Ludwig-Maximilians-Universität München

Institut für Informatik
Lehr- und Forschungseinheit Programmierung und Softwaretechnik

Diplomarbeit

Sicherstellung von Qualität in der agilen Softwareentwicklung

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Christian Kroiß
Andreas Wolf (GIGATRONIK München GmbH)

Bearbeiter: Matthias Wolf
Abgabedatum: 30. September 2009

Ludwig-Maximilians-Universität München

Institut für Informatik
Lehr- und Forschungseinheit Programmierung und Softwaretechnik

Erklärung zur Diplomarbeit

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Diese Arbeit hat bisher noch keiner anderen Prüfungskommission zur Begutachtung vorgelegen.

München, 30.09.2009 ……

(Unterschrift des Kandidaten)

Abstract

Eine gängige Kritik an agilen Softwareentwicklungsmethoden lautet, dass sich mit ihnen nicht die gleiche hohe Softwarequalität erreichen lässt wie mit klassischen, nicht-agilen Methoden. Diese Kritik ist mit Sicherheit nicht haltbar, da Verfechter der agilen Entwicklung bereits zahlreiche Beispiele für erfolgreiche und qualitativ hochwertige Projekte angeführt haben. Die Frage ist also nicht ob, sondern vielmehr wie Qualität in der agilen Softwareentwicklung sichergestellt werden kann.

In dieser Diplomarbeit werden die signifikanten Qualitätsfaktoren der agilen Softwareentwicklung herausgearbeitet. Im Anschluss daran wird ein konkreter agiler Prozess entwickelt, der die Überwachung und Beeinflussung dieser Qualitätsfaktoren optimal unterstützt.

Inhaltsverzeichnis

1 Einleitung

Die Firma GIGATRONIK bietet an fünf Standorten in Deutschland und Österreich Entwicklungsdienstleistungen im Bereich der Automobilelektronik und Informationstechnologie an. Zu den Kunden der Firma gehören vorrangig Automobilhersteller und deren Zulieferbetriebe, die hohe Ansprüche an die Softwarequalität stellen.

Die Abteilung *Umweltsysteme* ist spezialisiert auf Entwicklung und Betrieb von Software für Stoffmanagement und Compliancemanagement. Typische Projekte der Abteilung werden oft unter Zeitdruck in kleinen Teams implementiert und anschließend über Jahre betrieben, gewartet und weiterentwickelt. Aktuell werden zur Softwareentwicklung ein klassischer, schwergewichtiger Softwareentwicklungsprozess und die bereits etwas in die Jahre gekommene Technologie *Oracle Forms* eingesetzt. Die Abteilung erwägt, zukünftig agile Methoden einzusetzen und auf eine modernere Technologieplattform zu wechseln. Diese Entscheidungen sind jedoch mit dem Risiko verbunden, dass die bisherige Qualität nicht beibehalten werden kann.

Agile Methoden erlauben eine hocheffiziente Entwicklung in kleinen Teams und ermöglichen eine schnelle Auslieferung von funktions- und leistungsfähiger Software. Allerdings wird agile Softwareentwicklung allzu häufig als Wundermittel gepriesen, ohne zu beleuchten, welche Voraussetzungen überhaupt geschaffen werden müssen, damit erfolgreich agil entwickelt werden kann, und wann es überhaupt sinnvoll ist, agile Methoden einzusetzen. Nicht zuletzt aufgrund dieser unzureichenden Auseinandersetzung mit der Materie verlaufen agile Projekte immer wieder unbefriedigend und bestätigen Kritiker aufs Neue in ihren Einwänden, agile Softwareentwicklung sei chaotisch, nicht planbar und bringe Projekte mit unklarer Qualität hervor.

Die vorliegende Arbeit soll mit Vorurteilen aufräumen, echte Schwächen aufzeigen und konkrete Vorschläge für die Sicherstellung von Qualitätsanforderungen in einem agilen Entwicklungsprozess machen. Der Schwerpunkt der Arbeit liegt dabei auf der Frage, über welche Faktoren sich die Qualität in der agilen Softwareentwicklung sicherstellen lässt. Dabei sollen die wichtigsten „Stellschrauben“ herausgearbeitet und ihre Relevanz und Beeinflussbarkeit eingeordnet werden.

Zunächst einmal wird in Kapitel 2 kurz darauf eingegangen, aus welcher Situation die agilen Softwaremethoden hervorgingen, und was sie von den klassischen Methoden unterscheidet. In Kapitel 3 werden zunächst ausgewählte Methoden vorgestellt, verglichen und eingeordnet. Im Anschluss daran wird von einzelnen Methoden abstrahiert und eine allgemeinere Übersicht über Stärken und Grenzen „der agilen Methoden“ gegeben.

Das vierte Kapitel bildet den theoretischen Schwerpunkt der Arbeit, hier wird nun die konkrete Qualitätssicherstellung in der agilen Softwareentwicklung beleuchtet. Dabei muss zunächst die Frage nach der Definition von Softwarequalität erörtert werden, bevor im Kapitel 4.1 die signifikanten Qualitätsfaktoren erarbeitet werden. In Kapitel 4.2 wird auf Zertifizierungsmöglichkeiten für Personal und Prozesse eingegangen.

Kapitel 5 ist der Entwicklung eines qualitätsorientierten agilen Prozesses gewidmet. Da der Prozess möglichst kompatibel zum bestehenden Entwicklungsprozess der Firma GIGATRONIK sein soll, wird dieser zunächst vorgestellt. In Kapitel 5.2 wird der agile Prozess auf Basis von *Scrum* schließlich beschrieben. Die eigens für diesen Prozess erstellten Checklisten können im Kapitel 5.3 eingesehen werden.

Um Erfahrungen mit der von GIGATRONIK favorisierten neuen Technologieplattform und ihrem Einfluss auf die Qualität der Software sammeln zu können, wurde im Rahmen dieser Arbeit außerdem ein praktisches Entwicklungsprojekt durchgeführt. Im sechsten Kapitel wird das Projekt zunächst beschrieben, anschließend werden die Erfahrungen ausgewertet und kritische Qualitätsfaktoren analysiert.

2 Entstehung der agilen Softwareentwicklung

Zu Beginn des Computerzeitalters war die Beschaffung von Hardware noch um ein Vielfaches teurer als die Entwicklung von Software. Die Ein- und Ausgabe erfolgte überwiegend auf Lochkarten. Die meisten Programme waren daher auch nur Berechnungs- oder Sortieraufgaben, die zumeist in Form der Stapelverarbeitung abgearbeitet wurden. Erst mit der Einführung von Terminals, die eine direkte Eingabe auf einer Tastatur und eine direkte Ausgabe auf einem Bildschirm ermöglichten, wurden echte „Anwendungsprogramme“ möglich, bei denen die Endbenutzer einfach in Interaktion mit dem Rechner treten konnten.

Trotzdem war die Anzahl der Benutzer noch sehr klein, und auch der Umfang der Programme war durch Rechenleistung, Speicherkapazität und weitere Einschränkungen der Hardware äußerst begrenzt. Programmierer konnten daher in enger Zusammenarbeit mit den Benutzern ein Problem aufarbeiten, eine Lösung implementieren und diese gemeinsam mit dem Kunden so lange testen und weiterentwickeln, bis das Programm den Vorstellungen entsprach.[1]

Durch die breite Einführung von integrierten Schaltungen in der Computertechnik in den 1960er Jahren wurde die Hardware auf einen Schlag um ein Vielfaches leistungsfähiger und günstiger. In der Folge waren plötzlich deutlich ambitioniertere und komplexere Softwareprojekte möglich, an deren Realisierung viele Programmierer parallel arbeiteten. Für diese Herausforderung war die Softwareindustrie jedoch nicht gewappnet. „Große Projekte waren oft Jahre im Rückstand. Die Software kostete oft mehr als vorhergesehen, war unzuverlässig, schwer zu pflegen und arbeitete mangelhaft. Die Softwareentwicklung befand sich in einer Krise.“[2]

