ziele die Qualität des Produkts zu erhöhen, regelmäßig Feedback einzuholen, effektiv Wissen im Team auszutauschen und die erstellte Software leicht anpassungsfähig zu halten (Vgl. [Hrus09], S. 84-85):

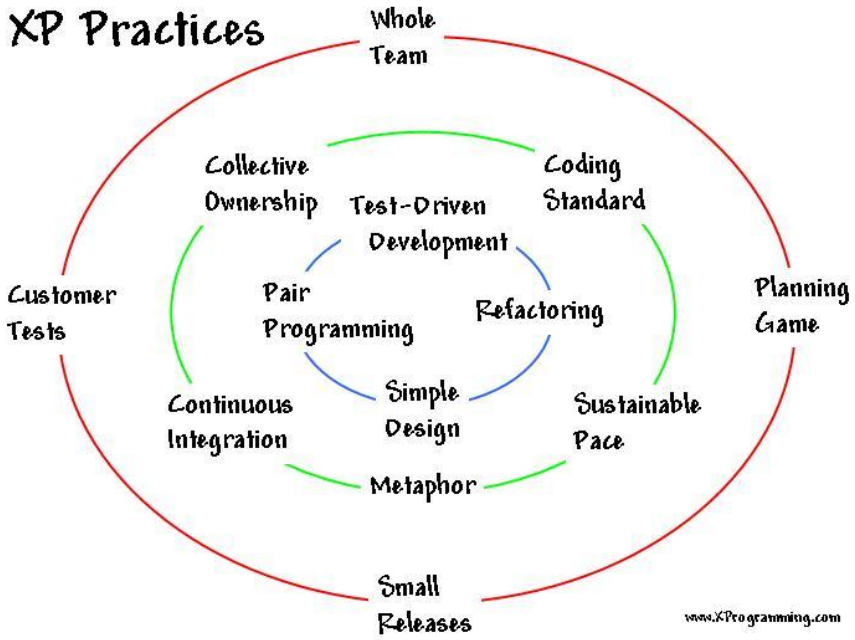

Abbildung 5: Best Practices von XP[7]

Im Folgenden werden einige der Praktiken erläutert:

Planungsspiel (Planning Game):

Ein iterativer Prozess zur Einschätzung der Kosten und Risiken für eine neue Funktionalität durch Entwickler, worauf basierend der Kunde entscheidet, was als nächstes realisiert werden soll.

Testgetriebene Entwicklung (Test-Driven Development):

Die Testfälle werden vor dem Codieren erstellt und automatisiert. So weiß der Programmierer, welche Erwartungen sein Code zu erfüllen hat und bekommt schnell eine Rückmeldung über das Ergebnis.

Einfaches Design (Simple Design):

Das Design und die Lösung für die Umsetzung einer Anforderung sollen „nur“ zu 100% der geforderten Anforderungen erfüllen. Keine Vorbereitungen für die wahrscheinlich noch später kommenden Anforderungen dürfen getroffen werden.

[7] Quelle: http://xprogramming.com/images/circles.jpg (Feb. 2012).

Ergebnisse gehören allen (Collective Ownership):

Alle Mitglieder des Teams können sich alle Ergebnisse und Artefakte ansehen und diese auch verändern.

Programmierung in Paaren (Pair Programming):

Es wird in Zweier-Teams entwickelt. Einer tippt an der Tastatur, der andere überprüft gleichzeitig seine Arbeit und behält den Überblick. Anschließend tauschen sie. So können Fehler schon in kürzester Zeit durch das Vier-Augen-Prinzip gefunden und vermieden werden.

Kontinuierliche Integration (Continuous Integration):

Der geschriebene und erfolgreich getestete Code wird mindestens einmal am Tag in das Gesamtsystem integriert. So hat man zu jeder Zeit ein lauffähiges System. Der Integrationsvorgang sollte nicht länger als 10 Minuten dauern.

Programmierstandards (Coding-Standards):

Alle Entwickler richten sich nach vorgegebenen und von allen akzeptierten einheitlichen Standards und Richtlinien.

Ständige Verbesserung (Refactoring):

Das Design und der Code werden ständig von den Entwicklern kritisch begutachtet, erweitert und durch Umstrukturierung verbessert.

40 Stunden-Woche:

Nur ein erholter Programmierer, ist ein guter Programmierer. Überstunden können sich auf die Qualität der Arbeit auswirken und sind zu vermeiden.

4.1.3 Rollen bei XP

XP kennt drei Rollen mit folgenden Aufgaben (Vgl. [Rump01]):

Der Projektleiter:

Diese Rolle übernimmt eine Person aus dem Management. Die Kosten, die Ressourcen und die Zeitpläne des Projekts liegen in seinen Händen. Bei Engpässen kann er auch als Entwickler eingesetzt werden.

Der Kunde:

Er ist der Repräsentant der späteren Anwender und somit der Arbeitgeberseite. Seine Aufgaben sind das Verfassen und das Priorisiert der Anforderungen. Der Kunde ist ein Teil des Teams und ist stets für die Fragen des Teams vor Ort.

Der Entwickler:

Der Entwickler realisiert die vom Kunden gewünschten Anforderungen mit allen dazu benötigen Aufgaben in Software.

P. Hruschka nennt anstelle des Projektleiters einen *XP-Coach* als eine wichtige Rolle eines XP-Projekts. Diese habe die Aufgabe, das Team in die richtige Richtung zu leiten und ihm bei methodischen Fragen zur Seite zu stehen ([Hrus09], S. 87).

4.1.4 Der Ablauf

Der Kunde schreibt in wenigen Sätzen die gewünschten Anforderungen als *User Storys* auf. Diese werden später den jeweiligen Entwicklern noch näher erläutert und können als Grundlage für die Akzeptanztests dienen. Die Entwickler schätzen den Aufwand für die Realisierung der Anforderungen. Der Kunden priorisiert diese nach der Wichtigkeit. Das Team zerteilt die Anforderungen auf einzelne Arbeitsabläufe, sogenannte Tasks, und plant einige für die nächste Iteration ein, die eine bis drei Wochen dauert. Die Entwickler-Paare wählen eine oder mehrere für die Iteration vorgesehene Tasks aus und erledigen eigenverantwortlich unter Berücksichtigung aller Werte, Prinzipien und Praktiken die sämtliche damit zusammenhängende Aufgaben. Bei täglichen kurzen *Standup-Meetings* berichten die Entwickler gegenseitig von dem, womit jeder sich am Tag davor beschäftigte, welche Probleme es gab und was er für den jeweiligen Tag plant. Eine User Story gilt als erledigt, wenn alle dafür vorgesehenen Tests erfolgreich durchlaufen und der Kunde sie abnimmt. Danach widmen sich alle der nächsten Iteration.

4.1.5 Bewertung von Extreme Programming

4.1.5.1 Einarbeitungszeit

Eine Umfrage aus dem Jahr 2008 hat ergeben, dass Extreme Programming eins der bekanntesten agilen Vorgehensmodelle sei. Das Vorgehen sei sehr „mächtig […] aber [lässt sich] nur schwer einführen und durchhalten“ ([Wolf08], S. 10-11). 36% der Befragten entwickeln agil, 35% haben Erfahrung mit Extreme Programming, 7% planen und 14% setzen das Vorgehen bereits ein.

Manche der XP-Praktiken bereiten in der Angewöhnungsphase besondere Schwierigkeiten. Den Umstieg zum Pair-Programming vergleicht J. Eckstein, freie Beraterin und Prozesstrainerin für Extreme Programming, in dem Artikel [Sixt03] von Die Zeit mit einem Kulturschock. T. Mumme schreibt im selben Artikel, nach 15 Jahren Berufserfahrung müsse er wegen einem der XP-Leitsätze das Programmieren neu erlernen.

XP fordert Disziplin und eine Menge an Erfahrung der Entwickler und der Manager. Unerfahrene Entwickler und deren Ausbildung lassen sich nur eingeschränkt in Projekte unterbringen (Vgl. [Buns08], S. 117). Der Einsatz eines XP-Coachs kann dem Team helfen

die Abläufe zu verstehen und sich schneller an diese zu gewöhnen. Er ist kein Entscheidungsträger und hat nur eine beratende und anleitende Funktion (Vgl. [Hrus09], S. 87).

XP ist demnach ein kompliziertes und umfangreiches Vorgehensmodell, das bei der Einführung auf Wiederstand und wenig Akzeptanz stoßen kann. Deswegen ist die Einarbeitungszeit bei XP mit ***lange*** bewertet.

4.1.5.2 Transparenz

Die *Standup-Meetings* tragen einen großen Teil zur Transparenz für das Entwicklungsteam bei. So weiß jeder, wie weit der andere ist, welche Schwierigkeiten er hatte, was er als nächstes vorhat. Außerdem können alle selbst durch die Praktik *Collective Ownership* zu jeder Zeit sehen, was und wie viel die anderen Teampaare geliefert haben. Demnach kann die eigene Leistung realistisch eingeschätzt werden.

Durch die inkrementelle Entwicklung wird dem Kunden nach jeder Iteration das Ergebnis vorgestellt. So bekommt er regelmäßig die Übersicht über den Fortschritt seines Projekts. Auch die Tatsache, dass er täglich mit dem Team zusammenarbeitet, fördert die Transparenz für den Kunden und das gegenseitige Vertrauen.

Die Transparenz bei XP ist aus aufgeführten Gründen mit ***sehr gut*** bewertet.

4.1.5.3 Skalierbarkeit und verteiltes Arbeiten

Für B. Rumpe sei die Größe eines XP-Teams von maximal 10 Programmierern ideal. Anschließend ergänzt er, es seien nach XP auch umfangreichere Projekte mit einer größeren Anzahl an Entwicklern erfolgreich abgeschlossen [Rump01].

P. Hruschka gibt als Erfahrungswert 15 Personen an. Um eine Aussage über größere Projekte zu geben, fehle die praktische Erfahrung. Es sei aber schon gelungen manche der XP-Praktiken, wie z.B. die kontinuierliche Integration, in Teams mit größerer Anzahl der Mitglieder erfolgreich anzuwenden ([Hrus09], S. 83).

A. Cockburn beschreibt einen der ersten XP-Projekte. Nachdem ein Team von 30 Personen ein Jahr lang in einem größeren Projekt keine Ergebnisse liefern konnte, habe es im selben Zeitraum ein 8er Team erfolgreich abgeschlossen. Ein kleineres Team setzte eine größere Anzahl an Methodiken ein, dass auch der Grund für den Erfolg sei. Er empfiehlt die Teams nicht mehr als auf 10 Programmierer zu skalieren ([Alis02] S. 169).

Unter Berücksichtigung aller XP-Praktiken ist eine Anzahl von mehr als 10 Teammitgliedern nicht mehr ohne weitere Maßnahmen praktikabel. Es erscheint als

problematisch sowohl aus der Sicht des Teams, das sich ständig miteinander austauschen muss, als auch aus der Sicht des Kunden, der alle aufkommenden Fragen zur Realisierung seiner Anforderungen und Wünsche beantworten muss.

Für ein erfolgreiches Zusammenarbeiten räumlich getrennter Teams, ob nun durch die Wände des Raumes, oder auch durch Wasser zwischen den Kontinenten, müssen Techniken vorhanden sein, um trotz der Trennung effektiv miteinander kommunizieren und somit arbeiten zu können. Die „Hierarchische Organisation“ sei nach B. Rumpe eine dieser Techniken. Sie sei eher für den lokalen Einsatz in einer größeren Firma gedacht. Das große Projekt werde in kleinere XP-Projekte zerteilt, denen eine auf bestimmte Weise zusammengesetzte Gruppe übergeordnet sei. Diese Gruppe koordiniere und überwache die Abläufe und beachte besonders, dass die Schnittstellen der Komponenten von den Teams zusammenpassen [Rump01].

Zur Überwindung größerer Entfernungen lassen sich manche Nachteile der Verteilung dank moderner Kommunikationswege, wie z.B. Videokonferenzen während der Standup-Meetings, minimieren.

Trotz allem ist XP für lokale Zusammenarbeit entwickelt worden und ist nur dafür besonders effektiv. Für große und verteilte Projekte sollte ein anderes Vorgehensmodell in Betracht gezogen werden. Demnach ist dieses Untersuchungskriterium mit ***ausreichend*** bewertet.

4.1.5.4 Dokumentationsaufwand

XP sieht vor Informationen von Angesicht zu Angesicht sowohl zwischen den Teammitgliedern, als auch mit dem Kunden, der sich ebenfalls vor Ort befindet, auszutauschen. Die direkte Kommunikation funktioniere, laut R. E. Jeffries, auf diese Weise schneller und unkomplizierter. Das mache die Dokumentation so gut wie überflüssig [Jeff01].

Des Weiteren konkretisiert Jeffries die Erstellung eines Anforderungsdokumentes nicht als einen Teil des XP-Prozesses. Die Anforderungen würden von dem Kunden in ein paar Sätzen auf „User Story Cards“ geschrieben. Diese können später als Abnahmekriterien dienen. Falls der Kunde ein bestimmtes Dokument (z.B. die Spezifikation) haben möchte, werde diese Aufgabe genauso wie eine Anforderung auf eine User Story Card aufgenommen und in einer der nächsten Iterationen umgesetzt.

Laut B. Rumpe seien die zahlreichen Testfälle und durch Coding Standards festgelegte Kommentierung des Quellcodes ein Ausgleich für die fehlende Dokumentation [Rump01].

Abschließend lässt sich feststellen, dass bei XP die Dokumentation hauptsächlich aus den User Stories, aus einem gut kommentierten Quellcode und aus den Testfällen besteht. Somit ist der vorgesehene Extra-Aufwand als ***sehr gering*** eingestuft.

4.1.5.5 Qualitätssicherungsmaßnahmen

Nach XP werden die Qualitätssicherungsmaßnahmen schon in den frühen Entwicklungsphasen eingeleitet. Die Testfälle müssen nach der XP-Praktik der testgetriebenen Entwicklung schon vor dem eigentlichen Codieren geschrieben und automatisiert werden. Auf diese Weise weiß der Entwickler, welche minimalen Bedingungen sein Code erfüllen muss. Die Entstehung des überflüssigen Codes wird so verhindert, dass der Praktik des einfachen Designs zugutekommt. Verschiedene Arten von Integrationstests, wie Unit-Tests (auch Komponententests genannt) und Regressionstests, kommen zum Einsatz. Bei der Vorführung der Software dem Kunden spielen die Akzeptanztests[8] eine wichtige Rolle. Somit sind die Qualität und der Überdeckungsgrad der Testfälle (sogenannte Code Coverage) für diese Aspekte entscheidend.

Durch das Pair-Programming findet außerdem ein ständiges Begutachten der aktuellen Arbeitsfortschritte durch eine zweite Person statt. Das sei nach J. Eckstein unter [Sixt03] vergleichbar mit einer „verschärfte[n] Form des permanenten Reviews." Kontinuierliche Integration und eine 40-Stunden Arbeitswoche sind ebenfalls wirkungsvolle Praktiken zur Steigerung der Qualität.

XP sieht viele Mittel zur Sicherstellung der Qualität vor. Aus den aufgeführten Gründen sind Qualitätssicherungsmaßnahmen von XP mit ***sehr gut*** bewertet.

4.1.5.6 Flexibilität bei Anforderungsänderungen

XP ist unter anderem für Projekte entwickelt worden, bei denen anfangs über die Anforderungen an die Software keine klaren Vorstellungen herrschen. So kann der Kunde jederzeit in Form von User Storys neue Wünsche äußern, die schnell und unkompliziert berücksichtigt werden können. Dieser Umstand ist auch dem Grundsatz der wenigen bis gar keinen Dokumentation zu verdanken, die durch das Festschreiben der Bedingungen den Vorgang nur änderungsresistenter gestallten würde. Das Design der Software wächst und verändert sich mit der Anzahl der implementierten Anforderungen.

Die Flexibilität ist eine wesentliche Stärke von XP und ist daher mit ***sehr gut*** bewertet.

[8] Akzeptanztests werden als Abnahmekriterien des Zwischen- oder Endzustands der Software von beiden Parteien festgelegt.

4.1.5.7 Kundeninvolvierung

Der Kunde spielt eine der wichtigsten Rollen in einem XP-Projekt. Er definiert und priorisiert die Anforderungen an das System in Form von User Stories und erläutert diese später den Entwicklern. Dieses Vorgehen erleichtert XP um das aufwendige Sammeln, Formulieren und Dokumentieren der geforderten Systemeigenschaften.

Der Kunde ist ein ständig präsentes Mitglied des XP-Entwicklungsteams und sorgt für die Berücksichtigung der Benutzerwünsche. Sein Einbezug ist unverzichtbar und ***sehr stark***.

4.1.6 Fazit

Das XP-Vorgehensmodell ist für Projekte mit anfangs vage formulierten, sich schnell ändernden Anforderungen entwickelt worden, wobei das Budget und die Zeit in Form von kurzen Auslieferungsterminen, sogenannten Releasezyklen, vorgegeben sind. Durch die Sammlung der User Story Cards werden die Anforderungen stets aktuell und lebendig gehalten, dass allerdings das Projektende schlecht vorhersehen lässt. Optimal besteht ein Team aus ca. 10 Personen. Um ein Projekt mit 100 Personen nach den XP-Werten, Prinzipien und Praktiken erfolgreich durchzuführen, muss dieses in kleinere Teilprojekte und das Team in kleinere Gruppen unterteilt werden. Dabei ist wichtig den Überblick über die verteilten Aufgaben nicht zu verlieren.

Die XP-Prinzipien sind nicht ganz unumstritten. Einen Kritikpunkt gibt es an dem Pair-Programming. Zwei Entwickler sind mit einer Aufgabe beschäftigt, die normalerweise durch eine Person erledigt wird. Da stellt sich die Frage, ob es nicht überflüssig sei und dadurch nur doppelte Kosten verursache? Mit der Antwort auf diese Frage habe sich, laut eines Artikels von Die Zeit, eine Universität in Utha 1999 beschäftigt. Es ergab, dass verglichen mit der Leistung einer einzigen Person 60% mehr Zeit benötigt werde, dafür aber deutlich weniger Fehler gemacht würden [Sixt03].

Ein weiterer Kritikpunkt ist die tägliche Integration und damit verbundene Ausführung aller Testfälle. Dies kann besonders bei großen und verteilten Systemen sehr aufwendig und zeitraubend werden.