2.1 Die Suche nach der Antwort auf die Softwarekrise

Als direkte Reaktion auf die Softwarekrise wurde 1968 auf der NATO-Konferenz in Garmisch-Partenkirchen von namhaften Größen aus Forschung und Industrie der Begriff „Software Engineering“ geprägt. Im Konferenzbericht wird deutlich, dass sich die Teilnehmer eine neue Herangehensweise an die Softwareentwicklung wünschten: „The phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.“[3]

In Anlehnung an die etablierten Ingenieurdisziplinen wurden in der Folge Vorgehensmodelle und Methoden entwickelt, die auf ausführlicher Planung und Dokumentation basierten. Barry Boehm bezeichnet diese Methoden als plangesteuert: „Plan-driven methods are characterized by a systematic engineering approach to software that carefully adheres to specific processes in moving software through a series of representations from requirements to finished code.“[4]

[1] vgl. Schwaber 2007, S. 56
[2] Sommerville 2007, S. 30
[3] Naur et al., S. 8
[4] Boehm, Turner 2008, S. 10

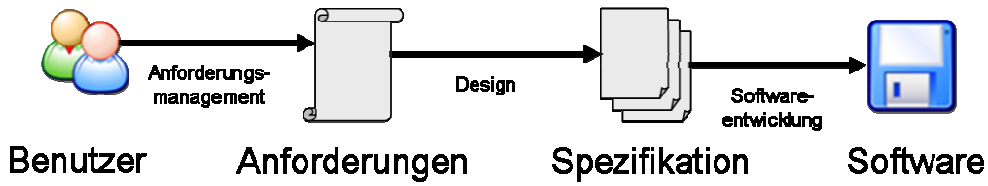

Abb. 1: In plangesteuerten Methoden wird durch Prozesse ein Plan erstellt und umgesetzt. Anforderungen, Spezifikation und Software sind verschiedene Repräsentationen desselben Konzepts.

Die plangesteuerte Denkweise führte zur Aufteilung der Belegschaft von Softwarefirmen in das Projektmanagement, das Pläne erstellt und die Ausführung überwacht, und in Arbeiter, die Pläne umsetzen. Eine weitere Auswirkung war die Spezialisierung und Standardisierung der einzelnen Teilaufgaben. So entstanden beispielsweise Jobs wie der des Anforderungsmanagers, des Software-Architekten und des Software-Testers. Große Softwareprojekte mit vielen beteiligten Entwicklern wurden damit unabhängiger von der Leistung Einzelner und stetig besser planbar und messbar.

Die Projekte waren jedoch äußerst anfällig gegenüber unzureichend oder falsch erfassten Anforderungen. Selbst wenn diese Anforderungen perfekt umgesetzt wurden, war das Ergebnis im schlimmsten Fall wertlos. Große Schwierigkeiten bereiteten auch Änderungen an den Anforderungen und damit am Plan während der laufenden Entwicklung. Um diesen Problemen entgegen zu wirken, wurde in viele Prozesse ein umfangreiches Anforderungs- und Änderungsmanagement integriert.

Der industrialisierten Softwareentwicklung auf der Basis von plangesteuerten Methoden sind viele wegweisende Softwaresysteme zu verdanken, darunter moderne Betriebssysteme und Steuerungen für sicherheitskritische Anlagen. Aufgrund dieser Erfolge etablierten sich die Vorgehensmodelle und auf ihnen basierende Softwareentwicklungsprozesse in der gesamten Branche. Umfangreiche plangesteuerte Methoden wurden in der Folge auch bei kleinen und kleinsten Softwareprojekten eingesetzt. Heute ist es auch üblich, dass Firmen oder die öffentliche Hand ihren Auftragnehmern konkret vorschreiben, welches Vorgehensmodell bei der Entwicklung von Software oder Systemen zu verwenden ist.

Diese Methoden schaffen jedoch auch einen großen Abstand zu den Endanwendern und einen gewaltigen Management- und Dokumentationsoverhead. „Aus einer Kommunikation von Angesicht zu Angesicht wurde eine Kommunikation auf Basis von Dokumentationen. Schnelle Richtungsänderungen mutierten zu langwierigen Phasen, in denen Anforderungen zusammengetragen wurden.“[5] Wegen der Standardisierung der Arbeitsabläufe und der Schnittstellen blieb vor allem die kreative Teamarbeit auf der Strecke. Die zunehmende Bürokratie veränderte zudem die Arbeitsbedingungen der Softwareentwickler: „People caught up in heavy process can [...] run the risk of becoming documentation generators instead of software developers.“[6]

Auch heute noch weisen große Projekte die typischen Symptome der Softwarekrise auf, obwohl die Softwarefirmen und Systemhäuser inzwischen umfangreiche plangesteuerte Management- und Entwicklungsprozesse einsetzen. Immer wieder werden Budgets und Zeitpläne überschritten oder Anforderungen nicht erfüllt. Besonders auffällig ist dabei, dass sich die Probleme im Vorhinein überhaupt nicht abzeichnen oder möglicherweise sogar intern so lange gezielt unterdrückt werden, bis

[5] Schwaber 2007, S. 56–57
[6] Boehm, Turner 2008, S. 13

sie unter zunehmendem Druck öffentlich werden und das Projekt mit einem Schlag zum Desaster wird. Als prominente jüngere Beispiele seien hier das 2001 komplett gescheiterte BKA-Informationssystem „INPOL-neu“ der Firmen debis und T-Systems[7], sowie das erst nach über einem Jahr Verspätung im Januar 2005 zumindest bedingt funktionsfähige deutsche LKW-Maut-System des Konsortiums Toll Collect (Daimler AG, Deutsche Telekom und Cofiroute)[8] in Erinnerung gerufen.

2.2 Agile Softwareentwicklung als Gegenentwurf

Durch die Standardisierung und Trennung von Rechnerarchitekturen und Programmiersprachen und dank integrierter Entwicklungsumgebungen, Off-The-Shelf-Komponenten, Frameworks und Middleware stehen Softwareentwicklern heute Möglichkeiten und Werkzeuge zur Verfügung, die eine in der Geburtsstunde des Software Engineering noch unvorstellbare Geschwindigkeit und Flexibilität bei der Entwicklung möglich machen.

„Auslöser der agilen Methoden war das Gefühl, durch Bürokratie, Prozesse und Dokumentationsvorgaben gelähmt und gehemmt zu werden. Gerade schnelle, risikoreiche und damit auch chancenreiche Projekte schienen darunter zunehmend zu leiden.“[9] Dies betraf auch kleinere Softwareprojekte, die in den 1970er Jahren im Gegensatz zu den Großprojekten übrigens gar nicht unter der Softwarekrise litten.[10] Unter diesem Eindruck machten sich verschiedene unzufriedene Projektmanager und Softwareentwickler in den 1990er Jahren zunächst unabhängig voneinander auf die Suche nach Alternativen zu den etablierten plangesteuerten Methoden.

Die agile Methode „Scrum“ wurde vor allem durch die Wissensmanagement-Forschung von Hirotaka Takeuchi und Ikujiro Nonaka geprägt. Jeff Sutherland, der geistige Vater von Scrum, berichtet: „We talked to a lot of leaders in the industry and at the end of the day we found the 1986 Takeuchi and Nonaka paper, ‘The New New Product Development Game’. They describe exactly how the best teams in the world were set up, how the management gave them incentives, how they empowered them, let them work on their own, challenged them with an aggressive goal but that they self had to figure out how to achieve it.“[11]

Alistair Cockburn blieb bei seinen Recherchen im Auftrag von IBM näher an der Softwareentwicklung. Er interviewte viele erfolgreiche Teams und analysierte ihre Arbeitsweise. „Die Quintessenz war in den meisten Fällen die gleiche:

- Bringe die Leute an einen Tisch und lasse sie oft und engagiert miteinander kommunizieren.
- Schaffe bürokratische Barrieren aus dem Weg und lasse sie gestalten.
- Binde einen betroffenen Endanwender mit ein.
- Stelle eine gute automatisierte Regressionstest-Suite bereit.
- Produziere so früh und so oft wie möglich auslieferungsreife Funktionalität.“[12]

Im Gegensatz zu den plangesteuerten Methoden steht bei keinem der agilen Ansätze ein umfassender Plan im Zentrum des Entwicklungsprozesses. Stattdessen finden Planung und Entwicklung bis zur Auslieferungsreife in kurzen Iterationen statt. Der

[7] vgl. Dietl 2003, S. 191–200
[8] vgl. Grässlin 2005, S. 114–117
[9] Schneider 2007, S. 183
[10] vgl. Naur et al., S. 9
[11] Moe et al. Juli 2009, S. 2
[12] Cockburn, Giesecke 2005, S. 15

Entwicklungsalltag ist durch intensive Kommunikation und produktive Arbeit an der Software geprägt.

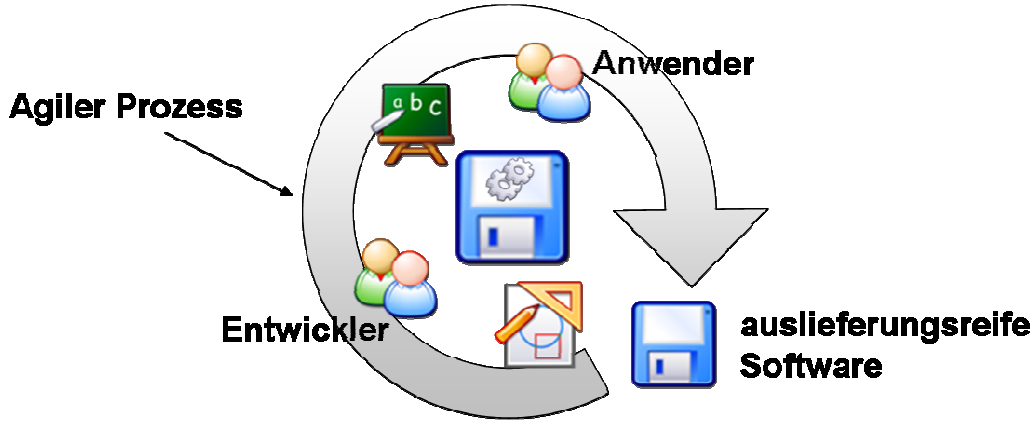

Abb. 2: In agilen Methoden steht kein umfassender Plan mehr im Zentrum des Entwicklungsprozesses.

Eine iterative Vorgehensweise wurde schon 1968 auf der NATO-Konferenz als erfolgreiches Beispiel beschrieben: „Design and implementation proceeded in a number of stages. Each stage was typified by a period of intellectual activity followed by a period of program reconstruction. Each stage produced a useable product and the period between the end of one stage and the start of the next provided the operational experience upon which the next design was based.“[13] Interessanterweise wurde auch schon damals das immanente Problem der plangesteuerten Methoden angesprochen: „The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job.“[14] Die agilen Methoden verbinden eine Alternative zu diesem problematischen Konzept mit einer iterativen Vorgehensweise.

Der Unterschied zu plangesteuerten Methoden mit iterativem Vorgehensmodell wird in der Praxis vor allem durch das Selbstverständnis der Entwickler deutlich. In plangesteuerten Prozessen erinnert die Arbeit an die eines Bandarbeiters. Aufgaben sind genau definiert und konkrete Arbeitsanweisungen liegen in Form von Plänen vor, die gewissenhaft umgesetzt werden müssen. In Analogie zur industriellen Fertigung von Produkten könnte man diese Arbeitsweise auch als *Softwarefertigung* bezeichnen.

In agilen Prozessen sehen sich die Entwickler dagegen eher als ein interdisziplinäres Team von Handwerkern, das in enger Zusammenarbeit mit dem Kunden ein Werkstück entwickelt: „In an agile culture, the people feel comfortable and empowered when there are many degrees of freedom available for them to define and work problems. This is the classic craftsman environment, where each person is expected and trusted to do whatever work is necessary to the success of the project.“[15]

[13] Naur et al., S. 11
[14] Naur et al., S. 21
[15] Boehm, Turner 2008, S. 49–50

3 Agile Methoden

In den 1990er Jahren entstand eine Vielzahl von agilen Methoden. Jeff Sutherland und Ken Schwaber begannen ab 1993 mit der Entwicklung von *Scrum*, Kent Beck erdachte 1995 das *Extreme Programming* (XP). Ebenfalls 1995 wurde die *Dynamic Systems Development Method* (DSDM) geboren. 1997 ersann Jim Highsmith das *Adaptive Software Development* (ASD) und Jeff de Luca das *Feature Driven Development* (FDD). Alistair Cockburns Methodenfamilie *Crystal* wurde erstmals 1998 praktisch eingesetzt. Obwohl alle Methoden als „agil" bezeichnet wurden, wiesen sie große Unterschiede in der Herangehensweise, in der Zielgruppe, in ihrem Umfang und in der Art und Weise der Beschreibung auf.

Begründer und Unterstützer agiler Methoden trafen sich 2001 im US-Bundesstaat Utah, um über Gemeinsamkeiten und Unterschiede ihrer Ansätze zu diskutieren. Aus dieser Konferenz gingen das agile Manifest („Agile Manifesto") und die „Prinzipien der agilen Softwareentwicklung"[16] hervor, die als gemeinsame Grundlage verstanden werden sollen. In aktueller „agiler" Literatur wird immer wieder auf das agile Manifest verwiesen und beschrieben, wie die agilen Prinzipien in der jeweiligen Methode umgesetzt werden.

Das agile Manifest trifft die folgende Gewichtung:

Wichtig	Wichtiger
Prozesse und Tools	Individuen und Interaktion
detaillierte Dokumentation	funktionierende Software
Vertragsverhandlungen	Zusammenarbeit mit dem Kunden
einem Plan folgen	sich Änderungen anpassen

Tab. 1: Die Gewichtung des agilen Manifests

Das agile Manifest stellt ein selbstorganisiertes, interdisziplinäres, motiviertes und kommunikatives Team in den Mittelpunkt der Softwareentwicklung. Das Team arbeitet konsequent mit dem Kunden zusammen. Oberstes Ziel ist die frühe, schnelle und regelmäßige Lieferung von lauffähiger, für den Kunden wertvoller Software. Diese Software ist gleichzeitig das primäre Mittel zur Messung des Projektfortschritts. Die Entwicklung selbst fußt auf fundiertem Fachwissen, gutem, aber möglichst einfachem Design, ausführlicher direkter Kommunikation und schneller Reaktion auf sich ändernde Anforderungen. Außerdem reflektiert das Team regelmäßig über seine Arbeit, um die bestehenden Prozesse und Verfahren zu verbessern und damit seine Effektivität zu erhöhen.

3.1 Vorstellung ausgewählter Methoden

In dieser Arbeit werden *Crystal Clear*, *Scrum* und *Extreme Programming (XP)* vorgestellt und näher untersucht. Die Methoden wurden im Rahmen einer Vorrecherche ausgewählt, bei der sowohl die Bekanntheit als auch der methodische Ansatz betrachtet wurden.

Crystal Clear ist Teil der „Crystal"-Methodikfamilie und eher ein Rahmenwerk zur Erstellung von agilen Methoden als eine konkrete Ausprägung. Scrum konzentriert sich vor allem auf das Projektmanagement und erfreut sich wegen diverser Zertifizierungsmöglichkeiten derzeit einer großen Beliebtheit. Extreme Programming

[16] vgl. Beck et al. 2001

ist heute die wohl bekannteste agile Methode überhaupt und wird oft sogar als Synonym für agile Softwareentwicklung verwendet. Wie der Name schon andeutet, ist XP allerdings eine sehr extreme Ausprägung, weshalb diese Gleichsetzung problematisch ist.

3.1.1 Crystal Clear

Crystal ist in der Sprache von Alistair Cockburn eine auf einem genetischen Code basierende Methodikfamilie für Projekte, in denen sich Fehler nicht lebensbedrohlich auswirken. Aus der Familie können unterschiedliche Familienmitglieder für verschieden umfangreiche Projekte erzeugt werden.