Die ständige Umstrukturierung (Refactoring) wirkt sich negativ auf die Aspekte der Wiederverwendbarkeit und (falls vorhanden) externer Dokumentation aus. Denn die Software ist auf die spezielle Problemstellung zugeschnitten und das Design wird ständig angepasst, was die Dokumentation schnell veralten lässt.

Auch die ständige Anwesenheit des Kunden erscheint aus erster Sicht ein überflüssiger Kostenfaktor. Dies ist aber bei näherer Betrachtung eines der wichtigsten Erfolgsfaktoren dieses Vorgehensmodells.

4.2 Scrum

Scrum ist ein Begriff aus dem Rugby und heißt „Gedränge“. Damit ist der Versuch den Ball wieder ins Spiel zu bringen gemeint. Für die Softwareentwicklung ist das ein Projektmanagementframework mit wenigen konsequent einzuhaltenden Regeln, das sich auf jegliche Art und Komplexität von Projekten anwenden lässt (Vgl. [Pich08], S. 1). Bei Scrum wird angenommen, dass der Entwicklungsprozess sich aufgrund der komplexen Abläufe nicht genau planen und vorhersehen lässt. Aus diesem Grund wird nur ein grober äußerer Rahmen für die Handlung vorgegeben und die Organisation und die Steuerung übernimmt das Entwicklungsteam selbst. Jeff Sutherland und Ken Schwaber entwickelten das agile Vorgehen in den frühen 1990ern.

4.2.1 Scrum-Praktiken

Aus der Tatsache, dass Scrum eine Projektmanagementmethode ist und keine Entwicklungspraktiken vorschreibt folgt, dass das Team während der Entwicklungsphase sich selbst organisieren muss und eigene Praktiken einsetzen kann. B. Gloger empfiehlt den Einsatz der folgenden, bereits aus Extreme Programming bekannten Praktiken: *Testgetriebene Entwicklung (Test-Driven Development)*, *Ständige Verbesserung (Refactoring)*, *Gemeinsame Verantwortung / Ergebnisse gehören allen (Collective Ownership)*, *Kontinuierliche Integration (Continuous Integration)*, *Programmierung in Paaren (Pair Programming)*. Diese lenkten das Team „bewusst in die richtige Richtung“, das Team solle trotz dem ständig das Bestreben zur Verbesserung aufweisen ([Glog09], S. 185-196).

4.2.2 Rollen bei Scrum

Scrum definiert drei Rollen mit klar voneinander abgegrenzten Verantwortungsbereichen (Vgl. [Hans10], S. 62-66):

Product Owner:

Diese Rolle übernimmt vielerlei Aufgaben. Für den Kunden ist sie die Schnittstelle zu dem Entwicklungsteam, um seine Interessen und Wünsche durchzusetzen. Auf der Seite des Arbeitnehmers ist sie ein Teil des Entwicklungsteams, verantwortlich für strategische und wirtschaftliche Entscheidungen für das Projekt. Am Anfang erstellt der Product Owner eine *Produktvision*, sammelt in einem *Product Backlog* die von dem Kunden gewünschten Anforderungen an das System, priorisiert sie, bestimmt die Abnahmekriterien und den Auslieferungszeitpunkt für die Softwareinkremente.

Entwicklungsteam:

Das Team ist für die selbstorganisierende Planung samt Umsetzung der Sprint-Ziele verantwortlich. Ein Sprint entspricht einer Iteration von wenigen Wochen. Die Zusammensetzung sollte gut durchdacht sein und einen Vertreter aus allen Spezialgebieten beinhalten. Die Anzahl der Personen sollte zwischen fünf und neun liegen. Die räumliche Nähe ist vorausgesetzt.

ScrumMaster:

Diese Rolle sorgt für die Einhaltung der Scrum-Prozesse durch das Team. Sie schützt es vor störenden Eingriffen während eines Sprints und fungiert als Berater und Vermittler. Bei Meetings übernimmt sie die Rolle des Moderators. Je erfahrener ein Team mit Scrum ist, desto seltener ist die Anwesenheit des ScrumMasters erforderlich.

4.2.3 Scrum-Artefakte

Zu den Artefakten in Scrum zählen Dokumente[9], die der Planung, der Information oder der Überwachung dienen. Das sind:

Produktvision:

Der Product Owner verfasst am Anfang eines Projekts eine Vision. Das ist eine begeisterungsfähige Ideenbeschreibung um die Projektbeteiligten zu motivieren und ein Ziel vorzugeben.

Product Backlog:

Dieses Dokument dient der Planung. Das beinhaltet alle für das Projekt bis dahin geforderten Funktionalitäten. Der Product Owner sammelt diese, priorisiert sie, hält sie auf dem aktuellen Stand und schätzt für jede dieser Anforderungen den Aufwand für die Umsetzung.

Sprint Backlog:

Dieses täglich zu aktualisierende Backlog ist ein Sprint-Planungsinstrument für das Team. Nachdem Definieren des Sprintziels zerteilt das Team die anstehenden Aufgaben in Arbeitspakete. Diese werden nach Aufwand in Stunden geschätzt und gleich einer Person zugewiesen.

Impediment Backlog:

Das ist eine vom ScrumMaster geführte und abzuarbeitende Liste mit priorisierten Hindernissen (engl. Impediments), die das Entwicklungsteam blockieren oder an effektiver Arbeit hindern (Vgl. [Glog09], S. 87).

[9] Die Dokumente liegen meistens in digitaler Form vor.

Releaseplan und Burndown Chart:

Zur visuellen Überwachung meist größerer Projekte erstellt der Product Owner das in Abbildung 6 gezeigte Diagramm. Der Verlauf stellt die Restaufwände in Tagen verteilt über die Sprints dar. Auf diese Weise kann jeder einschätzen, wann das Projekt endet. Solche Diagramme können z.B. auch Teams einsetzen, um die Restaufwände in einem Sprint zu visualisieren. Die *Velocity* stellt außerdem die Entwicklungsgeschwindigkeit pro Sprint zum Nachverfolgen dar.

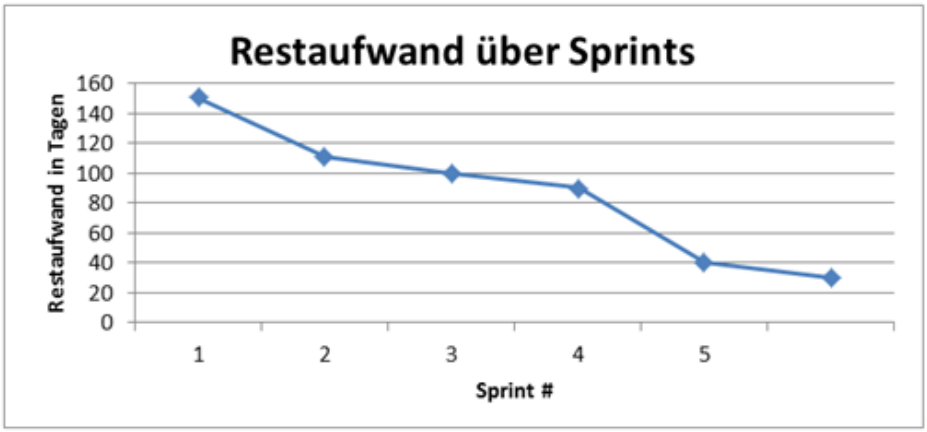

Abbildung 6: Burndown Chart eines Releases

4.2.4 Der Ablauf

Am Anfang eines Projekts erstellt der Product Owner eine Vision des Endprodukts, samt einer ersten Einschätzung des Aufwands und der Umsetzungsdauer. Nach diesen Vorarbeiten, falls entschieden wird das Projekt in Angriff zu nehmen, sammelt er die Anforderungen in das Product Backlog und priorisiert sie.

Das Projekt wird in Sprints durchgeführt, die Iterationen von je 30 Tagen entsprechen. Als nächstes wird eine Sprint-Planungssitzung einberufen. In dieser Sitzung beschließt das Team, wie viele der am höchsten priorisierten Anforderungen aus dem Product Backlog in dem nächsten Sprint realisiert werden müssen. Als Ergebnis dieser Sitzung entsteht ein Sprint Backlog, indem die zu erledigende Arbeiten, als Arbeitspakete zusammengefasst, mit der geschätzten Bearbeitungsdauer aufgelistet sind. Danach kann der Sprint beginnen. In dieser Phase darf das Team nicht von außen unterbrochen werden. Täglich findet ein Daily Scrum statt. Das ist ein 15 minutiges Statusmeeting für die Beteiligten, das im Stehen abgehalten wird. In dem Sprint Review wird das entwickelte Softwareinkrement dem Product Owner präsentiert, der über die Abnahmebedingungen entscheidet. In der anschließenden Sprint-Retrospektive hat das Team die Möglichkeit über den Sprint zu reflektieren, Verbesserungsmaßnahmen zu beschließen und daraus zu

lernen. Nach diesem Ablauf werden die anderen Sprints bis zum Abschluss des Projekts durchgeführt.

Abbildung 7 gibt eine Übersicht über den Scrum-Prozess.

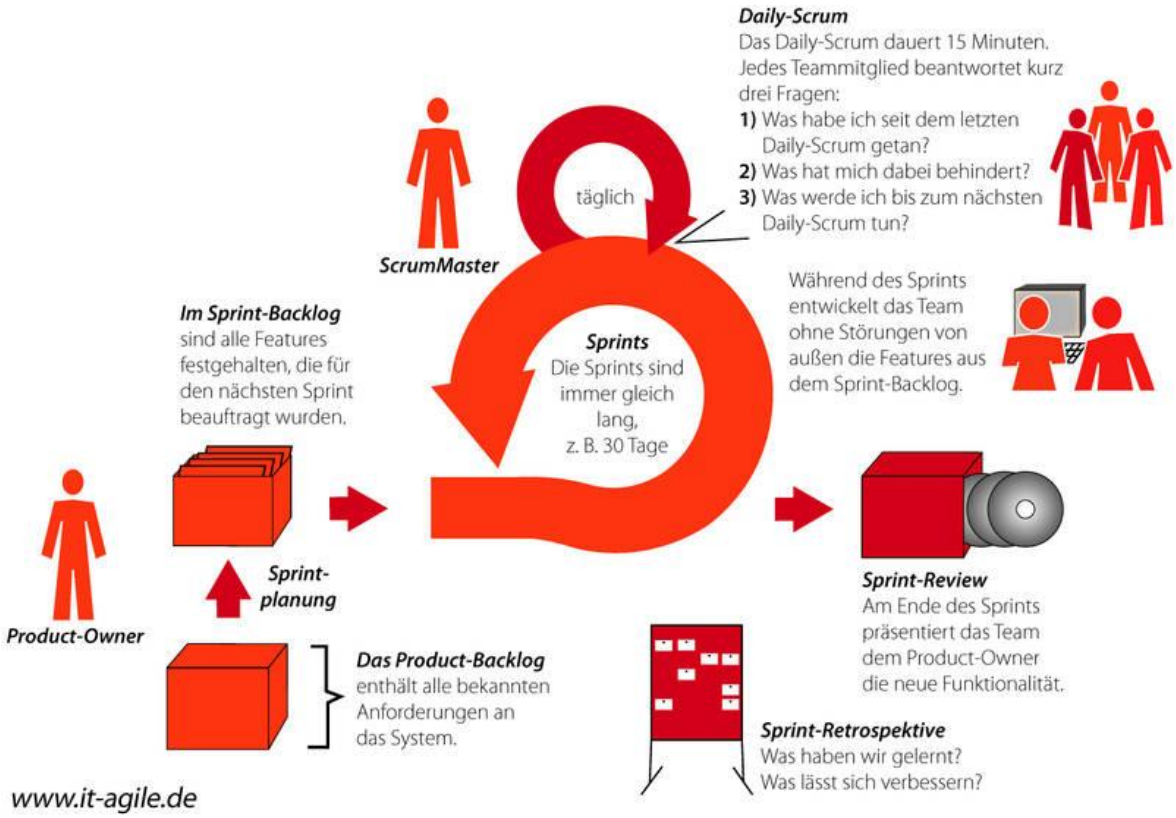

Abbildung 7: Scrum-Prozess in Kürze[10]

4.2.5 Bewertung von Scrum

4.2.5.1 Einarbeitungszeit

„Scrum ist kein Wundermittel“ ([Pich08], S. 2). Es ist ein empirischer Prozess, der das Team zum selbstständigen kreativen Arbeiten anregt und daraus das Lernen ermöglicht. Nach der Einführung von Scrum verschwinden die bestehenden Probleme nicht von alleine, es kommen sogar welche dazu. Gerade die ersten Sprints stellen eine Herausforderung für das Management und für das Projektteam da. Neue Abläufe müssen beherrscht, alte vergessen werden. Das unerfahrene Team müsse laut M. Cohn für sich herausfinden, wie viel Arbeit es pro Sprint erledigen könne, wie ausführlich das Product Backlog zu gestalten sei, wie es sich überhaupt selbst organisieren könne. Der ScrumMaster sei in dieser Phase besonders gefordert. Er müsse sicherstellen, dass die

[10] Quelle: http://www.it-agile.de/scrum.html (Feb. 2012).

Beteiligten ihre gewohnten Praktiken verändern und nicht die Scrum-Praktiken diesen anpassen. Wenn das Team an Erfahrung gewinne und schneller in der Arbeitsweise würde, werde der ScrumMaster weniger, dafür der Product Owner mehr benötigt. M. Cohn empfiehlt darauf bei mehreren Scrum-Teams einen ScrumMaster für zwei bis drei Teams heranzuziehen, wobei einem Product Owner nicht mehr als zwei Teams zuzuweisen (Vgl. [Cohn10], S. 156-157).

Scrum ist einfach zu verstehen, jedoch extrem schwer zu meistern (Vgl. [ScrG11], S. 3). Die Einarbeitungszeit bei Scrum ist mit ***lange*** bewertet.

4.2.5.2 Transparenz

Scrum bietet viele Instrumente zur Präsentation der aktuellen Informationen. Die Backlogs bieten dem Team gute Möglichkeit der permanenten Übersicht. Die gesamten aktuellen Anforderungen an das Produkt werden in dem Product Backlog, die festen anstehenden Aufgaben für einen Sprint in dem Sprint Backlog aufgelistet. Insofern werden diese gut einsehbar entweder in digitaler Form auf einem großen Bildschirm, oder auf Zetteln mit kurzer Beschreibung auf einer Pinnwand bekannt gegeben. Kennzahlen der Velocity und die Burndown Charts geben wichtige Informationen über den Projektverlauf und unterstützen die wirklichkeitsnahe Planung des Projekts. Durch diese Mittel kann das Team die Fortschritte realistisch einschätzen und dadurch eventuell zum Weiterarbeiten motiviert werden.

Der Fortschritt des Projekts wird außerdem an der kurzen Sprintdauer von ca. 30 Tagen schnell sichtbar. Innerhalb dieser Zeit werden alle Entwicklungsschritte ausgeführt und das funktionierende Softwareinkrement dem Product Owner vorgeführt. Dieser fungiert als eine Schnittstelle zwischen dem Kunden und dem Entwicklungsteam. Er präsentiert die Ergebnisse dem Kunden und sorgt dafür, dass sein Feedback in dem weiteren Projektverlauf berücksichtigt wird.

Aus Daily Scrum, dem Sprint Review und der Retrospektive können die Beteiligten weitere zahlreiche Informationen erhalten. Alle Betroffenen werden darüber informiert, was in den Sprints und auch im Projekt gut gelaufen ist, aber auch was aktuell Probleme bereitet.

Die zahlreichen Maßnahmen von Scrum ermöglichen eine ***sehr gute*** Transparenz dieses Vorgehensmodells.

4.2.5.3 Skalierbarkeit und verteiltes Arbeiten

Ken Schwaber und Jeff Sutherland, die Entwickler von Scrum, raten in [ScrG11] vor zu kleinen Teams ab. Demnach könne bei drei Teammitgliedern die Gefahr bestehen, dass nicht alle notwendigen Fähigkeiten für die Entwicklung einer Software vorhanden seien. Bei Teams über neuen Personen, werde ihrer Erfahrung nach die Koordination zu komplex.

M. Cohn stimme der allgemein anerkannten Größe für Scrum-Teams von fünf bis neun Personen zu, wovon er anschließend zahlreiche Vorteile auflistet. Allerdings bevorzuge er die Ansicht von „Zwei-Pizza-Team". Demnach solle ein Team so groß gewählt sein, dass für ein gemeinsames Essen zwei Pizzen ausreichen würden. Auf diese Weise sei eine effiziente Kommunikation zwischen den Teammitgliedern ermöglicht. Cohn habe schon Erfahrung mit einem Team von 14 Personen, aber 25 Personen seien, seiner Meinung nach, wegen dem zu hohen Kommunikationsaufwand nicht mehr zu empfehlen ([Cohn10], S. 208). Des Weiteren erklärt Cohn: „Die verteilte Entwicklung kann funktionieren, aber ein verteiltes Team wird niemals die gleiche Leistung erbringen wie ein lokales." Er führt eine Reihe von Techniken auf, mit denen man die Leistung eines verteilten Teams sehr nah an die eines lokalen bringen könne. Unter anderem seien persönliche Treffen und das Erschaffen eines gemeinsamen Verantwortungsgefühls die Schlüsselfaktoren und für ein verteiltes Projekt unverzichtbar ([Cohn10], S. 416).

B. Gloger beschäftigt sich näher mit dem Thema der Skalierbarkeit von Scrum in großen Projekten. Als allererstes gibt er den Rat: „Don´t do it!"[11]. Die Skalierung auf große Teams könne viele negative Auswirkungen auf das Projekt haben. Dazu zählen die Verlängerung des Projekts, die Erschwerung der Arbeitsweise und die Verringerung der Produktivität des Einzelnen und somit des Gesamtteams. Das seien Gründe, wieso große Teams oft erfolgslos aufgeben ([Glog09], S. 231).