Crystal basiert auf dem *ökonomisch-kooperativen Spielmodell.*[17] Dieses Modell betrachtet die Softwareentwicklung als eine Reihe von Spielen, in denen zwei Ziele um die Ressourcen konkurrieren: Die Fertigstellung der Software in dieser Runde und die Vorbereitung auf das nächste Spiel. Im Zentrum der Methodikfamilie steht also die Abwägung der wichtigen Frage, wie kurz- oder weitsichtig Planung, Design, Entwicklung und Dokumentation innerhalb einer Iteration betrieben werden. Ein zu kurzfristig orientiertes Verhalten bringt zwar schnelle Erfolge, erschwert aber die weitere Entwicklung. Ein zu langfristig orientiertes Verhalten bremst und behindert dagegen die Geschwindigkeit und Flexibilität der Entwicklung. Das richtige Verhalten soll das Entwicklungsteam in Crystal durch Kommunikation, Kooperation und wachsende Erfahrung in jeder Runde neu aushandeln.

Crystal Clear ist dabei das Familienmitglied für Teams von drei bis acht Entwicklern. Da das Team relativ klein ist, werden keine speziellen Kommunikationsprozesse etabliert. Es wird nur gefordert, dass *osmotische Kommunikation*[18] stattfinden kann. Konkret soll das Team in einem Raum oder mehreren miteinander verbundenen Räumen arbeiten, umfangreiche Whiteboard-Flächen zur Verfügung haben und sich jederzeit spontan austauschen können. Darüber hinaus fordert Crystal Clear, dass regelmäßige Lieferungen verwendbarer Software stattfinden und dass die Arbeitsweise des Teams reflektierten Verbesserungen unterliegt. Um sicherzustellen, dass die Vorteile dieser Arbeitsumgebung ausgenutzt werden und die Forderungen von Crystal Clear im Rahmen des ökonomisch-kooperativen Spieles umgesetzt werden, muss mindestens ein Teammitglied, der *Chefdesigner-/programmierer*, sehr erfahren sein und über ausgeprägte kommunikative Fähigkeiten verfügen.

Ein Team, das diese Forderungen umsetzt, praktiziert bereits Crystal Clear. Die Methode kann jedoch durch vier weitere optionale Eigenschaften und durch den Einsatz von Techniken an die Erfordernisse konkreter Projekte angepasst werden.

Die vier zusätzlichen Eigenschaften beschreiben makroorganisatorische Umstände, die das Funktionieren von Crystal Clear vereinfachen. Diese Eigenschaften sind: Persönliche Sicherheit (frei die eigene Meinung äußern zu können), Bildung von Schwerpunkten, einfache Kontaktaufnahme mit den Endanwendern sowie eine technische Umgebung mit automatisierten Tests, Konfigurationsmanagement und häufigen Integrationen.

Die zusätzlichen Techniken sind mikroorganisatorische Mittel, die das Team bei der Durchführung des ökonomisch-kooperativen Spieles unterstützen. Es handelt sich dabei um von Alistair Cockburn zusammengestellte „Best Practices“ aus der agilen

[17] vgl. Cockburn, Giesecke 2005, S. 249
[18] vgl. Cockburn, Giesecke 2005, S. 57–62

Softwareentwicklung. Er beschreibt Techniken zur Methodikgestaltung, tägliche Standup-Meetings und Reflexions-Workshops, Techniken für die Planung, Aufwandsschätzung und Fortschrittskontrolle, für das Interaktionsdesign und die Side-by-Side-Programmierung, sowie das „Miniaturverfahren", eine Technik zur schnellen Gewöhnung des Teams an agile Abläufe. Crystal Clear erlaubt es aber auch ausdrücklich, abgewandelte oder ganz andere Techniken einzusetzen, wenn sie zielführend sind.

3.1.2 Scrum

Scrum ist eine agile Projektmanagement-Methode für die Produktentwicklung, deren vorrangiges Ziel die möglichst frühe Lieferung von Produktteilen mit möglichst hohem geschäftlichem Nutzen ist. Eine zentrale Bedeutung kommt daher in Scrum der Rolle des *Product Owners* zu.[19] Dieser erfasst und priorisiert in Zusammenarbeit mit dem Kunden die Anforderungen im *Product Backlog*. Die Projektteile mit dem potentiell höchsten geschäftlichen Nutzen werden ausführlich beschrieben und stehen ganz oben. Die Liste kann auch später noch jederzeit erweitert, gekürzt oder neu geordnet werden.

Das *Team* aus maximal 6-8 Entwicklern ist verantwortlich für die Planung und Entwicklung des Produktes in *Sprints*, Iterationen von maximal 30 Tagen Dauer. Vor einem Sprint bespricht das Team die obersten Anforderungen aus dem Product Backlog mit dem Kunden und übernimmt anschließend die Einträge, deren komplette Realisierung innerhalb eines Sprints möglich scheint, in das *Sprint Backlog*. „Komplett realisiert" beinhaltet in der Regel, dass die Funktionalität geltenden Standards und Vorgaben entspricht, getestet und dokumentiert ist und ohne weitere Anpassungen im Realeinsatz verwendet werden kann. Das Team verpflichtet sich dazu, genau diese Funktionalität bereitzustellen. Um Planungssicherheit zu gewährleisten, kann das Sprint Backlog während eines Sprints nicht mehr verändert werden.

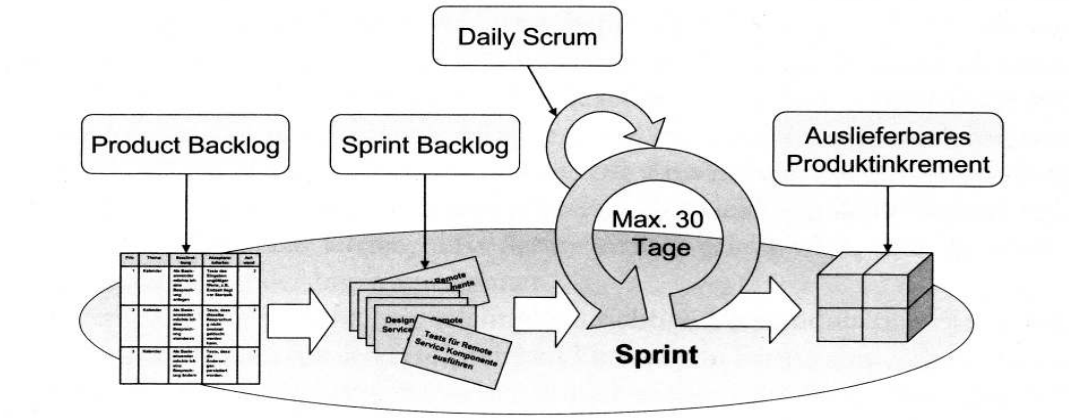

Abb. 3: Scrum im Überblick[20]

Nach jedem Sprint führt das Team dem Kunden im *Sprint Review* die Teile des Sprint Backlogs vor, die tatsächlich komplett realisiert werden konnten. Unfertige Produktteile, Prototypen oder Konzepte dürfen nicht vorgeführt werden. Die Erfahrungen aus dem Sprint sind damit sowohl für den Kunden als auch für das Team sofort nutzbar. Auf der Basis des Sprint Reviews kann der Kunde zusammen mit dem Product Owner Änderungsforderungen in das Product Backlog aufnehmen und die Anforderungen neu priorisieren. Das Team kann die tatsächliche Arbeitsgeschwindigkeit, eventuell aufgetretene Probleme und das verbesserte Verständnis der Projektmaterie bei der

[19] vgl. Pichler 2008, S. 10
[20] Pichler 2008, S. 7

Planung des nächsten Sprints berücksichtigen.

Scrum sieht noch zwei zusätzliche Meetings vor:

1. Ein kurzes tägliches Meeting während des Sprints, das *Daily Scrum*. Ein Ziel des Meetings ist es, die Arbeit des Teams zu synchronisieren, indem abgeglichen wird, an was die Teammitglieder gerade konkret arbeiten und was sie planen. Zusätzlich sollen Hindernisse (Impediments) innerhalb des Teams und des Unternehmens erkannt und in der Folge beseitigt werden.
2. Das *Sprint Retrospective* Meeting, in dem das Team nach jedem abgeschlossenen Sprint die eigenen Arbeitsprozesse kritisch hinterfragt und Verbesserungen beschließt.

Die Moderation dieser Meetings ist eine der Aufgaben des *Scrum Masters*. Darüber hinaus fungiert dieser als Coach für die richtige Anwendung von Scrum durch das Team und das Unternehmen. Er achtet darauf, dass im Sprint Review tatsächlich nur fertige Funktionalität präsentiert wird und sorgt für die umgehende Beseitigung aller Hindernisse.

3.1.3 Extreme Programming (XP)

Mit der Entwicklung von Extreme Programming verfolgte Kent Beck das Ziel, allgemein als sinnvoll erachtete Entwicklungspraktiken ins Extrem zu steigern, um Geschwindigkeit und Flexibilität von Teams bis an die Grenze des Machbaren zu bringen.[21] Daraus erwuchs eine ganze Reihe von sehr restriktiven Vorschriften und Vorgehensweisen. Um die Ressource Mensch unter diesen Bedingungen zu schonen und das Leistungsniveau nicht auszuhöhlen, schreibt XP explizit die Einhaltung fairer Arbeitszeiten vor.

Tabelle 2 zeigt die Wurzeln einiger XP-Entwicklungspraktiken:

Praktik	Steigerung in XP
Kundenorientierung	Kunde ist Teil des Teams (Customer On Site)
Code Reviews	Code Reviews finden ständig statt (Paarprogrammierung)
Tests	Ständige automatisierte Tests
Design	Design wird zum Teil der täglichen Arbeit (Refactoring)
Einfachheit	Implementierung des einfachsten funktionierenden Designs
Integrationstests	Täglich mehrfache Integration und Test
Kurze Iterationen	Ultrakurze Iterationen (Minuten/Stunden) beim Planning Game, häufiger Wechsel von Programmierpartnern

Tab. 2: Entwicklungspraktiken und ihre extremen Steigerungen im Extreme Programming

XP eignet sich für Teams mit 2-12 Entwicklern und setzt auf eher kurze Iterationen von ein bis drei Wochen Dauer. Die Methode zeichnet sich durch eine sehr enge Einbeziehung des Kunden, eine bis in die kleinsten Abläufe iterative Handlungsweise sowie starke gegenseitige Kontrolle aus. Planung, Design und Programmierung finden kollektiv statt, der dabei entstandene Code gehört allen und kann auch von jedem geändert werden.

Eine Besonderheit ist der *Customer On Site*, ein kompetenter und entscheidungsbefugter Anwender, der Vollzeitmitglied des Teams ist. Er soll Transparenz schaffen, schon bei kleinsten alltäglichen Entscheidungen mitwirken und Missverständnisse vermeiden.

[21] vgl. Sommerville 2007, S. 432

Vor jeder Iteration wird beim *Planning Game* in Zusammenarbeit mit dem Kunden die als nächstes zu entwickelnde Funktionalität in einem iterativen Verfahren ausgewählt und geschätzt. XP schreibt vor, dass die Funktionalität mit *User Stories* beschrieben wird, sehr informellen und kurzen Beschreibungen. Die Formulierung auf dem Papier dient nur als Gedankenstütze für die Entwickler, entscheidend ist die gedankliche Vorstellung vom System, die während der Diskussion mit dem Kunden und den anderen Teammitgliedern entsteht. Auf dieser Basis werden die User Stories vom Team in *Engineering Tasks* aufgeteilt. Jedes Teammitglied verpflichtet sich dann zur Umsetzung bestimmter Tasks innerhalb der nächsten Iteration.

Auf die folgenden *Engineering Practices* sei noch gesondert hingewiesen:

- **Paarprogrammierung**: Die Entwickler finden sich in häufig wechselnden Paaren zusammen, um gemeinsam vor einem Rechner zu entwickeln. Sie wechseln in schneller Folge zwischen aktiver Entwicklung und Kontrolle. Dadurch wird die Diskussion über Design-Entscheidungen angeregt, ein ständiges Code Review durchgeführt sowie Detailwissen über die Implementierung im Team verteilt.
- **Tests zuerst**: Entwickler schreiben Tests für ihre Engineering Tasks, bevor sie eine Lösung implementieren. Dadurch stehen die Tests sofort zur Überprüfung der Implementierung zur Verfügung und es wird verhindert, dass dem Code minderwertige Tests „auf den Leib geschneidert“ werden.
- **Einfaches Design und laufendes Refactoring**: YAGNI („You ain’t gonna need it”) heißt die Devise, die fordert, dass nur die einfachste funktionierende Lösung implementiert wird. Das Design soll sich explizit auf das aktuelle Problem und die aktuelle Situation beziehen. Allerdings sollen die Entwickler durch Refactoring auch jederzeit das Gesamtdesign verbessern und Redundanzen im Code beseitigen.
- **Fortlaufende Integration**: Änderungen werden mehrmals täglich in ein zentrales Repository integriert. Automatisierte Integrations- und Regressionstests sollen dabei sicherstellen, dass durch die Änderungen keine neuen Fehler auftreten. Wenn Tests fehlschlagen, müssen die Fehler sofort beseitigt werden.
- **Code ist Dokumentation**: Es gilt die Maxime „ask the code“. Bei Unklarheiten liegt die Antwort nicht in begleitenden Dokumenten, sondern einzig und allein im Quelltext oder in daraus automatisch erstellten Dateien. Dieses Vorgehen löst auf extreme Weise das Synchronisationsproblem zwischen Dokumenten und Code, das bei häufigen Änderungen zwangsläufig auftritt. Das heißt jedoch nicht, dass komplett auf Dokumentation verzichtet wird. Im Gegenteil: ohne einen sauberen und penibel dokumentierten Quellcode ist dieser Ansatz zum Scheitern verurteilt.

Jeder Tag der Iteration beginnt mit dem *Stand Up Meeting*, in dem das Team sich gegenseitig auf den aktuellen Stand bringt. Jedes Teammitglied schildert kurz woran es arbeitet und ob Probleme aufgetreten sind. Das Meeting findet im Stehen statt, um den informellen Charakter und die angepeilte kurze Dauer zu betonen.

XP sieht zwei Rollen vor, die über den Prozess wachen. Der *Tracker* beobachtet und visualisiert den Fortschritt des Teams und greift bei problematischen Verzögerungen steuernd ein. Der *Coach* hilft bei Problemen, vermittelt im Team und achtet darauf, dass die XP-Praktiken konsequent angewendet werden.

3.2 Vergleich und Einordnung

Um die vorgestellten Methoden auf einem abstrakteren Niveau vergleichen zu können, wurden Grundgedanke und Prozesstyp sowie die Vorgaben zum Projektmanagement und der konkreten Entwicklungstätigkeit betrachtet. In **Tabelle 3** sind die Eckdaten der drei Methoden kurz zusammengefasst.

	Crystal Clear	Scrum	Extreme Programming
Grundgedanke	Ökonomisch-kooperatives Spielmodell	maximaler geschäftlicher Nutzen	maximale Flexibilität und Produktivität
Prozesstyp	adaptiv, empirisch	adaptiv, empirisch	adaptiv, restriktiv
Projektmanagement	empfohlene Techniken	vorgeschriebene Techniken	vorgeschriebene Techniken
Entwicklung	empfohlene Techniken	empfohlene Techniken	vorgeschriebene Techniken

Tab. 3: Die drei vorgestellten agilen Methoden in der Übersicht

Im Gegensatz zu den *prädiktiven* plangesteuerten Methoden beruhen die vorgestellten agilen Methoden auf *adaptiven* Prozessen. Sie müssen keinerlei Vorhersagen treffen, an denen der Prozess ausgerichtet wird, sondern sind von Grund auf so gestaltet, dass sie sich schnell an veränderte Rahmenbedingungen anpassen.

Crystal Clear und Scrum sind zudem *empirische* Prozesse. Aus den Erfahrungen früherer Iterationen werden Verbesserungen des Prozesses abgeleitet, eingesetzte Techniken und Arbeitsweisen können angepasst oder ausgetauscht werden. Extreme Programming ist dagegen eher *restriktiv*. Da die XP-Praktiken wie Zahnräder ineinander greifen sollen und nur in Kombination wirklich sinnvoll sind, bietet die Methode kaum Spielraum zur Anpassung. Die einzige naheliegende Konsequenz aus negativen Erfahrungen ist, dass XP vom Team, der Technik oder der Organisation noch nicht gut genug umgesetzt wird.