Des Weiteren gibt B. Gloger ein Skalierungsmodell vor, das die Schwierigkeit der Kommunikation in einem großen Team weitestgehend kompensiere. Dazu seien drei zusätzliche Meetings notwendig: *Scrum of Scrums*, *Product Owner Daily Scrum* und *Weekly Scrum*. Die verschiedenen Anforderungen aus dem gemeinsamen Product Backlog würden von den verteilten Teams unabhängigen nach allen Scrum-Regeln bearbeitet. Die Synchronisation der Aufgaben und somit die erforderliche Kommunikation erfolge in den Meetings durch ausgewählte Vertreter jedes Teams, die nachfolgend die Informationen an seine Teammitglieder weiter trügen ([Glog09], S. 237). Bei einer größeren Anzahl an Teams und Personen wären, nach U. Kapp und J. P. Berchez unter [Kapp10], die festen und geordneten Prozessregeln umso wichtiger. Durch solche Maßnahmen ließe sich die verteilte Produktentwicklung mit Erfolg bewältigen.

[11] Frei übersetzt: „Mach es ja nicht!".

Jeff Sutherland beschreibt ein erfolgreiches, auf drei Standorte verteiltes Projekt mit 56 Entwicklern. Er leitet daraus folgende fünf Best Practices für verteilte Projekte ab: 1) tägliche Daily Scrum Meetings für alle Teams an allen Standorten; 2) tägliche Sitzungen des Product Owner Teams; 3) stündliche automatisierte Builds auf einem zentralen Repository; 4) keine Unterscheidung zwischen den Entwicklern der verschiedenen Standorte und dem eigenen Team; 5) nahtlose Integration der XP-Praktiken wie Pair-Programming in Scrum (Vgl. [Suth11], S. 96).

Die Skalierbarkeit und verteiltes Arbeiten ist demnach bei Scrum ermöglicht und ist wegen oben aufgeführten Argumente mit ***gut*** bewertet.

4.2.5.4 Dokumentationsaufwand

An der Menge der Scrum-Artefakte die gefüllt und gepflegt werden müssen wird deutlich, wie viel Aufwand für die Dokumentation notwendig ist. Obwohl das als ein Verstoß gegen die agilen Werte ausgelegt werden könnte, ist dieses Vorgehen für Scrum aus vielen Gründen wie zur Planung, Information und zur Überwachung unverzichtbar.

Der Aufwand für Dokumentation im Vergleich zu den anderen untersuchten Vorgehensmodellen ist als ***groß*** eingestuft.

4.2.5.5 Qualitätssicherungsmaßnahmen

Bei der Zusammenstellung des Teams empfiehlt J. Sutherland Mitglieder mit allen für das Produkt relevanten Fähigkeiten zu wählen, d.h. sog. Cross-funktionale Teams zu bilden. Dazu gehört ebenso ein Entwickler mit ausgeprägten Tester-Kenntnissen für die Sicherstellung der Qualität der Software. Scrum fördere diese Rolle nicht explizit, aber am Ende eines Sprints müsse, nach K. Schwaber und J. Sutherland, ein ausgiebig *getestetes* Stück Software mit anderen Inkrementen funktionieren und dem Product Owner präsentiert werden ([Suth11], S. 17).

Der spezialisierte Tester bekommt laut B. Gloger kein *vollständiges* Anforderungsdokument, woraus er die Testfälle ableiten könne. Denn das sei mit den agilen Methoden nicht praktikabel. Nach Scrum müsse er genauso wie die Entwickler „sich daran gewöhnen, häufigere und bedeutungsvollere Gespräche mit ihren Mitarbeitern und oftmals mit Personen außerhalb des Teams zu führen“ ([Glog09], S. 177).

Falls die oben beschriebenen Scrum-Praktiken eingesetzt werden, schreiben und automatisieren die Entwickler die Tests selbst. Der anschließend geschriebene Code muss

diese Tests erfolgreich durchlaufen und wird mehrmals am Tag in das Gesamtsystem integriert.

Demnach sieht Scrum vor, ausgereifte Softwareinkremente am Ende jeder Iteration auszuliefern. Somit gehört das ausgiebige Testen zum festen Bestandteil jeden Sprints, das die Qualitätssicherungsmaßnahmen dieses Vorgehensmodells mit ***sehr gut*** bewerten lässt.

4.2.5.6 Flexibilität bei Anforderungsänderungen

Der Product Owner ist der Herr über das Product Backlog. Er kann während eines Sprints neue Anforderungen sammeln und proirisieren. Diese darf er dem Team aber erst bei der nächsten Sprintplanung bzw. zwischen den Sprints vorstellen. In einem Sprint darf das Team mit keinen neuen Aufgaben unterbrochen werden. Dafür sorgt der ScrumMaster. Bei neu dazu gekommenen Anforderungen muss der Releaseplan im nächsten Sprint entsprechend angepasst werden.

Die Reaktionsdauer für die Berücksichtigung einer neuen Anforderung liegt bei 30 Tagen, das einer Sprintlänge entspricht. Die Flexibilität ist somit für alle Parteien gegeben und ermöglicht die Realisierung einer neuen und am höchsten priorisierten Anforderung innerhalb einer kurzen Zeit. Dieser Untersuchungspunkt ist demnach mit ***sehr gut*** bewertet.

4.2.5.7 Kundeninvolvierung

Der Product Owner ist ein Mitarbeiter des Arbeitnehmers und dient als Vertreter des Kunden. Er lenkt unter anderem das Team bezüglich der Produktentwicklung und arbeitet mit ihm eng zusammen. Er muss dafür sorgen, dass die Wünsche und Anregungen des Kunden Beachtung finden, hat jedoch als primäres Ziel die wirtschaftlichen Interessen seines Arbeitgebers. In regelmäßigen Workshops mit dem Kunden und den späteren Anwendern der Software ermittelt er die Anforderungen und holt von dem Kunden das Feedback ein (Vgl. [Pich08], S. 10).

Der Kunde ist am Entwicklungsprozess nicht direkt beteiligt, der Zwischenschritt über den Produkt Owner stellt jedoch keinen Nachteil für die beiden Parteien des Arbeitgebers und des Arbeitnehmers dar. Die Kundeninvolvierung bei Scrum ist demnach ***stark***.

4.2.6 Fazit

Viele Unternehmen scheitern bei der Einführung von Scrum weil sie glauben, die Änderungen würden sich nur auf die Entwicklungsabteilung auswirken. Die Umstellung betrifft jedoch nicht nur die Entwickler. Wenn die anderen relevanten Teilbereiche nicht ebenfalls umgestaltet werden, steht man nach kurzer Zeit wieder an Anfang. Der Umstieg ist trotz dem die Mühe wert, denn wenn man es richtig macht, erwarten einen kürzere Entwicklungszeiten - was auch den Kunden zufriedener macht, niedrigere Kosten und höhere Qualität des Produkts (Vgl. [Cohn10], S. 31-32).

Die feste Releaseplanung in einem agilen Projekt sei laut E. Hanser kein Wiederspruch zu den agilen Werten[12] [Mani01]. Denn in der nächsten Sprintplanung würden die neuen Anforderungen mit berücksichtigt und dementsprechend auch der Releaseplan neu angepasst werden ([Hans10], S. 77).

Scrum könne nach B. Gloger trotz des anfänglichen Abratens durchaus für große Projekte eingesetzt werden. Die verteilten Teams sollen dafür weitere Maßnahmen zur Synchronisation ergreifen und die empfohlene Größe von ca. 7 Personen nicht überschreiten ([Glog09], S. 231).

Das Framework hat somit wirkungsvolle Mechanismen, um auf die Wünsche der Kunden flexibel zu reagieren. Die hohe Transparenz sorgt dafür, dass die Anwender die neuen Abläufe akzeptieren, das Pull-Prinzip zur Planung des Sprintziels verhindert die Überlastung des Teams und die zahlreichen weiteren Maßnahmen ermöglichen eine hohe Qualität in kurzer Zeit zu liefern. Das Vorgehen lässt sich auf jegliche Art von Softwareentwicklungsprojekten anwenden.

Abschließendes Wort geht an die Erfinder von Scrum Jeff Sutherland und Ken Schwaber: „Obwohl es möglich ist, nur Teile von Scrum zu implementieren, ist das Ergebnis nicht Scrum.“[13]

[12] Vgl. der Wert: „Reagieren auf Veränderung mehr als das Befolgen eines Plans“.
[13] Aus der deutschen Übersetzung des *Scrum Guide* der benannten Autoren ([ScrG11], S. 17).

4.3 Crystal-Familie

Alistair Cockburn, einer der Verfasser des agilen Manifests, entwickelte 1990 anhand von zahlreichen Erfahrungen aus früheren Projekten die Crystal-Familie agiler Prozessmodelle. Das ist eine Ansammlung von Methoden, die Projekte nach Teamgröße und nach Konsequenzen eines Ausfalls des Systems, sogenannte Kritikalität, unterteilt und entsprechend der Projektart unterschiedliche agile Methoden vorgibt. Im Mittelpunkt stehen die Menschen und somit das Team, das gemeinsam plant und selbstgesteuert ohne einen direkten Projektleiter handelt. Das Team verfügt über spezielle Fähigkeiten und Erfahrungen, denen sich der Prozess anpassen muss und nicht umgekehrt. Nach dem Motto „Gerade eben ausreichend“ ist eine minimale Anzahl an Rollen, definierten Aktivitäten, definierten Ergebnissen und somit an Formalität für jedes Projekt individuell vorgegeben. Die Abbildung 8 zeigt die farbig angeordnete Einteilung der Projektarten nach diesen Kriterien. Je dunkler der Farbton ist (von links nach rechts), umso mehr Personen sind an der Entwicklung beteiligt, umso schwieriger ist die Koordination und die Planung und umso schwergewichtiger ist die agile Methode (Vgl. [Buns08], S. 131).

Der Name „Crystal“ heißt übersetzt „das Kristall“. Einerseits bezieht sich die Größe des Teams auf die unterschiedlichen Farbtöne des Kristalls, von glasklarem bis zum schwarzen Edelstein. Andererseits stehen die Kritikalitätsstufen für die Härtegrade des Kristalls, vom weichen Quarz bis zum härtesten Diamanten (Vgl. [Hrus09], S. 55). Bezogen auf die Softwareentwicklung gibt Crystal-Methodenfamilie für jede nach Farbe geordnete spezielle Rahmenbedingungen eines Projekts entsprechende Techniken, Rollen, Werkzeuge und Standards vor. Das kann zufolge haben, dass bei einfachen und kleinen Projekten auch die agilen Methoden eingesetzt werden, die aus anderen Vorgehensmodellen wie Extreme Programming bekannt sind. Die Unterteilungen der Crystal-Methodenfamilie lauten (von links nach rechts): Crystal Clear, Yellow, Orange, Orange Web, Red, Magenta, Blue. Es sind jedoch nur Crystal Clear, Crystal Orange und Crystal Orange Web vollständig definiert (Vgl. [Buns08], S. 131-132).

Im Folgenden wird hauptsächlich die Variante Crystal Clear näher betrachtet. An einigen Stellen werden die Eigenschaften von Crystal Orange ergänzend angegeben. Manche der Praktiken, die bei Crystal einsetzbar sind, werden im weiteren Verlauf näher beleuchtet.

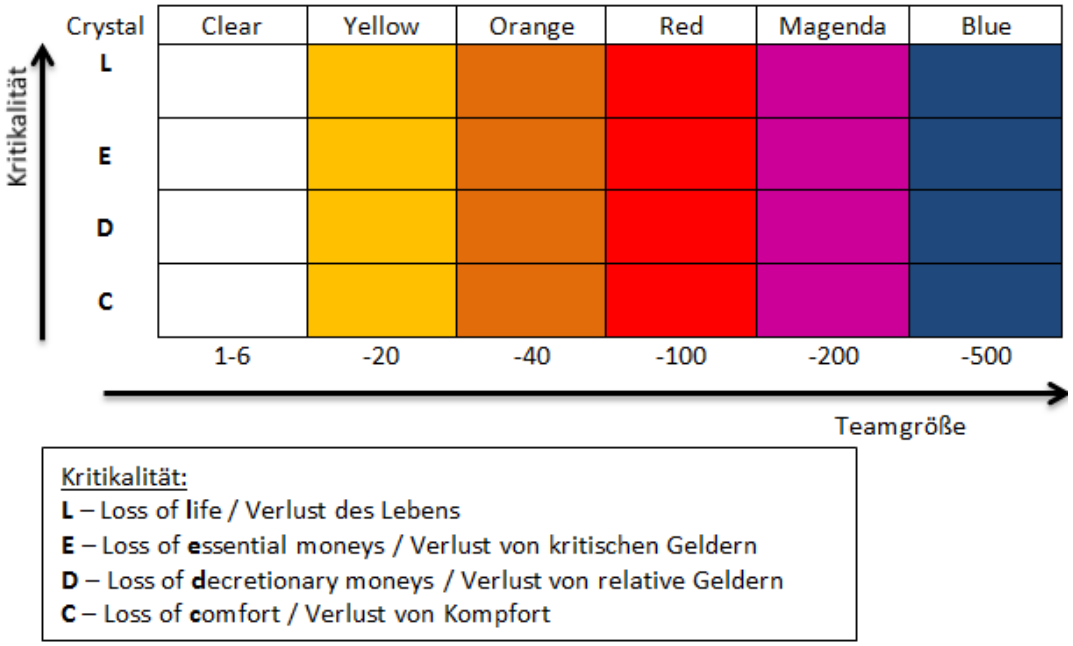

Abbildung 8: Einteilung der Crystal-Familie nach A. Cockburn

4.3.1 Die sieben Crystal-Eigenschaften

Folgende Projekteigenschaften wurden nach Umfragen von vielen Teammitgliedern als erfolgreich eingestuft und sind für Crystal-Projekte vorgeschrieben ([Hans10], S. 49-56):

Regelmäßige Lieferung:

Der Kunde bekommt in regelmäßigen Abständen das ausgereifte Softwareinkrement. Anhand dessen kann er die Erfüllung seiner Anforderungen überprüfen und das Entwicklungsteam mit Feedback versorgen. Die maximale Iterationsdauer für die Crystal-Methodenfamilie beträgt vier Monate.

Reflektierte Verbesserung:

In periodisch stattfindenden Reflexionsbesprechungen diskutiert das Team über die Erfolge und Misserfolge der im Verlauf des Projekts eingesetzten Praktiken. Die Sitzung ist mindestens alle drei Monate abzuhalten und sie darf von ca. 30 Minuten bis vier Stunden in Anspruch nehmen.

Verdichtete oder osmotische Kommunikation:

Ein Team kann bei kurzen Kommunikationswegen, z.B. per Videokonferenz oder Telefon, *verdichtet* kommunizieren. Dabei kann aber eine gewisse Kommunikationsreduktion stattfinden, denn am Telefon erwähnt man vielleicht manche Sachen nicht, die man bei einem persönlichen Gespräch ansprechen würde. *Osmotische Kommunikation* ist eine wesentlich kostengünstigere und effizientere Alternative, die bei kleinen sich in einem Raum befindlichen Teams im

„Vorbeigehen“ erfolgt. Ohne die direkte Beteiligung an einer Unterhaltung bekommen die Tischnachbarn Informationen von ihren Teammitgliedern. Die osmotische Kommunikation wird bei Crystal Clear-Projekten mit einer Teamgröße von sechs Personen eingesetzt.

Persönliche Sicherheit:

Jedes Teammitglied darf stets offen die Wahrheit über die projektspezifischen Themen äußern, ohne Konsequenzen befürchten zu müssen. Das trifft besonders dann zu, falls eine Planung als unrealistisch erscheint oder persönliche Wissenslücken bestehen. Das gehört zu einer vertrauensvollen Arbeitsatmosphäre und kann ein Projekt beschleunigen.

Schwerpunkte bilden:

Jedes Teammitglied übernimmt je nach seinen Stärken und Vorlieben die Verantwortung für zwei vom Management gebildete Schwerpunkte. Das sind priorisierte Aufgaben, die für das Unternehmen oder das Projekt als wichtig erscheinen. Diese Aufgaben werden zu seinen Hauptschwerpunkten, für die er Zeit für die Bearbeitung bekommt. Auf diese Weise sollen die Teammitglieder ihre Tätigkeiten für verschiedene Projekte nach ihren Hauptschwerpunkten priorisieren und denen größere Beachtung schenken.

Einfache Kontaktaufnahme mit Endanwendern:

Mit dieser Eigenschaft wird sichergestellt, dass der Kunde das bekommt, was er sich wünschte. Das schnelle Feedback durch die regelmäßigen Auslieferungen der Softwareinkremente ermöglicht eine zeitnahe Reaktion auf die Änderungen der Kundenwünsche.

Technische Umgebung mit automatisierten Tests, Konfigurationsmanagement und regelmäßigen Integrationen:

Automatisierte Tests lassen sich unkompliziert wiederholen. Eine zuverlässige Integration frisch erstellten Codes muss zweimal die Woche erfolgen und braucht ein werkzeuggestütztes Konfigurationsmanagement.

Die regelmäßige Lieferung, reflektierte Verbesserung und osmotische Kommunikation zählen zu den wichtigsten Eigenschaften für Crystal-Projekte.

4.3.2 Rollen bei Crystal

Crystal kennt viele Rollen, die mit der steigenden Mitgliederzahl, mit größerer Komplexität und mit größerem Umfang des Projekts dazukommen. Die ersten vier zählen

zu den wichtigsten Projektrollen und nur diese müssen von verschiedenen Personen belegt werden. Die folgenden acht Rollen seien, laut E. Hanser, ausreichend für Crystal Clear ([Hans10], S. 56-58). Diese würden lauten:[14]

Auftraggeber / Sponsor:

Er wird auch als Kunde bezeichnet. Er finanziert das Projekt und gibt die Ziele vor. Ihm unterliegen ökonomische und betriebswirtschaftliche Entscheidungen, auch die, über eine mögliche Aufgabe des Projekts.

Erfahrener Anwender / Anwendungsexperte:

Er kommt aus der Umgebung der späteren Anwender. Er weiß was an dem Vorgängersystem schlecht war und was verbessert werden soll. Er entscheidet über die Modellierung der Anwendungsoberfläche (GUI) und steht dem Entwicklungsteam in ausreichendem Umfang zur Verfügung.

Chefdesigner / Lead Designer/Programmierer:

Diese Rolle wird von einem erfahrenen Softwareentwickler und Designer übernommen, der für das gesamte Systemdesign verantwortlich ist. Die Person hat gleichzeitig die Verantwortung eines Projektleiters für Steuerung und Planung des Projekts.