Crystal Clear bietet mit Abstand die größte Freiheit bei der Anpassung des Prozesses auf ein konkretes Projekt. Diese Freiheit eröffnet jedoch auch viele Möglichkeiten, einen potentiell ungeeigneten, überflüssigen oder gar kontraproduktiven Prozess zu erschaffen. Es wird zwar gefordert, dass die Arbeitsweise des Teams durch reflektierte Verbesserungen angepasst wird, aber auch dieser Prozess kann natürlich ineffizient gestaltet werden. Der erfahrene Chefdesigner/-programmierer ist in Crystal Clear vorgesehen, um durch seine Expertise derartige Probleme zu verhindern.

Scrum geht einen Mittelweg zwischen Extreme Programming und Crystal Clear. Während im Bereich der Entwicklungstätigkeit kaum Vorgaben gemacht werden, ist ein konkreter Prozess für das Projektmanagement definiert. Dieser Prozess organisiert das Zusammenspiel der Rollen, die täglichen Abläufe und die Prozessverbesserung um den Grundgedanken der Maximierung des geschäftlichen Nutzens. Innerhalb dieses stabilen Rahmens kann Scrum flexibel an verschiedenste Projekte angepasst werden.

Beim Vergleich der Methoden fällt auf, dass sie sich wegen ihrer unterschiedlichen

Schwerpunkte nicht gegenseitig ausschließen. Im Gegenteil: Crystal Clear ist in so geringem Maße konkret formuliert, dass sich die Projektmanagement-Mechanismen von Scrum problemlos darin integrieren lassen. Alistair Cockburn schreibt selbst: „Scrum und Crystal passen gut zueinander“.[22] Da bei Crystal Clear und Scrum die Entwicklungspraktiken nicht konkret vorgeschrieben sind, können XP-Praktiken wiederum in beide Methoden integriert werden.

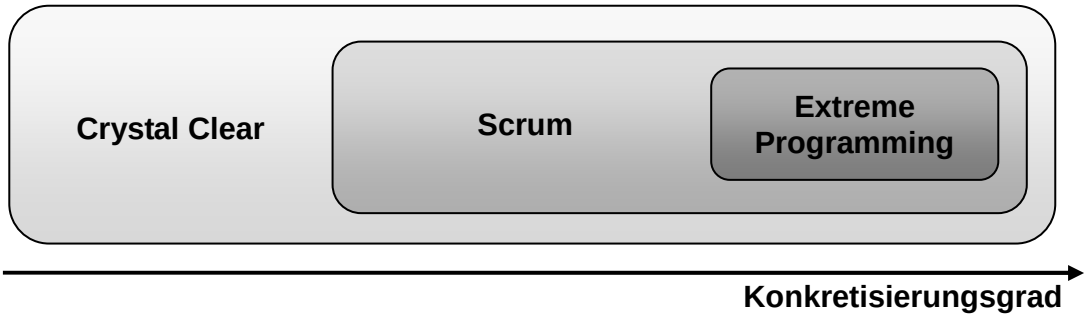

Abb. 4: Einordnung von Crystal Clear, Scrum und Extreme Programming

Durch die Integration einer anderen Methode verschiebt sich die Prioritätsgewichtung der äußeren Methode jedoch in Richtung der enthaltenen Methode. Cockburn macht deutlich, dass bei einer Integration von Scrum oder gar Extreme Programming in Crystal Clear die Gewichtung des ökonomisch-kooperativen Spiels deutlich in Richtung des „kurzfristigen Erfolgs“ verschoben wird und damit die rechtzeitige Fertigstellung eines Gesamtprojektes gefährdet werden kann. Aus der Sicht von Scrum ist das selbstverständlich gewollt: Es geht ja genau darum, die geschäftswichtige Funktionalität zuerst zu implementieren und ein Projekt möglicherweise abzubrechen, wenn nur noch unwichtige Punkte offen sind.

Es wird deutlich, dass die verschiedenen agilen Methoden trotz ihrer unterschiedlichen Grundgedanken und Prioritäten durchaus kompatibel sind. Unter Wahrung der notwendigen Differenzierung werden im nächsten Kapitel gemeinsame Stärken und Grenzen untersucht.

3.3 Stärken und Grenzen agiler Methoden

Viele Bücher von Agilisten lesen sich wie Motivationsbücher oder weltanschauliche Schriften. Nicht selten vermitteln die Autoren den Eindruck, die jeweilige Methode sei nicht nur die einzig richtige Art und Weise Software zu entwickeln, sondern außerdem auch noch der Weg zu persönlichem Glück, Reichtum, einem erfüllten Leben und beliebigen anderen Zielen. Tatsächlich gibt es Erfolgsgeschichten, in denen agile Teams unter bestimmten Rahmenbedingungen hochproduktiv arbeiteten und in Rekordzeit beachtliche Erfolge erzielten. Nur wenige Autoren setzen sich jedoch auch wirklich kritisch mit den Grenzen der eigenen Ansätze und der agilen Softwareentwicklung an sich auseinander.

Experten aus den Reihen der klassischen, plangesteuerten Softwareentwicklung sparen zwar nicht mit Kritik, doch nur wenige sind auch bereit, den agilen Paradigmenwechsel wirklich anzuerkennen und eine objektive Bewertung vorzunehmen. Viele Autoren machen auch den Fehler, bestimmte agile Methoden (zum Beispiel das dazu bekanntermaßen extrem ungeeignete Extreme Programming) einfach mit „der agilen Softwareentwicklung“ gleichzusetzen und ziehen aus dieser Annahme gänzlich unzutreffende Schlüsse.

[22] Cockburn, Giesecke 2005, S. 269

Um zu entscheiden, ob für ein Projekt agile Methoden in Frage kommen, ist es jedoch wichtig, die Stärken und Grenzen der agilen Entwicklung zu kennen. Eines der wenigen objektiven Bücher zu diesem Thema, wenn nicht gar das bisher einzige, ist „Balancing Agility and Discipline (A Guide for the Perplexed)“ von Barry Boehm und Richard Turner.[23] Boehm und Turner wägen die Stärken und Schwächen des plangesteuerten und des agilen Entwicklungsansatzes gegeneinander ab, ohne dabei grundsätzlich Partei für eine Seite zu ergreifen oder unzulässig zu generalisieren. Sie kommen zu folgendem Fazit: „[We] have concluded that there are five critical factors involved in determining the relative suitability of agile or plan-driven methods in a particular project situation. These factors [...] are the project's size, criticality, dynamism, personnel, and culture factors.“[24]

In den folgenden Unterkapiteln werden diese fünf Faktoren untersucht und konkrete Stärken und Grenzen agiler Methoden herausgearbeitet.

3.3.1 Projektgröße

Bei der Untersuchung der agilen Methoden ist deutlich geworden, dass die Entwicklungsteams sich üblicherweise in einer Größenordnung von unter 10 Personen bewegen. Erfahrungswerte haben gezeigt, dass diese Art von Selbstorganisation nur in kleinen Gruppen funktioniert. Größere Gruppen teilen sich üblicherweise unbewusst in kleinere Teilgruppen auf, wobei die Kommunikation zwischen den einzelnen Teilgruppen jedoch fast komplett eingestellt wird. Der Hauptgrund für dieses Verhalten liegt darin, dass Kommunikationskosten bei der linearen Erhöhung der Anzahl der Kommunikationspartner quadratisch ansteigen. Da agile Entwicklungsteams einen hohen Synchronisations- und Kommunikationsbedarf haben, sind sie jedoch auf Umstände angewiesen, die „osmotische Kommunikation“ nach Alistair Cockburn erlauben.

Agile Teams können also nicht beliebig vergrößert werden. Agilisten weisen zwar ständig darauf hin, dass selbst ein kleines agiles Team um ein vielfaches produktiver und schneller sein soll als ein großes plangesteuertes Team, aber früher oder später ergibt sich für ein einzelnes Team aus der Größenbegrenzung eine obere Schranke der Projektgröße.