Designer/Programmierer:

Die zwei Rollen des Designers und des Programmierers gehören nach Crystal zusammen. Ein Programmierer muss designen und ein Designer muss programmieren können. Durch die Vereinigung sollen die möglichen Missverständnisse bei der Kommunikation zwischen diesen zwei Rollen vermieden werden.

Koordinator:

Die Person in dieser Rolle übernimmt die Koordination des Projekts, erfasst seine Kennzahlen und verbreitet sie an alle Beteiligten. In der Regel fungiert der Chefdesigner zusätzlich zu seinen Aufgaben in dieser Rolle. Zu den typischen Aufgaben gehört die Erstellung des Releaseplans und der Meilensteine, das Verfassen der Berichte über den Projektstatus und die Führung einer priorisierten Risikoliste.

Fachexperte / Geschäftsexperte:

Diese Rolle übernimmt zusätzlich zu seinen Aufgaben ein *erfahrener Anwender*, der über die Kenntnisse über die Geschäftsabläufe und das spätere Umfeld des Systems verfügt. Er muss den Entwicklern den fachlichen Hintergrund des Projekts vermitteln und ausreichend oft für die Fragen zur Verfügung stehen.

[14] Die alternativen Bezeichnungen der Rollen stammen aus [Stey10], S. 27-28.

Tester:

Den geschriebenen Code soll entweder ein externer Tester überprüfen oder die Entwickler testen ihren Code gegenseitig. Keiner soll seinen Code selbst testen, denn oft steht man seinen eigenen Fehlern blind gegenüber.

Autor / Schreiber:

Der Autor kann ein spezialisiertes Teammitglied oder ein externer Mitarbeiter sein. Er schreibt die externen Anwenderdokumente wie die Benutzerhandbücher oder auch die Abnahmetests.

P. Hruschka zähle, abweichend von E. Hanser, zu den notwendigen Rollen für die Crystal Clear-Projekte nur die vier wichtigsten. Das wären der Sponsor, der Anwender, der Senior Designer/Programmierer und der Designer/Programmierer ([Hrus09], S. 56).

4.3.3 Der Ablauf

Der Ablauf des Entwicklungsprozesses richtet sich nach der eingesetzten Methode, den Praktiken und den beteiligten Rollen. Es könnte sich im Folgenden beschriebener und in Abbildung 9 dargestellte Ablauf ergeben.[15]

Am Anfang eines Projekts müssen *Use Cases* gesammelt werden, die nach bestimmten Vorgaben der Erfassung von Anforderungen dienen. Das geschieht z.B. durch Interviews und Meetings mit dem Kunden. Sie sind in der Sprache und aus der Sicht des Kunden bzw. der späteren Anwendergruppen verfasst. Use Cases beschreiben anhand von Szenarien die möglichen Interaktionen der Benutzer (Akteure) mit dem System. Zunächst werden die reibungslosen Abläufe erfasst (*Happy-Day-Szenario*), die anschließend durch denkbare, vom unproblematischen Ablauf abweichende Szenarien ergänzt werden (Vgl. [Stey10], S. 77).

Zum Planen schlägt Crystal die Methodik *Blitz-Planning* vor. Demnach identifizieren die Beteiligten in einem Meeting alle aus den Use Cases resultierenden Aufgaben und schreiben diese auf Kärtchen. Die verschiedenen Kärtchen gilt es in der Reihenfolge und nach der Möglichkeit der Parallelisierung anzuordnen. Die Person, deren Schwerpunkt eine Aufgabe darstellt, der ein oder i.d.R. mehrere Kärtchen zugewiesen sind, schätzt mit Einverständnis der anderen den Aufwand für die Erledigung der Aufgaben. Dann wird der Grad der Parallelität und die gleichmäßige Verteilung der Verpflichtungen erneut optimiert. Als nächstes erfolgt die Zuordnung der Aufgaben zu Releases und Iterationen, wobei die von dem Kunden als hoch priorisierten, aus der Sicht des Teams als risikobehaftet identifizierten, in der Bearbeitungsliste weiter nach oben geschoben werden (Vgl. [Stey10], S. 43). Diese Art der Anforderungsverwaltung wird bei Crystal auch

[15] Erst Crystal Orange definiert eine systematische Entwicklung der Inkremente.

Iceberg List genannt. Ein Eisberg als Metapher suggeriert, dass die Aufgaben für die nächste Iteration die aus dem Wasser ragende Spitze eines Eisbergs repräsentiert und alles andere bleibt noch „unter Wasser“ (Vgl. [Stey10], S. 70).

Nach der Planung und nach Verteilung der Aufgaben erledigen die Teammitglieder im Rahmen der Iteration jeden seiner Schwerpunkte. Die Seite des Arbeitgebers wird bei Fragen herangezogen und außerdem mit zwei Reviews zur Überprüfung und für Feedback in das Projekt involviert. In regelmäßigen Abständen, z.B. wöchentlich, werden Teammeetings durchgeführt. Darin werden unter anderem auch die neuen Anforderungen des Kunden aufgenommen, bewertet und entsprechend verteilt. Die abschließende Codeintegration samt den automatisierten Tests beendet die Iteration.

Nach jeder Iteration führt das Team einen *Reflection Workshop* durch. Die Ergebnisse und Beschlüsse werden nach dem Muster *Informative Workspace* für alle Teammitglieder sichtbar ausgehängt. Die daraus entstandenen Verbesserungen müssen in den nächsten Iterationen umgesetzt werden. Ein Softwareinkrement umfasst in der Regel mehrere Iterationen (Vgl. [Stey10], S. 52).

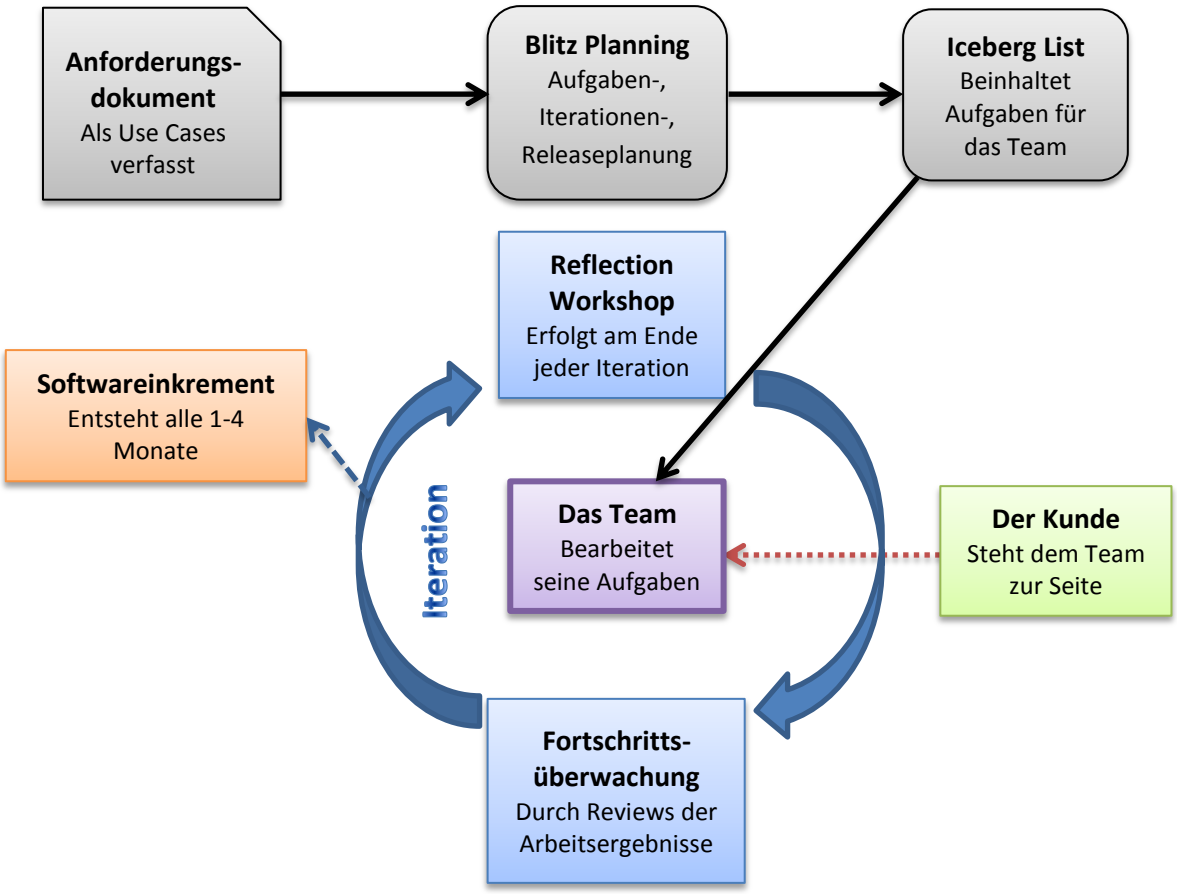

Abbildung 9: Crystal-Ablauf

4.3.4 Bewertung der Crystal-Familie

4.3.4.1 Einarbeitungszeit

Crystal bietet eine Vielfalt an Methoden, die sich auf beinah jede Projektart zugeschnitten anwenden lassen. Jede dieser Methoden kann individuell angepasst und eventuell mit anderen agilen Praktiken ergänzt werden. Beispielswese wäre es möglich, ein Unternehmen, das nach Extreme Programming vorgeht, an die Crystal-Methodik anzupassen und manche der XP-Praktiken dort weiter anzuwenden. Dadurch lässt es sich in viele Unternehmen integrieren und kann sich als sehr lukrativ erweisen.

Das Team agiert selbstorganisierend und entscheidet letztendlich selbst, welche Praktiken es einsetzt. Diese Entscheidung kann durch die vorgesehenen Reflexionsworkshops infrage gestellt und angepasst werden. Die Teammitglieder können gleichzeitig in mehreren Rollen und in mehreren Teams eingesetzt werden. Das könnte einen Umstand darstellen, an den alle sich erst gewöhnen müssen.

Die Einarbeitungszeit bei Crystal-Methodenfamilie ist demnach mit ***kurz*** bewertet.

4.3.4.2 Transparenz

Für ein kleines Team von drei bis sechs Mitgliedern, die alle in einem Raum arbeiten, sind keine besonderen Visualisierungsmittel notwendig. Die direkte und unkomplizierte Kommunikation in einem solch kleinen Team lassen den Stand der aktuellen Iteration oder des gesamten Projekts, der erledigten und noch zu erledigenden Aufgaben, leicht erfahren. Crystal Clear sieht trotzdem vor, auf einer Tafel oder einem Whiteboard die wichtigsten Informationen auszuhängen. Dazu können die Anforderungen, eine Designskizze und ein Releaseplan gehören. Bei Projekten mit mehreren Beteiligten bleiben die Teams genauso klein. Die einfachen Visualisierungsmittel der Informationen, die nur für den Aufgabenbereich dieses Teams wichtig sind, bleiben bestehen. Für die Veranschaulichung der teamübergreifenden Informationen können weitere Maßnahmen, wie z.B. digitale Hilfsmittel, ihren Einsatz finden.

Der Projektfortschritt würde, laut C. Bunse, bei Crystal durch Arbeitsergebnisse und durch Stabilität des Systems gemessen. Die Bestandteile der Arbeitsergebnisse seien: *Start*, *Review 1*, *Review 2*, *Test*, *fertige Softwareversion*. Die Kategorien *starke Änderungen, Änderungen, stabil genug für das Review* würden die Stabilität des System beschreiben. Um Transparenz für den Kunden herzustellen seien bei Crystal Orange für jedes Softwareinkrement drei Begutachtungen durch diesen vorgesehen. Zum einen würden Kundeninterviews und Teambesprechungen durchgeführt, um die Art des Projekts zu bestimmen und dafür die Methoden auszuwählen. Vor jeder Entwicklungsphase, danach und am besten auch zwischendurch, würden zum anderen Gruppenbesprechungen

abgehalten. Dieses gebe dem Kunden Einsicht in die Entwicklung seiner Software und stelle sicher, dass es sich in die von ihm vorgesehene Richtung bewege ([Buns08], S. 135).

Die Transparenz ist bei Crystal im Sinne dieses Untersuchungspunkt ***sehr gut*** gegeben.

4.3.4.3 Skalierbarkeit und verteiltes Arbeiten

Für Crystal Clear-Projekte darf das Team aus höchstens *sechs* Personen bestehen. Die Teammitglieder arbeiten alle im selben Raum, womit die Möglichkeit der osmotischen Kommunikation gegeben ist. Erlaubt sind jedoch auch kleine Teams, die sich in benachbarten Büros befinden und sich oft für persönliche Gespräche (bezüglich der Arbeit) gegenseitig aufsuchen. Eine Ausdehnung auf verschiedene Gebäude oder Kontinente ist nicht vorgesehen (Vgl. [Hans10], S. 54).

Crystal Orange gibt für die Bewältigung des Koordinationsaufwandes stärkere Mechanismen vor. Gefordert sind, im Unterschied zur Crystal Clear-Variante, eine ausführliche Dokumentation der Anforderungen, der Architektur und der Schnittstellen. Die Anzahl der Projektrollen ist auf 14 erhöht. Die Teams sind spezialisiert, bestehen aus drei bis sechs Mitgliedern und arbeiten parallel (Vgl. [Buns08], S. 133).

Ein Crystal Orange-Projekt kann bis auf 40 Personen skaliert und auf 1-2 Jahre geplant werden. Die Einteilung der Teams, abweichend von den anderen untersuchten Vorgehensmodellen, erfolgt nach unterschiedlichen Funktionen – in sogenannte *Function-Teams*, die sich mit einem konkreten System-Modul beschäftigen. Ein *Infrastructure-Team* kümmert sich um die technischen Voraussetzungen für die Arbeit der Function-Teams, deren Analysten, Geschäfts- und Anwendungsexperten wiederum für detaillierte Anforderungsdefinition sorgen. Ein *System-Planning-Team* übernimmt die Planung des Projekts. Dieses Team umfasst auch einen Architekten, der mit den führenden Entwicklern aus den Function- und Infrastructure-Teams das *Architecture-Team* bildet. Außerdem überwacht ein *Project-Monitoring-Team*, bestehend aus dem Projektmanager und den Leitern anderer Teams, den Projektfortschritt. Die Treffen der Fachexperten zur Definition von Richtlinien und zum notwendigen Informationenaustausch werden von *Technology-Teams* durchgeführt. Solche Teams können aus Datenbankexperten, UI-Designern, Security-Experten und Testern bestehen (Vgl. [Stey10], S. 66-67).

Crystal gibt, zumindest für die Versionen Clear und Orange, keine Maßnahmen zum verteilten Arbeiten vor. Jedoch wird für jedes Projekt eine Methode aus der Crystal-Familie ausgewählt, die von dem selbstgesteuerten Team an bestimmte Bedingungen weiter angepasst werden kann. Die Skalierungsmöglichkeiten für Crystal

Orange sind aber durchaus definiert. Aus diesen Gründen erfolgt die Bewertung der Skalierbarkeit und des verteilten Arbeitens bei Crystal mit ***ausreichend***.

4.3.4.4 Dokumentationsaufwand

Die Dokumentation ist in einem gewissen und minimalen Maße unverzichtbar. Der Dokumentationsaufwand steigt mit der steigenden Anzahl der Teammitglieder. Wenn bei einem Team mit sechs Personen alle Informationen auf direktem Weg zwischen diesen Personen fließen können, ist das bei fünfzig oder hundert nicht mehr möglich. Aus diesem Grund ist eine ausführliche Dokumentation der Entscheidungen und der Ergebnisse mit steigender Zahl der Projektbeteiligten umso wichtiger und unverzichtbar.

Crystal Clear zählt zu der leichtgewichtigsten Variante der in dieser Arbeit untersuchten Vorgehensmodelle. Die Dokumentation erfolge laut C. Bunse lediglich auf Whiteboards ([Buns08], S. 133). Die erforderlichen Dokumente seien nach P. Hruschka folgende Produkte: Release-Plan, Review-Plan, einfache Use Cases, Design Skizzen, laufender Code, Objektmodell und das Benutzerhandbuch. Bei Crystal Orange kämmen dazu das User-Interface-Dokument, die Schnittstellenspezifikationen, das Requirements-Dokument und die Statusberichte ([Hrus09], S. 56).

Betrachtet man den Dokumentationsaufwand der Crystal-Familie insgesamt, so lässt sich feststellen, dass für jede Ausprägung bezogen auf eine Projektart der Aufwand stets minimal gehalten wird. Deswegen erfolgt die Bewertung mit ***gering***.

4.3.4.5 Qualitätssicherungsmaßnahmen

Die Erfüllung der sieben Crystal-Eigenschaften (siehe Kapitel 4.3.1), insbesondere der Einsatz der täglichen Integration, der automatisierten Tests und des Konfigurationsmanagements zählen zu den wichtigen Erfolgsfaktoren. Mehrere Reviews, die Teammeetings und die regelmäßige Auslieferungen steigern erheblich die Qualität der Produkte, sowie auch die Zufriedenheit des Kunden. Über den Entwicklungsprozess wir nach jeder Iteration in einem Workshop reflektiert und es werden nach Bedarf zeitnahe Konsequenzen gezogen.

Bei Crystal Clear kann das Team seinen Code gegenseitig testen, ab der Stufe Orange ist die Rolle des Testers vorgeschrieben (Vgl. [Hrus09], S. 56). Die Testfälle werden von den Use Cases abgeleitet. Die Rollen des Designers und des Programmierers sind zusammengelegt. So werden die möglichen Probleme bei der Kommunikation zwischen diesen Stationen bereits im Vorfeld ausgeschossen. Des Weiteren steigt mit der Anzahl der Beteiligten und der Kritikalität auch die Anzahl der verschiedenen kontrollierenden

Rollen. Der Prozess wird formeller, wonach z.B. nach der Orange-Variante bei mehreren Teams eine genaue Definition der Schnittstellen gefordert ist.

Zusammenfassend bieten die Crystal-Methoden viele Mechanismen und somit ***gute*** Maßnahmen zur Qualitätssicherung.

4.3.4.6 Flexibilität bei Anforderungsänderungen

Der Kunde hat am Anfang des Projekts die Gelegenheit, dem Team seine Anforderungen und Wünsche mitzuteilen. Diese werden für Iterationen und für Releases eingeplant. Am Ende jeder Iteration und während der regelmäßigen Meetings hat er die Möglichkeit, neue Funktionalitäten in Auftrag zu geben und alte streichen zu lassen. Keiner der Beteiligten darf aus Prioritätsgründen die sich in Bearbeitung befindlichen Anforderungen als weniger wichtig definieren und sie somit in der Abarbeitungsreihenfolge nach unten bewegen. Angefangene Aufgaben sind zu Ende zu führen.

Die relativ lange Iterationsdauer von bis zu vier Monate kann sich auf die Flexibilität bei den Anforderungen nachteilig auswirken. Zusammenfassend ist sie mit ***gut*** bewertet.

4.3.4.7 Kundeninvolvierung

Für die Informationsversorgung über die Systemanforderungen, die Einsatzumgebung und die Geschäftsabläufe sind mehrere Rollen vorgesehen. Für Crystal Clear sind das z.B. der Auftraggeber, der erfahrene Anwender und der Fachexperte.

Um herauszufinden, was der Kunde nun wirklich benötigt, werden Interviews durchgeführt. Die Anfertigung eines Prototyps deckt auf und minimiert die möglichen Risiken. Dadurch wird auch dem Kunden klarer, welche Eigenschaften sein System aufweisen soll, was nicht jedem am Anfang eines Projekts schon bewusst ist.

Die Auslieferungen der Softwareinkremente geschehen bei Crystal Clear alle zwei bis drei Monate, bei Crystal Orange alle drei bis vier Monate (Vgl. [Hrus09], S. 56). Der Kunde wird in alle Aktivitäten regelmäßig einbezogen. Bei Crystal Clear wird empfohlen, zwei Anwender-Reviews pro Iteration anzusetzen. Zudem muss der Weg der Kontaktaufnahme zu den Rollen der Kundenseite kurz sein, aber seine ständige Anwesenheit ist nicht explizit gefordert (Vgl. [Stey10], S. 37).

Die aufgeführten Maßnahmen der Kundeninvolvierung erscheinen bei Crystal als ***ausreichend*** im Sinne dieses Untersuchungspunkts.

4.3.5 Fazit

Dem Crystal-Ansatz zufolge können verschiedene Projekte nach keinem Standard-Vorgehensmodell durchgeführt werden. Es ist eine speziell zugeschnittene Methodik nötig, die bei Bedarf erweiterbar ist. Aus diesem Grund entwickelte A. Cockburn die Crystal-Methodenfamilie, die für viele Projektarten jeweils eine passende Ausprägung anbietet. Die ständige Anwesenheit des Kunden (Customer On-Site) ist nicht gefordert, lediglich dass er in ausreichender Form zur Verfügung steht.

Zu wichtigen Leitsätzen von Crystal zählen die maximale Selbstorganisation des Teams und die minimale Kontrolle. Je nach zunehmendem Farbton wird Crystal formeller und somit schwergewichtiger. Größere Anzahl der Beteiligten und auch die Kritikalitätsstufe des Projekts machen es erforderlich.

Crystal besteht auf die Einhaltung zweier wichtiger Regeln. Zum ersten wird die Iteration auf maximal vier Monate begrenzt. Die Dauer kann kürzer, aber auf keinen Fall länger andauern. Zum zweiten soll jedes Team die für das Projekt gewählte Methode in regelmäßigen Workshops auf Effizienz prüfen und diese entsprechend anpassen. Dazu wird das Team während einer Iteration zur Einschätzung der Fortschritte befragen und demnach gehandelt (Vgl. [Hrus09], S. 57-58).

Die einfache Handhabung bezüglich der Auswahl der eingesetzten Praktiken und maßgeschneiderte Methodenvielfalt zeichnen Crystal besonders aus und machen das Vorgehensmodell besonders effektiv.

4.4 Kanban für die Softwareentwicklung

Kanban, als ein Vorgehensmodell für die Softwareentwicklung, stellte David J. Anderson im Jahr 2007 vor. Bis dahin war das eine Technik der „Lean Production“[16] aus der Automobilindustrie des japanischen Herstellers Toyota, um die Lagerbestände und somit die Kosten gering zu halten. Analog dazu sollen in der Softwareentwicklung die Durchlaufzeiten reduziert und die Durchlaufzahlen erhöht werden.

Der Name setzt sich aus den japanischen Wörtern *kan* (übersetzt Signal) und *ban* (übersetzt Karte) zusammen und bedeutet eine „Signalkarte“. Der Grundgedanke dieser Technik sieht vor, bei Bedarf mit einfachen Mittel Aufmerksamkeit zu erregen. Das ist dann notwendig, wenn der gleichmäßige Fortschritt eines Ablaufs in Gefahr zu geraten droht.

Kanban ist eine Veränderungsmanagement-Methode (engl. Change Management Method). Die Einführung erfolgt nicht revolutionär von heute auf morgen, sondern evolutionär, durch das Bestreben nach ständiger Verbesserung (jap. Kaizen), in kleinen Schritten. Die Veränderungen an bestehenden Strukturen und Abläufen erfolgen durch die Menschen selbst. Die charakterisierenden Elemente von Kanban sind:

- Das Nehmen und nicht das Geben der Arbeit
- Limitierung der Mengen
- Veröffentlichung der Informationen
- Kontinuierliche Verbesserung der Abläufe

4.4.1 Die Werte von Kanban

Kanban basiere nach T. Epping auf folgenden Werten der schlanken Softwareentwicklung ([Eppi11], S. 40):

Eliminate Waste:

Eliminiere den Ballast. Als Ballast wird die Eigenschaft eines Arbeitsschritts bezeichnet, die für das Produkt und somit für den Kunden keinen Mehrwert darstellt, bzw. die diesen in irgendeiner Weise verlangsamend oder behindernd belastet.[17]

Amplify Learning:

Ermögliche den Beteiligten aus den Fehlern zu lernen, wie auch sich in neue Fachbereiche einzuarbeiten. Das ist das Fundament der kreativen Arbeit.

[16] Übersetzt: schlanke Produktion, im Sinne von frei von unnötigem Ballast.
[17] Zu Formen von Waste siehe [Eppi11] S. 48, Kapitel 3.5.2.1.

Decide as Late as Possible:

Entscheide so spät wie möglich. Dadurch sind zum Zeitpunkt der Entscheidung die meisten und aktuellsten Informationen verfügbar.

Deliver as Fast as Possible:

Durch kurze Durchlaufzeiten und somit schnelleres Arbeiten ist es nicht nötig viele Aufgaben parallel zu bearbeiten. Mehrere Aufgaben können nacheinander schneller bewältigt werden, als parallel.

Empower the Team:

Verteile das Wissen über das Projekt auf das gesamte Team. Der Ausfall eines Einzelnen ist dann nicht mehr so problematisch.

Build Integrity in:

Wege die Notwendigkeit der Faszination des Kunden für erhöhte Benutzerakzeptanz gegen die Bereitschaft Ballast zu produzieren ab.

See the Whole:

Sorge für das ausgewogene Ganze und weniger für die ausgewogenen, aber nicht aufeinander abgestimmten Teile des Ganzen. Das gilt für die Komponente einer Software, wie auch für die Wissensverteilung auf Personen eines Teams.

4.4.2 Charakterisierende Elemente von Kanban

Folgende vier Elemente charakterisieren, nach T. Epping, die oben beschriebenen Werte von Kanban ([Eppi11], S. 53-65):

Pull – Arbeit wird genommen, nicht gegeben:

Nachdem eine Aufgabe an einer Station abgearbeitet ist, wird die nächste aus den vorgelagerten der Station davor selbstständig gezogen (engl. pull) und nicht von der übergeben (engl. push). Auf diese Weise organisiert sich das Team selbst und seine Überlastung ist verhindert.

Limitierung der Mengen:

Die Anzahl an parallel abzuarbeitenden Aufgaben (sog. WiP, Work in Progress) jeder Station ist begrenzt. Eine Aufgabe muss erst vollständig abgeschlossen sein, damit eine neue angefangen werden kann. Das steigert die Durchlaufzahlen und vermeidet gleichzeitig die Überlastung. Auf der anderen Seite kann das die Engpässe einer Station aufzeigen.

Transparente Information – Informationen werden veröffentlicht:

Jedes Teammitglied muss sich innerhalb kürzester Zeit einen Überblick über den aktuellen Fortschritt der Entwicklung verschaffen können. Dazu gehören

Informationen über alle Phasen samt Limits, alle darin befindlichen Aufgaben und die dafür verantwortlichen Personen. Diese fortschrittsbeschreibenden Projektkennzahlen sind möglichst auf eine einfache Art und Weise, wie z.B. auf den Whiteboards mit Notizzetteln zu visualisieren.

Kontinuierliche Verbesserungen der Arbeitsabläufe:

Die Arbeitsabläufe sind ständig auf Schwachstellen zu untersuchen. Ist eine gefunden, muss diese schnellstmöglich beseitigt werden.

Kanban lässt sich um weitere Elemente und Techniken nach Belieben erweitern. Im Vordergrund stehen kontinuierliche Verbesserung des Arbeitsablaufs und für das Team die ständige Steigung der Lernkurve.

4.4.3 Techniken von Kanban

T. Epping nennt in seinem Buch drei zentrale und acht weitere Techniken[18], die er anhand von Beispielen aus einem realen Softwareentwicklungsprojekt erläutert ([Eppi11], S. 85-129):

Kanban-Board:

Das ist eine der zentralen Techniken, um dem Team eine Übersicht über das ganze Projektgeschehen zu geben. Jedes Board ist projektindividuell aufzubauen. Mit der Zeit lässt sich schnell erkennen, an welchen Stationen und mit welchen Abläufen es Probleme gibt und an welchen Veränderungen vorzunehmen sind. Ein Beispiel für den Aufbau eines Boards ist in Abbildung 10 dargestellt.

Standups:

Das sind tägliche, ca. 15 Minuten dauernde Treffen des Teams. Die finden immer am selben Ort und zur selben Zeit statt. Am besten ist dafür die Nähe des Kanban-Boards geeignet, wo jeder sich gleich eine Übersicht verschaffen kann. Jeder berichtet den anderen Teammitgliedern über seinen aktuellen Stand der Arbeit. Es gilt folgende Fragen zu beantworten:

- Was habe ich seit dem letzten Treffen gemacht?
- Was werde ich bis zum nächsten Mal tun?
- Was behindert mich derzeit an meiner Arbeit?

[18] Die drei zentralen gefolgt von acht weiteren Techniken werden nacheinander erläutert.

Retrospektiven:

Das sind wöchentliche oder monatliche Treffen des gesamten Teams, um über die letzten Erfolge oder Misserfolge zu reflektieren. Es gilt die Schwächen im Ablauf, wie auch an einzelnen Stationen zu identifizieren und Maßnahmen zu beschließen.

User Stories:

Sie beschreiben aus der Sicht des Kunden in einer bestimmten Form eine oder mehrere Anforderungen. Daraus können die Tests abgeleitet werden.

Planungspoker:

Das Team diskutiert über die User Stories und schätzt den Aufwand um sie zu realisieren. Als Einheit für die Schätzung kann die zu erwartende Komplexität, die benötigte Zeit in Tagen für die Bearbeitung durch eine Person oder auch die ersten acht Fibonacci-Zahlen[19] gewählt werden.

Code Reviews:

Nach der Realisierung einer Anforderung begutachten die Entwickler den Ablauf mit dem Ergebnis und diskutieren darüber. Als Resultat des Reviews können neue User Stories entstehen, die z.B. ein Refactoring des Codes beschreiben.

Continuous Integration:

Die Entwickler integrieren mindestens einmal am Tag ihren geschriebenen Code in das Gesamtsystem. Auf diese Weise wird der noch nicht vollständige Code durch automatisierte Tests auf Fehler und auf Funktionsfähigkeit mit dem Gesamtsystem geprüft und anschließen einer anderen Bearbeitungsstation zur Verfügung gestellt (z.B. der Qualitätssicherung).

Abnahmekriterien:

Diese beschreiben den Zustand einer Anforderung oder des Gesamtsystems aus der Sicht des Kunden, nachdem alle qualitätssichernden Maßnahmen abgeschlossen sind. Die Abnahmekriterien können in den User Stories beschrieben sein. Sie signalisieren die Bereitschaft zur Übergabe an den Kunden.

[19] Fibonacci-Zahl ist eine durch die Fibonacci-Folge errechnet Zahl, die sich aus der Addition der letzten beiden Zahlen errechnet.

Testautomatisierung:

Der Ablauf der gesamten Testfälle ist automatisiert, damit der Zwischenstand der Entwicklung regelmäßig und schnell geprüft werden kann.

Entwicklungsgeschwindigkeit:

Diese stellt den geschätzten Aufwand für die bereits realisierten Anforderungen innerhalb einer Zeitspanne dar. Sie ist ein nützliches Visualisierungsinstrument für das Team und für das Projektmanagement.

Durchlaufzeit:

Diese Angabe gibt die Zeitspanne wieder, für die eine Anforderung zum Durchlaufen aller Entwicklungsphasen benötigt. Daraus lässt sich errechnen, wie schnell ein Produkt auf den Markt kommt.

4.4.4 Rollen bei Kanban

Kanban schreibt kein festes Rollenmodell vor. Das ist eine Methode des Veränderungsmanagements und aus diesem Grund bleiben die im Unternehmen bereits etablierten Rollen anfangs bestehen. Durch die ständige kritische Begutachtung der Abläufe, dem Wert der kontinuierlichen Verbesserung, wird schnell klar, welche Rollen abgeschafft und welche ergänzt werden müssen.

Aus folgenden Projektphasen lassen sich jedoch mögliche Belegungen der Rollen mit folgenden Verantwortlichkeiten ableiten (Vgl. [Eppi11], S. 70-84):

Phase:	Aufgaben:
Requirements Engineering	Erhebung, Dokumentation, Prüfung und Verwaltung der Anforderungen.
Entwicklung	Umwandlung einer Anforderung in Software.
Qualitätssicherung	Verifizierung der Umsetzung einer Anforderung.

Zusätzlich sollte eine Belegung folgender Rollen erfolgen:

Projektmanagement	Er ist zuständig für erfolgreiche Durchführung des Projekts nach den Werten und charakteristischen Elementen von Kanban.
Auftraggeber	Er vertritt die Interessen der späteren Anwender. Er kann sich entscheiden, ob er sich aktiv am Projekt beteiligt, es passiv unterstützt oder es ignoriert.

4.4.5 Der Ablauf

Abbildung 10 zeigt den beispielhaften Aufbau eines Kanban-Boards.

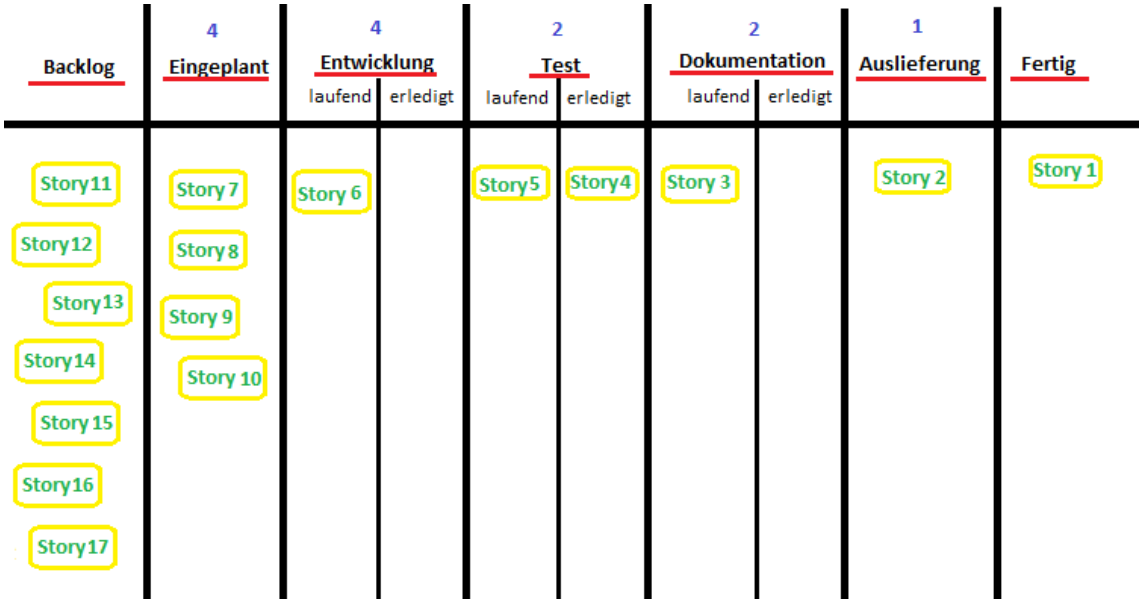

Abbildung 10: Kanban-Board

Die Spalten des Boards stellen die Projektphasen dar. Im Backlog befinden sich anfangs die gesammelten Anforderungen in Form von User Stories. Das Backlog ist nicht priorisiert. Die Entscheidung, welche Story als nächste bearbeitet wird, fällt nach dem Wert „Decide as Late as Possible“ so spät wie möglich. Diese Entscheidung kann in regelmäßigen Priorisierungsmeetings fallen.

Die Stories wandern im Verlauf der Entwicklung von links nach rechts, wobei jede Station gleichzeitig nur eine bestimmte Anzahl von denen aufnehmen kann (blaue Zahl). Zum Beispiel an der Station *Test* in Abbildung 10 beträgt das Limit für parallele Abarbeitung der Anforderungen zwei und da befinden sich aktuell auch zwei User Stories. Das bedeutet, dass diese Station keine neuen Aufgaben anfangen kann, solange die nächste Station *Dokumentation* sich keine neue Aufgabe „gezogen“ hat (in diesem Fall wäre das Story 4). Sobald eine Anforderung die Station *Fertig* erreicht, gilt sie als abgeschlossen.

Gerade am Anfang der Kanban-Einführung kann es schnell passieren, dass die Limits nicht gut gewählt sind und eine Station voll belegt ist. Dadurch können auch die anderen Stationen beim Erreichen ihrer Limits nicht weiterarbeiten und es entsteht ein Stau. In diesem Fall kommt eine Technik namens „Swarming“ zum Einsatz, dass übersetzt „das Ausschwärmen“ bedeutet. Dabei werden alle Teammitglieder zur Behebung des Problems herangezogen.