Um auch bei größeren Projekten agile Methoden einsetzen zu können, ist es notwendig, die Mitarbeiter in mehreren kleinen Teams zu organisieren. Crystal Orange, Cockburns Methodik für Projekte mit etwa 40 Mitarbeitern, teilt die Mitarbeiter in sieben Teams auf, die jeweils nach Crystal Clear vorgehen. Leider gibt es derzeit keine Praxisberichte über den Einsatz von Crystal Orange, die nicht von Cockburn selbst stammen. Belastbare Erfahrungswerte liegen damit nicht vor. Anders ist die Situation bei Scrum: Auch Scrum sieht vor, mehrere Teams parallel an der Software arbeiten zu lassen. Die Koordination der Teams wird wiederum in einem eigenen Scrum-Projekt organisiert. Dadurch entsteht ein hierarchisches „Scrum of Scrums“, das laut Ken Schwaber hervorragend skaliert. Zu derartigen Projekten gibt es viele Erfahrungswerte, inzwischen haben verschiedene Autoren umfangreiche Methoden für das Management großer Projekte mit Scrum beigesteuert.[25]

Sehr große Projekte sind häufig gleichzeitig verteilte Projekte. Aus den Anforderungen des Kunden oder der Struktur von beteiligten Unternehmen ergibt

[23] Diese Quelle erwies sich als sehr hilfreich bei der Bewertung von Aussagen anderer Autoren.
[24] Boehm, Turner 2008, S. 54
[25] vgl. z. B. Pichler 2008, S. 125–158

sich oft die Situation, dass ein Projekt von Teams an verschiedenen Standorten entwickelt wird. Diese Verteilung ist ein Problem für agile Methoden, da bei der Organisation der Zusammenarbeit dieser Teams eine osmotische Kommunikation nur sehr eingeschränkt möglich ist. Mit Webcams, Videokonferenzen und Chats mag sich zwar eine Art „verdichtete Kommunikation“[26] erreichen lassen, doch die persönliche Interaktion lässt sich dadurch nicht ersetzen. Erfolgversprechende Techniken für die Kommunikation zwischen verteilten Scrum-Teams sind Botschafter, gemeinsame Besprechungen der Product Owner sowie gemeinsame Meetings zu Sprint Review und Sprint Retrospective. Es ist aber offensichtlich, dass auch diese Techniken nicht optimal sind, außerdem viel Zeit kosten und immense Reisekosten verursachen.

Agile Methoden können also auch problemlos für größere Projekte eingesetzt werden, solange sich die Teams an einem Ort oder in der Nähe befinden. Ob eine verteilte Entwicklung gelingt ist abhängig davon, wie viel Erfahrung die Mitarbeiter mit verteilter agiler Entwicklung haben. Verteilte Entwicklung ist außerdem langsamer und teurer.

3.3.2 Kritikalität

Als „kritisch“ bezeichnet man Software oder Systeme, deren uneingeschränkte und korrekte Funktion sehr wichtig ist. Als Beispiele seien medizinische Geräte, Steuerungen für Kraftwerke oder Verkehrsmittel, aber auch Systeme für Finanztransaktionen genannt. Kurz: Überall dort wo der Verlust von Menschenleben oder hohen Geldsummen auf dem Spiel steht wird erwartet, dass Software absolut zuverlässig arbeitet.

Derartige Erwartungen an Software werden in der Regel in Form von nichtfunktionalen Anforderungen formuliert. Die plangesteuerte Entwicklung ermöglicht es, solche Software risikogesteuert zu planen, formal zu spezifizieren und zu verifizieren. Wenn der Plan die Erfüllung der Anforderungen garantiert und die korrekte Umsetzung des Plans in Code dokumentiert ist, lässt sich damit die Sicherheit des Systems belegen. Da nichtfunktionale Anforderungen meist schon von Anfang an bekannt sind und kaum Änderungen unterliegen, kommen die Vorteile des plangesteuerten Ansatzes hier voll zum Tragen.

Agile Techniken kollidieren dagegen eher mit diesen Erwartungen. Da sich nichtfunktionale Anforderungen auf Gesamtsysteme beziehen, können sie nur durch einen Top-Down-Entwurf garantiert werden. Aus diesen planerischen Vorgaben erwächst jedoch eine Einschränkung der Entwickler: Bei jeder Änderung laufen sie Gefahr, die nichtfunktionalen Anforderungen zu gefährden. Am Ende der Entwicklung fehlt durch die nicht vorhandene Beweiskette von Anforderungen über den Plan bis zum Code ein belastbarer Nachweis über die Erfüllung der Anforderungen.

Für sehr kritische Projekte bieten agile Methoden derzeit keine erfolgversprechenden Ansätze. Einzig Alistair Cockburn geht am Rande der Beschreibung von Crystal Clear kurz darauf ein, dass für kritischere Projekte seine Methodiken Quarz oder Diamond geeignet seien. Diamond sei „eine Methodik, die sowohl wiederholte Überprüfungen des Designs als auch der Implementierung der Algorithmen verlangt.“[27] Leider gibt es zu dieser Methodik derzeit auch in anderen Büchern keine weiteren Erklärungen, weshalb offen bleibt, inwiefern die Methodik agil sein soll, und wie sie sich von

[26] vgl. Cockburn, Giesecke 2005, S. 58
[27] Cockburn, Giesecke 2005, S. 18

plangesteuerten Ansätzen unterscheidet.

Viele nichtfunktionale Anforderungen können allerdings auch bereits durch die Auswahl von passenden Off-The-Shelf-Komponenten erfüllt werden. So ist beispielsweise die maximale Anzahl von gleichzeitigen Benutzern eines Systems in viel höherem Maße von der Auswahl der Hardware und des Datenbanksystems abhängig als von der Implementierung der Anwendungssoftware. In so einem Fall macht es keinen Unterschied, ob plangesteuerte oder agile Methoden zum Einsatz kommen.

Bei monolithischen Softwaresystemen gilt jedoch uneingeschränkt die Zusammenfassung von Boehm und Turner: „[T]he plan-driven world is considerably ahead of the agile world in dealing with quality or nonfunctional requirements such as reliability, throughput, real-time deadline satisfaction, or scalability.“[28]

3.3.3 Dynamik

Plangesteuerte Methoden sind darauf angewiesen, dass Anforderungen einen gewissen Bestand haben. Schließlich geht der Entwicklungstätigkeit eine ausführliche Planungsphase voraus, in der genau spezifiziert wird, welche Anforderungen auf welche Weise umgesetzt werden. Wenn nun ständig Änderungen einfließen, kann die Planung nicht abgeschlossen und die Entwicklung nicht begonnen werden. Daher werden die Anforderungen üblicherweise zu einem bestimmten Zeitpunkt eingefroren. Je nach Vorgehensmodell gibt es dann keine Möglichkeit mehr für Änderungen, oder sie werden während der laufenden Entwicklung nach und nach in den Plan eingearbeitet. Dafür ist jedoch ein umfassendes Änderungsmanagement erforderlich, das die möglichen Auswirkungen einer Änderung vorab untersucht und dann vorgibt, wie mit den Abweichungen zwischen dem neuen Plan und der bestehenden Software zu verfahren ist. Ein derart ausgestalteter Prozess ist jedoch nur begrenzt leistungsfähig, verlangsamt zudem die Entwicklung und bindet Personal.

Natürlich gibt es Projekte, die sehr wenig Dynamik aufweisen oder erwarten lassen. Als Beispiel aus der Praxis[29] wäre die Erweiterung eines Personalverwaltungssystems aufgrund einer Gesetzesänderung vorstellbar, wobei die Anforderungen durch den Gesetzestext und durch die bestehende Benutzeroberfläche weitgehend vorgegeben sind. Große Überraschungen im Bezug auf Anforderungen wird es hier kaum geben.

Auf der anderen Seite gibt es aber auch Projekte, deren Anforderungen zu Beginn sehr unscharf sind oder sich schneller ändern als die Entwicklung voranschreitet. Solche Projekte bringen plangesteuerte Methoden schnell an ihre Grenzen. Der übliche Ansatz zur Klärung der Anforderungen ist die Konstruktion von Prototypen. Unmöglich wird so ein Projekt nicht, aber es zieht sich in die Länge und wird sehr schnell sehr teuer.