4.4.6 Bewertung von Kanban

4.4.6.1 Einarbeitungszeit

Eingeführt wird Kanban nach dem Prinzip des evolutionären Veränderns und Verbesserns und nicht des Neuerschaffens. Auf diese Weise ist die Einführung von geringem Aufwand, wird schnell akzeptiert und kann auch mittendrin in einem Projekt erfolgen (Vgl. [Eppi11], S. 131). Die bestehenden Strukturen eines Unternehmens werden nicht über Nacht verworfen, sondern es findet ein weicher Übergang statt. Auch die etablierten Rollen bleiben am Anfang bestehen. Erst nach einer kritischen Begutachtung in einer Retrospektive kann sie bei Bedarf verändert werden. Kanban gibt dem Team die Möglichkeit, die Schwächen ihrer bis dahin bestehenden Arbeitsweise selbstständig aufzudecken und zu minimieren. Das ist auch gleichzeitig ein Punkt, der sich auf Dauer als unwirksam herausstellen kann. Wenn die Mitarbeiter keine Schwächen an dem Prozess und an ihrer Arbeitsweise finden können, dann bleibt alles beim Alten. Die Vorgaben für die durchzuführenden Schritte bei der Einführung, genauso wie unterstützende Hilfestellung, sieht Kanban nicht vor. Somit liegen der Erfolg und die Akzeptanz des Vorgehensmodells bei der Qualität und Erfahrung der verantwortlichen Projektmitarbeiter.

Die Einarbeitungszeit von Kanban kann sehr kurz sein, aber ohne geschultes Personal bleibt das ganze Potential von Kanban lange nicht ausgeschöpft. Trotzdem ist anzunehmen, dass die Einführung eines neuen Vorgehensmodells nicht ohne entsprechendes Knowhow angegangen wird, deswegen ist die Einarbeitungszeit bei Kanban als ***kurz*** bewertet.

4.4.6.2 Transparenz

Die periodischen Retrospektiven haben als Wirkung eine ständige Verbesserung der Abläufe. Als Nebeneffekt gilt die Verbreitung der Informationen über die Schwierigkeiten und die Änderungsmaßnahmen als ein gutes Mittel zur Transparenz des Verbesserungsprozesses.

Die Transparenz des Projektverlaufs ist durch die Darstellung des aktuellen Arbeitsfortschritts auf dem Kanban-Board jederzeit für jedermann gegeben. Das Team weiß auch durch die Techniken, wie z.B. die täglichen Standup-Meetings oder durch Projektkennzahlen, wie viel es bereits geschafft hat, wie viel in Bearbeitung ist und wie viel es noch zu schaffen gilt.

Die Transparenz bei Kanban ist aufgrund der vorgesehenen und aufgeführten Techniken sehr gut gegeben. Die mangelnde Klarheit bei der schrittweisen Umstrukturierung der

Rollen und der Abläufe in einem Unternehmen lassen sie in diesem Untersuchungspunkt nur mit ***gut*** bewerten.

4.4.6.3 Skalierbarkeit und verteiltes Arbeiten

Kanban ist im Jahr 2007 vorgestellt worden und ist somit ein relativ neues Vorgehensmodell. Aus diesem Grund gibt es nur wenige Berichte zur Skalierung und zum Einsatz in verteilter Umgebung. Da Kanban eine Veränderungsmanagement-Methode ist, macht sie keine Vorgaben für die Teamgröße oder für die Skalierungsmöglichkeit. Aus der Ferne betrachtet lassen sich zwar manche Techniken, wie z.B. das Kanban-Board, für verteiltes Arbeiten in elektronischer Form umsetzen, jedoch stellt das keine Kompensation für die anderen Techniken dar. Für ein verteiltes Team besteht die Gefahr die gesamte Wertschöpfungskette aus dem Blick zu verlieren, indem es sich nur auf seine Aufgabe konzentrieren würde und somit die Werte, wie die kontinuierliche Verbesserung, vernachlässigen würde.

Genaue Zahlen zu der Teamgröße kann Kanban ebenfalls nicht angeben. Jedes Unternehmen muss unter Einsatz der Kanban-Techniken selbst herausfinden, wie groß das Team für ein gegebenes Projekt sein muss, angefangen mit der üblichen, bis hin zur optimalen.

Dem folgend wird dieser Untersuchungspunkt als ***mangelhaft*** bewertet.

4.4.6.4 Dokumentationsaufwand

Je nach Organisation der Projektphasen kann die Dokumentation eine der Stationen aus der Wertschöpfungskette sein. Das ist oft dann der Fall, wenn der Kunde eine externe Dokumentation fordert.

Kanban vertritt durch die Technik *Eliminate Waste* die Werte des agilen Manifests[20]. Demnach ist jeder Arbeitsschritt überflüssig, der keinen direkten Vorteil für das Produkt bringt. Folglich, wenn keine Dokumentation explizit verlangt ist, wird auch keine erstellt.

Demnach ist der Aufwand an Dokumentation bei Kanban als ***sehr gering*** eingestuft.

4.4.6.5 Qualitätssicherungsmaßnahmen

Testen kann ebenfalls eine Station der Wertschöpfungskette darstellen. Die Testfälle können aus den User Stories gewonnen werden und als Abnahmekriterien dienen. Die gilt

[20] Vgl. der Wert: „Funktionierende Software mehr als umfassende Dokumentation“ [Mani01].

es anschließend durch den Einsatz von Continuous Integration zu automatisieren (Vgl. [Eppi11], S. 103).

Code-Reviews ist eine wirkungsvolle Technik zur Fehlerfindung. Die Reviews werden nach Abschluss der Entwicklung einer Anforderung durchgeführt, woraus sich neue Stories ergeben können.

Die Qualitätssicherung bei Kanban beschränkt sich nicht nur auf die Software. Eine kontinuierliche Überprüfung bestehender Abläufe und Techniken, wie der Retrospektiven, tragen indirekt auch zur Qualität der Software bei. Die Qualitätssicherungsmaßnahmen sind demnach ***sehr gut***.

4.4.6.6 Flexibilität bei Anforderungsänderungen

Anforderungen sind in einem Backlog gesammelt. Anschließend werden sie nach Priorität aus der nächsten Phase „Eingeplant“ (siehe Kapitel 4.4.5, Abbildung 10) übernommen. Falls eine neue Anforderung entsteht und am höchsten priorisiert ist, wird diese bei Freiwerden eines Platzes in der folgenden Phase als nächstes bearbeitet. Auf diese Weise können immer wieder neue Anforderungen aufgenommen und die wichtigsten Aufgaben zuerst erledigt werden. Die Flexibilität ist somit bei Kanban ***sehr gut*** gegeben.

4.4.6.7 Kundeninvolvierung

Kanban sieht keine festen Rollen für die Entwicklung vor, daher auch keinen Vertreter des Kunden. T. Epping empfiehlt jedoch eine Belegung der Rolle des Auftraggebers. Dieser könne den Einsatz von Kanban für sein Projekt komplett ignorieren oder passiv oder aktiv unterstützen. Wenn der Auftraggeber diesen ignoriere, könne er über eine Informationstransformation auf dem Laufenden gehalten werden. Dabei bekomme er von dem Auftragnehmer in die Kundensprache „übersetzte“ Informationen über das Projekt geliefert, den er auf dem umgekehrten Wege auf dieselbe Art mit Informationen versorgen könne ([Eppi11], S. 69, 82-84).

Bei einer passiven Unterstützung beteiligt sich der Kunde nicht aktiv an dem Kanban-Projekt. Er weiß aber mit der Informationstransformation über alle Aspekte Bescheid. Entscheidet er sich für eine aktive Projektbeteiligung, dann wird der Kunde zu einem Teil des Teams und unterstützt dieses für den Erfolg des Projekts in jeglicher Hinsicht.

Da die feste Einbeziehung des Kunden nach Kanban nicht gefordert ist, wird der Untersuchungspunkt der Kundeninvolvierung als ***schwach*** bewertet.

4.4.7 Fazit

Kanban ist eine Methode der evolutionären Veränderung. Der Fokus ist nicht allein auf die Entwicklung der Software gesetzt, vielmehr stehen der gesamte Prozess und die damit verbundenen Abläufe und Techniken auf dem Prüfstand. Dieses Vorgehen wird häufig in Projekten für Wartung und Betrieb eingesetzt, wo es hauptsächlich darum geht, kurze Durchlaufzeiten zu bekommen und die internen Strukturen nicht ab sofort vollständig umzukrempeln.

Die Probleme macht das Kanban-Board schnell in Form von Blockierungen und daraus entstandenem Stau, der sich bis zum Anfang der Prozesskette fortpflanzen kann, sichtbar. Dadurch müssen diese frühzeitig gelöst werden. Jede angefangene Arbeit wird somit auch beendet und das nicht alles auf einmal erst am Projektende.

Im Kapitel 4.4.5 in Abbildung 10 stellen die Spalten „erledigt“ an mittleren Stationen den Puffer bzw. einen Lager für die Ergebnisse des Arbeitsflusses dar. Es gilt diesen Puffer möglichst gering befühlt zu halten, denn das würde bedeuten, dass die benachbarten Stationen gut aufeinander abgestimmt sind und die Durchlaufzeiten optimal sind. Ein voller Puffer ist ein Zeichen für einen Engpass.

Ein Nachteil bei Kanban lässt sich dennoch erkennen, der sich aus der Tatsache ergibt, dass es kein wirkliches Framework ist. Es gibt nämlich keine wirkliche inhaltliche Unterstützung bei der schrittweisen Umstrukturierung des Unternehmens. Die Personen, die nach diesem Modell für die Verbesserung sorgen sollen, stehen so gut wie alleine im Vordergrund.

Im Oktober 2011 fand das erste Treffen der europäischen Kanban-Community statt. Rund 200 Besucher haben sich Vorträge und Erfahrungsberichte angehört. Das Treffen wäre, einem Artikel von K. Leopold zufolge, ein gelungenes Event, das nächstes Jahr mit „Lean Kanban Central Europe 2012“ in Wien fortgesetzt werde [Leop11].

4.5 Feature Driven Development (FDD)

FDD hat der Australier Jeff De Luca für ein größeres Projekt mit festen Rahmenbedingungen im Jahr 1997 entwickelt. Der Zeitrahmen war auf 15 Monate und die Entwicklerzahl auf 50 Personen beschränkt. Das Hauptaugenmerk dieser Methodik ist auf die geforderten Systemeigenschaften (engl. Features) gerichtet, die iterativ-inkrementell realisiert werden. Die agilen Werte werden weitestgehend eingehalten, weswegen FDD zu den leichtgewichtigen Vorgehensmodellen zählt (Vgl. [Wulf11], S. 164, 167-169). FDD lässt sich besonders einfach in klassisch organisierte Unternehmen einführen, weil die Strukturen mit dem Prozess- und Rollenmodell von FDD gut harmonieren. Der Entwicklungsprozess besteht aus fünf Phasen. Die ersten drei dienen der Konzeption und der Planung, die letzten zwei der eigentlichen Umsetzung und entsprechen einer Iteration.

4.5.1 Rollen bei FDD

FDD definiert ein festes Rollenmodell. Dieses lässt sich in *Schlüsselrollen*, *unterstützende Rollen* und *zusätzliche Rollen* einteilen. Die Schlüsselrollen werden im Folgenden vorgestellt:

Projektmanager (Project Manager):

Das ist der oberste Projektverantwortliche. Er entscheidet über die Zeitplanung, die personelle, finanzielle und sachliche Ressourcenverteilung. Vor dem Start der ersten Entwicklungsphase stellt er das Modellierungsteam aus Entwicklern und Experten der Fachbereiche zusammen.

Chefarchitekt (Chief Architect):

Der Chefarchitekt muss hohes technisches Können und viel Erfahrung aufweisen. Er ist zuständig für den gesamten Entwurf und somit der Entscheidungsträger bezüglich des Systemdesigns.

Entwicklungsmanager (Development Manager):

Die Person in dieser Rolle verwaltet die technischen Ressourcen und koordiniert die Entwicklungsarbeiten des Teams.

Chefprogrammierer (Chief Programmer):

Diese Rolle hat die Erfahrung und den Überblick über die Schritte des gesamten Entwicklungsprozesses. Sie führt ein Team von ca. sechs Entwicklern an und fungiert ebenfalls als ein Entwickler. Chefprogrammierer müssen dafür sorgen, dass die Realisierung eines Features in hoher Qualität zum vorgesehenen Zeitpunkt zur Verfügung steht.

Klassenbesitzer (Class Owner):

Das sind die eigentlichen Entwickler der Software, die alle Aufgaben für die Umsetzung der Features übernehmen. Sie designen, implementieren, testen und dokumentieren ihnen zugeteilten Klassen und haben alleine das Recht diese zu verändern.

Domänenexperte (Domain Expert):

Er wird auch Fachexperte, Kunde oder Endbenutzer genannt und repräsentiert die Seite des Arbeitgebers. Er versorgt die Entwickler mit Einzelheiten über die Systemumgebung und mit präzisen Anforderungen an das System.

4.5.2 FDD-Teilprozesse / Der Ablauf

Der Entwicklungsprozess besteht aus fünf Teilprozessen (siehe Abbildung 11), wobei die letzten zwei nacheinander wiederholt ablaufen können und somit die Iterationen nachbilden. Bei einer Projektlänge von sechs Monaten sind für die ersten drei Planungsphasen zwei bis drei, für die Iterationen der zwei Entwicklungsphasen jeweils zwei Wochen vorgesehen. Für jede Phase sind Eintrittskriterien, Aufgaben, Verifikation und Austrittskriterien formuliert. Der Ablauf ist mit folgenden Teilprozessen definiert (Vgl. [ItAg08]; [Wolf11], S. 165-166):

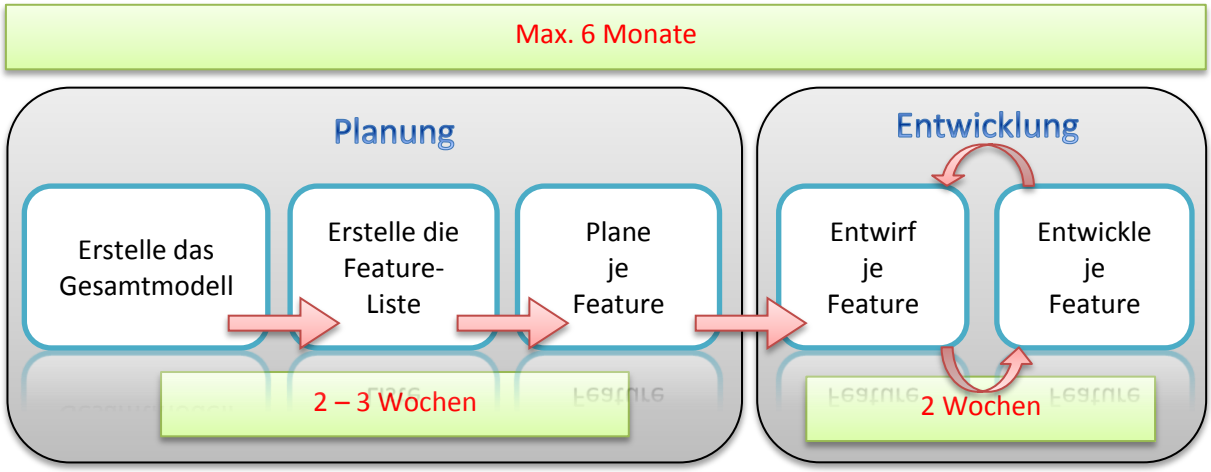

Abbildung 11: FDD-Prozessablauf

Erstelle das Gesamtmodell (Develop an Overall Model):

In dieser ersten Planungsphase beschreiben die vom Projektmanager für das Projekt ausgewählten Domänenexperten zusammen mit den Entwicklern unter der Leitung eines Chefarchitekten die Funktionalitäten des Systems und erstellen den groben Systementwurf – das Domänenmodell. Die Systembereiche werden schrittweise verfeinert. Kleinere Teams aus Fachexperten und Entwicklern fertigen daraufhin Modelle der Systemabschnitte an, stellen sie den anderen zur

Begutachtung vor, eventuell verbessern sie. Auf diese Weise entsteht das fachliche Kernmodell des Gesamtsystems in Form von UML-Diagrammen. Die Ergebnisse dieser Phase können als Ersatz des traditionellen Pflichtenhefts angesehen werden.

Erstelle die Feature-Liste (Build a Features List):

Ein Team aus Chefprogrammierern aus der ersten Phase zerlegt die festgelegten Systembereiche in Geschäftstätigkeiten und diese dann in Features. Beispielweise könnte der Bereich *Verkauf* heißen, die Tätigkeit wäre *Rechnungserstellung* und das Feature *Berechnung der Endsumme*. Die Features werden nach dem Schema „<Aktion> <Ergebnis> <Objekt>“ in die Feature-Liste nach Kategorien aufgenommen.

Plane je Feature (Plan by Feature):

Projektmanager, Entwicklungsmanager und die Chefprogrammierer planen und priorisieren in dieser Phase die Reihenfolge der Features aus der Liste der Phase zwei. Der daraus entstandene Entwicklungsplan muss viele Randbedingungen, wie die Komplexität und Abhängigkeiten der Features oder die Auslastung der Entwicklungsteams, berücksichtigen. Außerdem werden der Fertigstellungstermin und die verantwortlichen Entwickler (zunächst nur der Chefprogrammierer) bestimmt.

Entwirf je Feature (Design by Feature):

Der Chefprogrammierer hat eine Reihe von Features, die er realisieren muss. Er wählt für die nächste Iteration diejenigen aus, die ihm innerhalb von zwei Wochen machbar erscheinen und möglichst solche, die dieselben Klassen und dasselbe Fertigstellungsdatum haben. Danach identifiziert er die Klassenbesitzer, die diese Features implementieren und bildet somit die Feature-Teams. Die Teams erstellen ein oder mehrere Sequenzdiagramme, anhand deren der Chefprogrammierer die Klassenmodelle und somit auch das gesamte Domänen-Modell verfeinert. Die Entwickler schreiben Klassen- und Methodenrümpfe. Als letztes unterziehen die Teams mit dem Chefprogrammierer den Designentwurf einer Inspektion.

Entwickle je Feature (Build by Feature):

Nach dem Entwurf der Features folgt die Implementierung der Klassen durch die Klassenbesitzer. Zur Sicherstellung der Qualität kommen Codeinspektionen und Komponententests (Unit-Tests) zum Einsatz, deren Abfolge der Chefprogrammierer beschließt. Am Ende dieser Phase entstehen ein oder mehrere ausgereifte und dokumentierte Features, die dem Kunden vorgeführt werden.

4.5.3 Best Practices von FDD

Domain Object Modeling:

Am Anfang des Projekts muss ein Überblick über den Problembereich geschafft werden. Die Systemabschnitte mit ihren Beziehungen werden in Form von Klassen- und Sequenzdiagrammen modelliert. Diese Diagramme werden im Laufe des gesamten Projekts zur Übersicht benötigt und schrittweise verfeinert.

Developing by Feature:

Der grobe Systementwurf wird ausführlich in erforderliche Klassen, Beziehungen zwischen denen und Methoden solange zerlegt, bis eine gewünschte Funktionalität innerhalt von zwei Wochen realisierbar ist.

Class Ownership:

Die Abschnitte bzw. die Klassen des Codes gehören einzelnen verantwortlichen Entwicklern, die diese bestens kennen und verändern dürfen. Einem Entwickler können mehrere Klassen gehören, aber eine Klasse keinen mehreren Entwicklern.

Inspections:

Das Design und der Code unterliegen regelmäßiger Überprüfung durch andere Teammitglieder. Während einer Inspektion erklärt der Entwickler seinen Code. Dadurch haben die anderen eine Gelegenheit falls vorhandene Fehler aufzuzeigen und auch gleichzeitig voneinander zu lernen.

Feature Team:

Diese Teams bestehen aus Klassenbesitzern und realisieren unter Leitung eines Chefprogrammierers ein konkretes Feature. Je nach Auslastung können Klassenbesitzer auch mehreren Feature Teams zugeordnet sein.

Regular Build Schedule:

Die regelmäßigen Builds des Gesamtsystems beinhalten alle bis dahin implementierten Features. Sie prüfen das Zusammenspiel zwischen den Features und erzeugen ein lauffähiges System, welches den aktuellen Entwicklungsstand für z.B. eine Vorführung wiederspiegelt. Auch die Tester, falls welche vorhanden sind, erhalten nach einem Build ein Softwareinkrement zur Untersuchung.

Configuration Management:

Ein Versionskontrollsystem ist einzusetzen, das nicht nur den Code, sondern auch andere wichtige Artefakte verwaltet.

Visibility of Progress and Results:

Der Projektmanager, aber auch andere Schlüsselrollen in einem Entwicklungsprozess, benötigen zur Planung stets aktuelle Informationen. Dafür dient beispielsweise das Verhältnis der abgeschlossenen Features zu

Gesamtanzahl. Bei laufenden Features wird das Prinzip der *sechs Meilensteine* angewandt, das folgend beim Untersuchungskriterium der *Transparenz* beschrieben wird.

4.5.4 Bewertung von Feature Driven Development

4.5.4.1 Einarbeitungszeit

Die Einführung von FDD erfolgt bei klassisch organisierten Unternehmen besonders unproblematisch, da dortige Strukturen dem Vorgehen ähneln (Vgl. [ItAg07]). Die Beschreibung der Methodik umfasst überschaubare 10 Seiten. Die hierarchischen Rollenstrukturen machen die Selbstorganisation nicht mehr möglich, was sich positiv auf die untergeordneten Rollen auswirkt. Die Aufgaben werden mit der übergeordneten Rolle abgesprochen bzw. von ihr verteilt. Dadurch kann die Entwicklung nicht in die falsche Richtung verlaufen und man hat bei Fragen einen Ansprechpartner.

Die aufwendigen Phasen der Planung und des Entwurfs wirken sich auf die wechselnden Rahmenbedingungen eher störend aus. Dieser Umstand erschwert anfänglich die möglichen Versuche der Anpassung des Prozesses durch die Projektmitwirkende.

Die Einarbeitungszeit ist somit bei klassischer Organisation ***sehr kurz***. Bei einem agil handelnden Unternehmen könnte es deutlich länger dauern. Das Rollenmodell muss sich erst etablieren, die planenden und entscheidungstragenden Rollen müssen sich eine gewisse Erfahrung aneignen und das Team muss sich an den Ablauf gewöhnen.

4.5.4.2 Transparenz

Der geringe Umfang eines Features, das innerhalb von maximal zwei Wochen fertiggestellt wird, macht die Implementierung im Aufbau klar und überschaubar. Den Projektfortschritt machen *sechs festgelegte Meilensteine* für jedes Feature transparent, die auf die Iterationsphasen vier und fünf verteilt sind. *Drei* sind für den Entwurf und *drei* für die Implementierung vorgesehen. Anhand dieser lässt sich der aktuelle Entwicklungsstatus eines Features während einer Iteration leicht erfassen. Befindet sich beispielsweise ein Feature in der Implementationsphase, so ist es zu 44% fertig. Dieser Anteil setzt sich aus den drei *Entwurfsmeilensteinen* zusammen: 1% Domänenanalyse, 40% Entwurf, 3% Entwurfsinspektion. Die Restlichen drei *Implementierungsmeilensteine* verteilen sich auf: 45% Implementierung, 10% Implementierungsinspektion, 1% Auslieferung (Vgl. [Buns08], S. 145).

Die Anzahl der geforderten und die der bereits realisierten Features machen den Projektverlauf für die Entwickler und für den Kunden übersichtlich. Der Kunde findet

hauptsächlich in der Planungsphase seinen Einsatz. In die Entwicklungsphasen wird er nicht einbezogen, was seine Einfluss- und Einsichtsmöglichkeiten einschränkt.

Des Weiteren macht die Rollenhierarchie das Weitertragen der gesamten Informationen bis an die unterste Ebene überflüssig. In jedem Feature-Team existiert eine übergeordnete Person - der Chefprogrammierer. Er leitet das Team, das auch relativ unerfahren sein kann, und behält den Überblick, wie z.B. über die zu beachtenden Anhängigkeiten von Features. Aus diesem Grund repräsentiert diese Rolle die erste Anlaufstelle für einen Klassenbesitzer, falls er Schwierigkeiten bei dem Ablauf oder bei technischen Fragen hat. Die Notwendigkeit für die Klassenbesitzer über den Tellerrand der Entwicklung seines Features zu blicken, um das Gesamtsystem im Ganzen zu überschauen, fällt somit weg.

Die aufgeführten Punkte machen Feature Driven Development ***ausreichend*** transparent.

4.5.4.3 Skalierbarkeit und verteiltes Arbeiten

FDD eignet sich besonders gut für große und verteilte Projekte, ist aber aufgrund der aufwändigeren Planung und Modellierung schwergewichtiger als z.B. Scrum oder XP (Vgl. [Buns08], S. 148).

Für ein einzelnes Feature mit einer Entwicklungsdauer von max. zwei Wochen wird kein großes Team benötigt, es sei denn, es sind viele unterschiedliche Klassen und somit im ungünstigsten Fall viele Klassenbesitzer involviert. Auf der Ebene der *Leitung* (siehe Abbildung 12) sind, abhängig von der Größe des Projekts, zwei (mindestens Projektmanager und Chefarchitekt) bis sechs leitende Personen erforderlich. Die *Entwicklungsebene* sollte die Anzahl von fünf Mitgliedern, das einem durchschnittlich großen Team entspricht, nicht unterschreiten (Chefprogrammierer, drei Klassenbesitzer und ein Fachexperte) und ist nach oben unbegrenzt. Das Rollenmodell erlaubt an dieser Stelle eine hohe Anzahl an Chefprogrammierer, Klassenbesitzer und Fachexperten, das in der richtigen Proportion eine hohe Parallelisierung der Entwicklungsarbeiten mehrerer Features durch viele Teams und dadurch hohe Skalierung möglich macht (Vgl. [Esse11]).

Die Stärke des kleinsten Projekteteams beträgt demnach sieben Personen. Einem Projekt dieser Größenordnung wäre aber die Mächtigkeit von FDD, das sehr gute Mechanismen für Größe und Komplexität aufweist, nicht adäquat.

Besonders das hierarchische FDD-Rollenmodell macht demzufolge eine ***sehr gute*** Skalierung möglich.

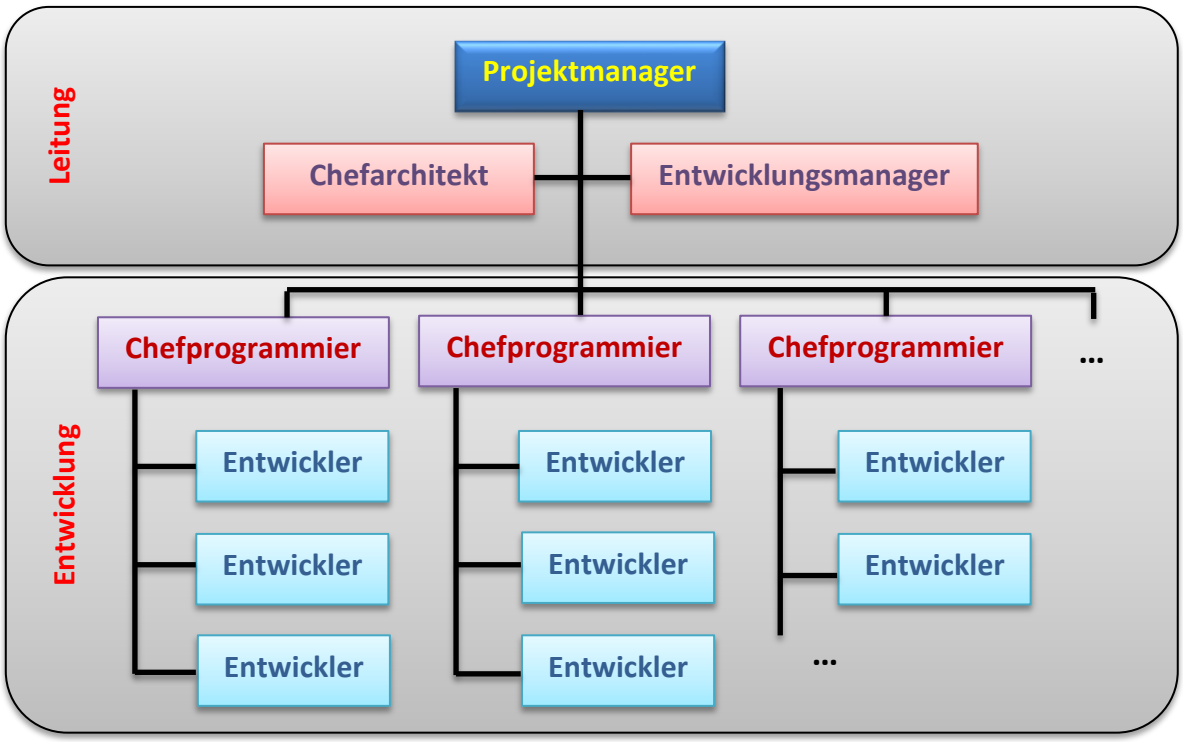

Abbildung 12: FDD-Rollenhierarchie

4.5.4.4 Dokumentationsaufwand

In den ersten beiden Projektphasen wird der Umfang der geforderten Funktionalitäten beschlossen und es entsteht das Gesamtmodell des Systems, das schrittweise bis zur Implementierung verfeinert wird. Dieser Entwurf sollte ausführlich und nachvollziehbar dokumentiert werden. Besonders bei vielen beteiligten Entwicklern ist das unverzichtbar, denn die mündliche Weitergabe der Informationen bis zu der untersten Ebene (den Entwicklern) ist fehleranfällig und ineffizient.

Zahlreiche weitere Dokumente entstehen während der Entwicklung, wie die Feature-Liste, Sequenzdiagramme, aus Inspektionen entstandene Protokolle und die Designentwürfe der Features aus Phase vier. Eine zusätzliche Rolle des *Technical Writers* verfasst die Dokumentation für die Benutzer, z.B. das Benutzerhandbuch.

Die Menge der erforderlichen Artefakte ist bei FDD im Vergleich zu den anderen untersuchten Vorgehensmodellen ***groß***.

4.5.4.5 Qualitätssicherungsmaßnahmen

Die ausgiebigen Analyse- und Planungsphasen durch hoch qualifizierte Schlüsselrollen minimieren das Risiko der Fehlerkosten. Die Entwickler testen ihren Code mit Unit Tests

selbst, die Rolle des Testers ist nach dem FDD-Rollenmodell nur eine zusätzliche Rolle. Das Zerlegen des Gesamtsystems in kleinere Bereiche bis auf die einzelnen Methodenrümpfe, wie das in der vierten Projektphase Entwirf je Feature passiert, verhindert schon bei der Planung die Entstehung überflüssiger Methoden. Regular Build Schedule sorgt bei regelmäßigem Builden und regelmäßiger Integration des Codes für reibungsloses Zusammenspiel der Klassen und der Komponenten.

In Inspektionen kontrollieren sich Entwickler gegenseitig, dass als eine der effektivsten aber auch zeitaufwendigsten Arten der Fehlererkennung erscheint. Durch Class Ownership können die Klassenbesitzer bei gefundenen Fehlern Änderungen an ihrem eigenen Code schneller durchführen und somit mögliche Fehldeutungen beim Lesen fremden Codes vermeiden. Außerdem, wenn jemand als einziger für etwas verantwortlich ist, ist er mehr dazu geneigt höherwertigere Ergebnisse zu produzieren.

Die in FDD vorgesehenen Maßnahmen zur Qualitätssicherung, die in Form von Unit-Tests von den Entwicklern durchgeführt werden und die zwar zeitaufwendigen, aber umso effektiveren Inspektionen werden als umfangreich und ***gut*** bewertet.

4.5.4.6 Flexibilität bei Anforderungsänderungen

FDD wirkt zunächst durch die definierte Abfolge und keine beabsichtigten Sprünge zwischen den Phasen wasserfallartig und unflexibel bei wechselnden Anforderungen. Die Flexibilität ist dennoch gegeben. Die relativ kurzen Phasen von zwei Wochen ermöglichen zeitnahe Reaktion auf Änderung der Rahmenbedingungen. Diese Zeitspanne und die ausgiebige Planung machen dieses Vorgehen schwergewichtiger als andere. Gleichzeitig ist es aber auch resistenter gegen Störungen des Entwicklungsprozesses, z.B. durch den Kunden, der sich bezüglich einer Funktionalität oft anders entscheidet (Vgl. [ItAg07]).

Besonders die letzten zwei Phasen (die einer Iteration entsprechen) unterstützen die agilen Werte, wonach auf Änderungen der Anforderungen kurzfristig und effektiv reagiert werden kann (Vgl. [Buns08], S. 145).

Trotz der gegebenen Agilität ist FDD auf größere Festpreisprojekte mit festem Umfang zugeschnitten. Der Kunde kennt anfangs alle Anforderungen, daher bleiben seine Einflussmöglichkeiten während des Projekts eingeschränkt. Somit ist die Flexibilität bei Anforderungsänderungen nicht so stark gefordert und nur ***ausreichend*** gegeben.

4.5.4.7 Kundeninvolvierung

Der Kunde ist als Domänenexperte nach FDD-Rollenmodell die Schlüsselrolle. Er ist hauptsächlich als Informationsquelle in der ersten Planungsphase beim Modellieren der

Domäne, woraus später die einzelnen Features abgeleitet werden, in das Projektgeschehen involviert. Als letzte Tätigkeit überprüft er die Ergebnisse einer Iteration, indem ihm das implementierte und getestete Feature vorgeführt wird.

In den letzten zwei Entwicklungsphasen hat der Kunde nur eingeschränkt Einfluss auf das Projektgeschehen. Demnach wird der Kundeneinbezug nach FDD als ***ausreichend*** gewichtet.

4.5.5 Fazit

FDD ist in Deutschland weniger verbreitet. Die Ansätze sind besonders ansprechend für klassisch organisierte Unternehmen und große Projekte mit ausgiebigen Analysephasen, bei denen die Selbstorganisation von Teams schwierig ist. Die Phasen für die Planung und Entwicklung sind strikt vordefiniert. Das kann von vielen Unternehmen als eine Einschränkung in ihrer Handlungsweise angesehen werden, das, allem Anschein nach, den Grund für das geringe Interesse daran wiedergibt (Vgl. [ItAg07]; [WoRo11] S. 17).

Die Einführung in klassische Organisationsgliederung erfolgt leichter, da die Prozesse und die Rollen mit den oft bereits vorhandenen Unternehmensstrukturen gut übereinstimmen. Trotz des wasserfallartigen Ablaufs der fünf klar definierten Phasen ist das Vorgehen leichtgewichtig, flexibel und agil. Der Kunde bekommt, im Vergleich zum Wasserfallmodell, relativ schnell die fertigen Features zum Ausprobieren vorgestellt und nicht alles auf einmal als Gesamtsystem erst alles am Projektende. Für Projekte mit festem Anforderungskatalog, einem Festpreis und einem heterogenen Team mit speziellen Fähigkeiten ist das Vorgehensmodell besonders erfolgsversprechend. Die Entwickler können unerfahren sein, wodurch die Entwicklungskosten niedrig gehalten werden, aber die Hauptrollen müssen mit Kompetenz und Erfahrung gefüllt sein.

FDD wurde bereits erfolgreich in großen und komplexen Projekten eingesetzt (Vgl. [Buns08], S. 145). Bei vielen beteiligten Entwicklern, die für hohe Produktivität sorgen, macht das FDD-Rollenmodell die Koordination beherrschbar. Keine automatischen Integrationstests sind Pflicht, verboten sind diese aber auch nicht. Für den wesentlichen Anteil an Qualität sorgen die Inspektionen des Designs in den frühen und des Codes in den späteren Projektphasen.

Kapitel 5

Schlussbetrachtung

Ein Artikel berichtet über das alljährliche Treffen der „Agile 2011“, der weltgrößten Konferenz zum Thema Agilität (Vgl. [Ecks11]). An diesem Treffen nahmen dieses Jahr über 1600 Interessente aus 43 Ländern teil. Im Jahr 2009 waren das knapp 1300 Teilnehmer aus 40 Ländern (Vgl. [Ecks09]). Das zeigt, wie erfolgreich die „Agile Bewegung“ ist und wie schnell sie wächst. Bei der Konferenz hat man die Gelegenheit sich mit Gleichgesinnten, und mit Glück auch mit den Autoren des agilen Manifests, auszutauschen und zu diskutieren. Leider hat der Erfolg auch eine negative Seite. So werde es laut J. Eckstein jedes Mal schwieriger „das eigene Netzwerk zu pflegen und sich zufällig mit anderen Teilnehmern zu treffen.“[21]

Eine Umfrage ergab, dass agil nach den Angaben von 71% und somit von zwei Drittel der Befragten hauptsächlich bei Projekten zur Individualentwicklung vorgegangen würde. Agil würden Web-Projekte zu 72%, Produktentwicklung zu 70%, eingebettete Systeme zu 55% und Entwicklung von Standardsoftware zu 47% durchgeführt ([Wolf08], S. 12-13).

Einige der agilen Praktiken sind nicht ganz unumstritten. Standup-Meetings erzeugen beispielsweise einen sozialen Druck auf die, die in der Zwischenzeit weniger geschafft haben. Oder die Tatsache, dass der Kunde die anstehenden Anforderungen beliebig entfernen und ergänzen kann, macht den gesamten Projektfortschritt und damit das Projektende schlecht vorhersehbar.

Ein anderes Beispiel sind die User Stories. Sie werden bei XP und Kanban zum Sammeln der Anforderungen benutzt, die in Worten des Kunden die gewünschte Funktionalität beschreiben.[22] Die umfassen jedoch keine genauere Anforderungsbeschreibung, was dem Entwickler großen Spielraum lässt. Wenn der Kunde beim Gespräch mit dem Entwickler die Funktionalität ungenügend beschrieben und sich aber viel mehr darunter vorgestellt hat, kann das für Projekte mit einem festen Umfang ein Risiko darstellen und später als

[21] Vgl. J. Eckstein: „Treffen der Agilen“, [Ecks11].
[22] Bei Scrum und Crystal-Familie ist diese Technik zum Sammeln der Anforderungen nicht explizit verboten.

Nichterfüllung der Vertragsbedingungen ausgelegt werden. Die Use Cases, die bei Crystal-Methodenfamilie eingesetzt wird, bieten dem eine gelungene Alternative. Die beschreiben bestimmte Interaktionen zwischen den verschiedenen Benutzern und dem System. Dies ist eine ausführlichere Variante der Anforderungserfassung und lässt sich in Form von Use Case-Diagrammen grafisch darstellen.

Die Technik Customer On-Site (eine Zusammenarbeit des Kunden mit dem Team vor Ort) kann aus der Sicht des Auftraggebers ein unnötiger Kostenfaktor sein. Das kann außerdem als Konsequenz haben, dass die Systemeigenschaften, wie z.B. die Benutzeroberfläche, nur den persönlichen Vorlieben dieser Person entsprechen. Damit ist aber zumindest sichergestellt, dass das System wenigstens einer Gruppe von Benutzern entspricht, das wiederum besser ist, als wenn es gar keinem gefallen würde. Der Kunde, als ein hoher Kostenfaktor, muss nicht nur passiv den Entwicklern bei der Arbeit zusehen, sondern er kann vor Ort seinen eigenen Tätigkeiten nachgehen und lediglich für die Fragen verfügbar sein (Vgl. [Stey10], S. 61).

Die Skalierung agiler Methoden ist nicht immer und nicht nach jedem Vorgehensmodell möglich. M. Cohn vertritt die Ansicht, dass mehr beteiligte Personen auch deutlich mehr Fehler verursachen würden. Das würde durch eine Studie über Produktivität von Teams unterschiedlicher Größe rausgefunden und in „Communications of he ACM“ beschrieben ([Cohn10], S. 212). Jedoch bietet beispielsweise FDD durch sein Rollenmodell gute Ansätze für Organisation mehrerer parallel arbeitender Teams, die besonders ein Festpreisprojekt enorm beschleunigen können.

Der Aufwand für Dokumentation gilt es bei kleinen Teams ganz nach den Werten des agilen Manifests minimal zu halten. Für größere Projekte mit mehreren Teams ist es jedoch ratsam, die wichtigen Entscheidungen ausführlich zu dokumentieren. Auf diese Weise müssen nicht alle Beteiligten an allen Meetings teilnehmen, was ab einer gewissen Anzahl nicht machbar ist, und so kann jeder für sich die wichtigen Informationen einholen. Der Grundgedanke von Crystal mit seinem hierarchischen Rollenmodell ist in diesem Zusammenhang besonders interessant.

Die Einführung eines neuen Vorgehensmodells muss gut geplant und umgesetzt werden. Kanban fordert keine sofortige Umstellung aller bestehenden Arbeitsabläufe und Rollen und erfolgt in einem sanften Wechsel. Die Veränderung findet durch die Mitarbeiter selbst auf eine evolutionäre und sehr transparente Weise statt. Die Methodik konzentriert sich unter anderem auf die Verkürzung der Durchlaufzeiten und somit schnellere Auslieferung der Software, weswegen die sich genauso für agil, wie auch für klassisch arbeitende Unternehmen eignet. Das Team wird durch Limitierungen und das Pull-Prinzip vor Überlastung geschützt. Das unterscheidet Kanban von den anderen untersuchten Vorgehensmodellen. Im Gegensatz dazu besteht Scrum auf sofortige

Umsetzung und genaue Einhaltung der vorgesehenen Richtlinien und Rituale, was für die Umstellung ein gutes Coaching und viel Offenheit der Mitarbeiter erfordert.

Im Allgemeinen können evolutionäre Techniken, die einem Team schrittweise dabei helfen die Prozesse der Entwicklung selbstständig zu analysieren und zu verbessern, einen besser verständlichen und akzeptierten Übergang zu einer anderen Arbeitsweise ermöglichen. Falls alles von heute auf morgen nach einem neuen, bis aufs kleinste Detail durchgeplanten Vorgehen anders gemacht werden muss, könnte das schnell auf Wiederstand und Unverständnis stoßen.

Die Ergebnisse der Untersuchung sind in Tabelle 1 zusammengefasst und Tabelle 2 führt weitere Merkmale auf. Erfolgsversprechend scheinen die Vorgehensmodelle zu sein, die sich durch die Beteiligten an die jeweilige Projektart anpassen lassen. Die früher mit Erfolg eingesetzten Praktiken verschiedener Modelle werden zu einem eigenen zusammengefasst, wodurch neue Kombinationen, wie das Scrum-Ban (Vgl. [Core08]) oder XP@Scrum entstehen, die es noch weiter zu untersuchen gilt. Das Team, das ca. 10 Personen umfasst, entscheidet selbst über den Einsatz bestimmter Praktiken, die sich aus seiner Sicht bewährt haben und aussichtsreich sind. Die regelmäßigen Retrospektiven über den Erfolg dieser Praktiken ermöglichen weitere Anpassung und haben einen Lerneffekt zufolge. Die Technologien, wie die Versionsverwaltung, Continuous Integration und die Automatisierung der Tests gehören bereits zum Standard und sind ein fester Bestandteil der modernen Softwareentwicklung.

	Extreme Programming	Scrum	Crystal-Familie	Kanban	Feature Driven Development
Einarbeitungszeit	Lange	Lange	Kurz	Kurz	Sehr kurz
Transparenz	Sehr gut	Sehr gut	Sehr gut	Gut	Ausreichend
Skalierbarkeit und verteiltes Arbeiten	Ausreichend	Gut	Ausreichend	Mangelhaft	Sehr gut
Dokumentationsaufwand	Sehr gering	Groß	Gering	Sehr gering	Groß
Qualitätssicherungsmaßnahmen	Sehr gut	Sehr gut	Gut	Sehr gut	Gut
Flexibilität bei Anforderungsänderungen	Sehr gut	Sehr gut	Gut	Sehr gut	Ausreichend
Kundeninvolvierung	Sehr stark	Stark	Ausreichend	Schwach	Ausreichend

Tabelle 1: Ergebnisse der Bewertung

Weitere Eigenschaften	XP	Scrum	Crystal-Familie	Kanban	Feature Driven Development
Kunde vor Ort	Ja, als Teil des Teams.	Ja, in der Rolle des Product Owners als Schnittstelle zum Kunden.	Möglich, sonst steht er ausreichend zur Verfügung.	Undefiniert.	Bestenfalls ja, in manchen Phasen Teil des Teams.
Rollenmodell	Fest. -Projektleiter, Kunde, Entwickler (evtl. XP-Coach)	Fest. - Product Owner, Entwicklungsteam, ScrumMaster	Fest, wachsend mit steigender Komplexität des Projekts. Mindestens: - Auftraggeber, erfahrener Anwender, Chefdesigner, Designer/Programmierer.	Undefiniert, anfangs bestehende Rollen werden übernommen.	Fest. -Projektmanager, Chefarchitekt, Entwicklungsmanager, Chefprogrammierer, Klassenbesitzer, Domänenexperte. Außerdem noch möglich sind unterstützende und zusätzliche Rollen.
Projektarten	Projekte mit wagen, sich schnell ändernden Anforderungen. Schnelle Lieferung von Teilfunktionalitäten.	Kleine, große und komplexe Projekte jeglicher Art.	Auswählbar und an die Projektart anpassbar.	Wartung und Betrieb.	Größere Fixpreisprojekte mit festgelegtem Umfang.

Tabelle 2: Weitere Eigenschaften

Glossar

Akzeptanztest	Das sind von dem Kunden vorgegebene Bedingungen oder Testfälle, die als Abnahmekriterien für die Software dienen. Sie können zwischen Arbeitgeber und Arbeitnehmer vertraglich festgelegt werden.
Backlog	Eine Sammlung von Produktanforderungen, wie z.B. bei Scrum, die der Planung dient. Sie wird geführt als eine Liste mit den Angaben, wie die Anforderungs-ID, die Beschreibung oder die Aufwandschätzung.
Best Practices	Eine Ansammlung von nützlichen und sich bewährten Methoden und Verfahren.
Code Coverage	Abdeckung der ausgewerteten Informationen. Für die Softwaretests gibt das den Überdeckungsgrad und somit die Anzahl der ausgeführten Codezeilen im Verhältnis zu der Gesamtanzahl.
Domäne	Die Domäne der Software beschreibt ihre spätere Umgebung, z.B. die vorhandenen Nachbarsysteme.
Fibonacci-Zahl	Diese Zahl entsteht bei einer unendlichen Zahlenfolge, die die folgende Zahl durch Addition der letzten vorherigen berechnet.
Funktionalität	Mit diesem Begriff ist eine von der Software zu erfüllende (Teil-) Aufgabe gemeint.
Iteration	Ein Zyklus während der Entwicklung, der nach der Durchführung beinhaltender Abläufe ein Teilprodukt ergibt.

Offshoring	Geografische Verlagerung der Geschäftsprozesse eines Unternehmens zur Kostensenkung an eine Niederlassung im Ausland.
Outsourcing	Organisatorische Auslagerung von Geschäftsprozessen eines Unternehmens zur Kostensenkung an spezialisierte Drittunternehmen ins Ausland.
Produktvision	Es ist eine begeisterungsfähige Ideenbeschreibung, um die Projektbeteiligten zu motivieren und ein Ziel vorzugeben. Sie wird bei Scrum durch Product Owner in der Anfangsphase eines Projekts erstellt.
Wertschöpfungskette	So werden nacheinander ablaufende Prozesse zur Fertigstellung eines Produkts genannt.

Quellenverzeichnis

[Alis02] Alistair Cockburn, *Agile Software Development*, Addison-Wesley Verlag, 2002.

[Beck04] K. Beck und C. Andres, *Extreme Programming Explained – Embrace Change*, 2nd Ed. Addison-Wesley Verlag, 2004.

[Buns08] Christian Bunse, Antje von Knethen, *Vorgehensmodelle kompakt*, 2. Auflage, Spektrum Akademischer Verlag, 2008.

[Cohn10] Mike Cohn, *Agile Softwareentwicklung - Mit Scrum zum Erfolg!*, Addison-Wesley Verlag, 2010.

[Comp11] www.computerwoche.de, *Agile Methoden im Vergleich*, 28.12.2011, http://www.computerwoche.de/software/software-infrastruktur/2352712/ (Feb. 2012).

[Core08] Ladas Corey, *Lean Software Engineering – Essays on the Continuous Delivering of High Quality Information Systems*, 2008, http://leansoftwareengineering.com/ksse/scrum-ban/ (Feb. 2012).

[Ecks09] Jutta Eckstein, *Agile 2009 - Der Krise getrotzt*, Sept. 2009, http://heise.de/-849320 (Feb. 2012).

[Ecks11] Jutta Eckstein, *Agile 2011 - Treffen der Agilen*, Aug. 2011, http://heise.de/-1330264 (Feb. 2012).

[Eile10] Karl Eilebrecht und Gernot Starke, *Patterns kompakt*, 3. Auflage, Spektrum-Verlag, 2010.

[Eppi11] Thomas Epping, *Kanban für die Softwareentwicklung*, Springer-Verlag, 2011.

[Esse11] A. Esser, S. Achtelik, N. Ernst, Fallstudienarbeit: *Beschreibung und Analyse von Feature Driven Development*, Hochschule für Oekonomie & Management Düsseldorf, 2011, http://winfwiki.wi-fom.de/index.php/Beschreibung_und_Analyse_von_Feature_Driven_Development#cite_note-p.26fs29-12 (Feb. 2012).

[Glog09] Boris Gloger, *Scrum – Produkte schnell und zuverlässig entwickeln,* Hanser Verlag, 2009.

[Glog11] Boris Gloger, André Häusling, *Erfolgreich mit Scrum – Einflussfaktor Personalmanagement*, Carl Hanser Verlag München, 2011.

[Hans10] Eckhart Hanser, *Agile Prozesse: Von XP über Scrum bis MAP*, eXamen.press Springer-Verlag Berlin Heidelberg, 2010.

[Hrus09] Peter Hruschka, Chris Rupp, Gernot Starke, *Agility Kompakt*, 2. Auflage, Spektrum Akademischer Verlag Heidelberg, 2009.

[ItAg07] Whitepaper von www.it-agile.de, *Agil mit FDD*, 2007, http://www.it-agile.de/fileadmin/docs/Whitepaper-FDD.pdf (Feb. 2012).

[ItAg08] Jeff De Luca, übersetzt von Stefan Roock für www.it-agile.de, *Feature Driven Development (FDD)*, 2008, http://www.it-agile.de/fileadmin/docs/FDD-Prozesse.pdf (Feb. 2012).

[Jeff01] Ronald E. Jeffries - An Agile Software Development Resource, *Essential XP: Documentation,* 2001, http://xprogramming.com/articles/expdocumentationinxp/ (Feb. 2012).

[Kapp10] U. Kapp, J. P. Berchez, *Kriterien für eine Entscheidung für Scrum oder Kanban – Brüder im Geiste*, 02.09.2010, Teil 2 aus http://heise.de/-1071172 (Feb. 2012).

[Leop11] Klaus Leopold, *Von anderen lernen wollen – Treffen der europäischen Kanban-Community*, 31.10.2011, http://heise.de/-1368408 (Feb. 2012).

[Logo08] Doina Logofătu, *Grundlegende Algorithmen mit Java*, Vieweg-Verlag, 2008.

[Mani01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, *Manifest für agile Softwareentwicklung*, 2001, http://www.agilemanifesto.org/iso/de/ (Feb. 2012).

[Neum11] Alexander Neumann, *Qualitätssicherung in der Softwareentwicklung wird professioneller*, 14.09.2011 - KW 37, http://heise.de/-1342486 (Feb. 2012).

[Pich08] Roman Pichler, *Scrum – Agiles Projektmanagement erfolgreich einsetzen,* dpunkt.verlag, 2008.

[Rasm10] Jonathan Rasmusson, *The Agile Samurai*, Pragmatic Programmers, 2010.

[Rump01] Bernhard Rumpe, *Extreme Programming-Back to Basics?*, in: *Modellierung 2001*, Workshop der Gesellschaft für Informatik e.V.(GI) 28.-30.3.2001, Bad Lippspringe. pp. 121-131, GI-Edition, Lecture Notes in Informatics, 2001.

[Scha10] A. Schatten, M. Demolsky, , D. Winkler , S. Biffl, E. Gostischa-Franta, Th. Östreicher, *Best Practice Software-Engineering – Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeuge*, Spektrum Akademischer Verlag Heidelberg, 2010.

[ScrG11] Ken Schwaber, Jeff Sutherland: *Scrum Guide – The official rulebook*, Oktober 2011, http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20DE.pdf (Feb. 2012).

[Sixt03] Mario Sixtus, *Gemeinsam auf die Spitze treiben*, in: DIE ZEIT, 22.12.2003, http://www.zeit.de/2004/01/T-Extremprogrammierer/komplettansicht (Feb. 2012).

[Stey10] Manfred Steyer, *Agile Muster und Methoden*, entwickler.press, 2010.

[Suth11] Jeff Sutherland, Anton Viktorov, Jack Blount, Nikolai Puntikov, *Distributed Scrum: Agile Project Management with Outsourced Development Teams*, Paper aus: *The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework* von Jeff Sutherland und Ken Schwaber, 2011, http://jeffsutherland.com/ScrumPapers.pdf (Feb. 2012).

[Tach09] Emanouel Tachtsoglou, Rafael Arndt, *Dokumentationsanforderungen im IT-Projektmanagement*, 26.01.2009, http://winfwiki.wi-fom.de/index.php/Dokumentationsanforderungen_im_IT-Projektmanagement (Feb. 2012).

[Wiki11] Wikipedia, *Liste von Softwareentwicklungsprozessen*, 09.11.2011, http://de.wikipedia.org/wiki/Liste_von_Softwareentwicklungsprozessen (Feb. 2012).

[Wiki12] Wikipedia, *Agile Softwareentwicklung*, 06.02.2012, http://de.wikipedia.org/wiki/Agile_Softwareentwicklung (Feb. 2012).

[Wolf08] H. Wolf, A. Roock, *Agilität wird Mainstream: Ergebnisse der Online-Umfrage 2008,* in: OBJEKTspektrum, Mai/Juni 2008, Nr. 3, http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2008/03/marktstudie_OS_03_08.pdf (Feb. 2012).

[Wolf11] Henning Wolf, Wolf-Gideon Bleek, *Agile Softwareentwicklung – Werte, Konzepte und Methoden*, 2. Auflage, dpunkt.verlag, 2011.

[WoRo11] Henning Wolf, Arne Roock, *Agile Softwareentwicklung - Ein Überblick*, 3. Auflage, dpunkt.verlag, 2011.

Versicherung über Selbstständigkeit

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

Hamburg, den _____ _____