Genau auf solche Projekte zielt der iterative Ansatz der agilen Softwareentwicklung ab. Er basiert auf der Annahme, dass Kunden oft erst wissen was sie wollen, wenn sie es in den Händen halten. Vor einer Iteration wird nur gerade so viel geplant wie unbedingt notwendig oder sinnvoll ist. Während oder nach der Iteration kann der Kunde das Ergebnis begutachten und auf dieser Basis die weitere Entwicklung beeinflussen. Ken Schwabers Beschreibung von Scrum macht deutlich, wer in diesem Prozess das Steuer übernimmt: „Der ScrumMaster ist verantwortlich für die

[28] Boehm, Turner 2008, S. 39
[29] Für diesen Hinweis möchte ich Christoph Schütte danken.

Beseitigung jeglicher Hürden, die zwischen den Entwicklungsteams und zwischen dem Product Owner und den Kunden stehen, sodass die Kunden direkt die Entwicklung lenken können."[30]

Wie schlagen sich die agilen Methoden nun bei sehr statischen Anforderungen? Extreme Programming vertritt konsequent die Ansicht: „Und sie werden sich doch ändern" und setzt auf einfaches Design und YAGNI. Das kann ineffizient sein, weil Code geschrieben und mehrfachem Refactoring unterzogen wird, obwohl das endgültige Design von Anfang an absehbar war. Scrum beinhaltet keine solche Vorschrift, weshalb es zumindest erlaubt ist, bereits am Anfang des Projektes langfristiger zu planen. Ob das geschieht hängt aber von der Erfahrung des Teams ab. Das ökonomisch-kooperative Spielmodell von Crystal Clear gibt dagegen eine klare Antwort: Unter diesen Umständen macht es Sinn, in den ersten Spielen viele Markierungen zu setzen, da diese später die Arbeit erleichtern. Im Klartext: Da sich die Anforderungen kaum ändern, ist es im Sinne des Modells, zuerst ausführlicher zu planen und später auf dieser Grundlage zu entwickeln.

Agile Methoden sind den plangesteuerten bei hoher Dynamik des Projektes überlegen. Bei geringer Dynamik sind agile Methoden aber nur dann weniger geeignet, wenn die Planung zu kurzsichtig erfolgt. Zumindest Crystal Clear und Scrum bieten die Möglichkeit, sich an die jeweiligen Gegebenheiten des Projektes anzupassen.

3.3.4 Personal

Die Fähigkeiten von Menschen fallen sehr unterschiedlich aus. Abhängig von der grundsätzlichen und situationsabhängigen Leistungsfähigkeit bestimmen fachliche wie soziale Kompetenz und Erfahrung die Fähigkeiten eines Mitarbeiters. Die fachliche und die soziale Komponente stehen allerdings wiederum für eine Vielzahl von einzelnen Fähigkeiten, die jeweils zu einem bestimmten Grad beherrscht werden.

Umfassend fachlich kompetente und erfahrene Mitarbeiter sind gut für jedes Projekt, egal ob plangesteuert oder agil entwickelt wird. Leider sind solche Mitarbeiter eher selten. Bei der Verwendung plangesteuerter Methoden ist es möglich, durch wenige sehr kompetente Menschen einen Plan erstellen zu lassen, der anschließend von sehr vielen weniger kompetenten Entwicklern umgesetzt wird. Je mehr soziale Kompetenz die Entwickler besitzen, umso besser, aber auch hier gilt wieder: Es geht zur Not auch ohne. Der Plan wird vorgegeben und ist umzusetzen, eine Diskussion ist nicht notwendig.

Die Planung in agilen Methoden basiert dagegen hauptsächlich auf implizitem, im Team verteiltem Wissen. Um dieses Wissen in ein Design umzusetzen, ist viel Kommunikation und Abstimmung notwendig. Agile Methoden fordern daher von allen Entwicklern eine hohe soziale Kompetenz. Besonders wichtig für die Arbeit im Team sind eine offene, freundliche Art und eine ausgeprägte Kommunikationsbereitschaft. Da Planung und Design zu den täglichen Aufgaben des Teams gehören, ist außerdem ein höherer Anteil fachlich sehr kompetenter und erfahrener Mitarbeiter notwendig als bei plangesteuerten Methoden.

Ein agiles Team ist also in aller Regel hochkarätiger besetzt und damit teurer als ein plangesteuertes. Die agilen Teams haben dabei jedoch auch einen deutlichen Vorteil: In einem derart kommunikativen Umfeld mit sehr kompetenten Kollegen sammeln

[30] Schwaber 2007, S. 56

auch Einsteiger schnell die notwendige Erfahrung, um ein vollwertiges Mitglied des Teams zu werden.

3.3.5 Firmenkultur

Viele Firmen sind stark hierarchisch organisiert. Alle Mitarbeiter bekommen Aufträge von ihren Vorgesetzten, und je weiter oben ein Mitarbeiter selbst in der Hierarchie steht, umso mehr Entscheidungsgewalt erhält er. Diese Struktur passt gut zur plangesteuerten Softwareentwicklung, da aus organisatorischer Sicht die Planer über denjenigen angeordnet werden können, die den Plan ausführen. Dadurch finden sich Entwickler in einer Arbeitssituation wieder, in der die Machtverhältnisse, die Regeln und die von ihnen erwartete Arbeit genau definiert sind. Eine solche hierarchisch geprägte Firmenkultur kann den Mitarbeitern durchaus Sicherheit und Orientierung bieten.

Innerhalb eines agilen Entwicklungsteams gibt es allerdings keine Hierarchien. Sollte es Meinungsverschiedenheiten geben, zählen nur die besseren Argumente. Nur wenn allen Teammitgliedern das gleiche Vertrauen entgegengebracht wird und alle die gleichen Freiheiten besitzen, kann auch von allen der notwendige volle Einsatz erwartet werden. Natürlich können auch innerhalb einer hierarchisch organisierten Firma agile Teams eingerichtet werden. Mitarbeiter, die an eine hierarchische Kultur gewöhnt sind, fallen aber oft zurück in ihre Rollen als Befehlsgeber oder Befehlsempfänger. Agile Methoden lassen sich daher wesentlich leichter in Firmen mit flachen Hierarchien einsetzen.

Ein anderer Aspekt der Firmenkultur ist die Flexibilität von Strukturen. Manche Firmen sind in dieser Hinsicht sehr konservativ. Der Grund dafür ist verständlich: Die bestehenden Strukturen haben sich möglicherweise über eine lange Zeit bewährt, weshalb eine Änderung dieser Strukturen ein unkalkulierbares unternehmerisches Risiko darstellt. In solch einem Unternehmen ist der Einsatz von agilen Methoden jedoch problematisch. Die Prozessverbesserungsmechanismen der agilen Methoden basieren darauf, dass alle Hindernisse, die dem Team im Weg stehen, sehr schnell aufgedeckt und beseitigt werden. Diese Hindernisse betreffen nicht selten die Struktur des Unternehmens. Ken Schwaber warnt daher: „[Es ist] beim Einsatz von Scrum erforderlich, dass sich die jeweilige Firma vielen organisatorischen Änderungen unterzieht, um von allen Vorteilen von Scrum zu profitieren.“[31]

[31] Schwaber 2007, S. 76

4 Qualität in der agilen Softwareentwicklung

Der Begriff der Software-Qualität lässt sich nur schwer fassen. Je nach Perspektive bestehen sehr verschiedene Ansichten darüber, welche Faktoren eine qualitativ hochwertige Software auszeichnen. Während Anwender zumeist eine einfache Bedienung und Absturzfreiheit im Blick haben, erwarten Unternehmen in erster Linie, dass sich die Investition in die Software innerhalb der veranschlagten Zeit finanziell lohnt. Softwareentwickler haben dagegen im Hinblick auf Qualität vor allem Faktoren wie Codetransparenz, Testbarkeit, Wartbarkeit und Portierbarkeit im Blick. All diese Faktoren stehen jedoch in einem Spannungsfeld und schließen sich zum Teil sogar gegenseitig aus. Die Optimierung der Laufzeit einer Echtzeitsteuerung auf Wunsch des Unternehmens kann zum Beispiel deutliche Einschränkungen bei der Transparenz und Wartbarkeit der Software bedeuten und die Portierung unmöglich machen. Genauso kann die Bedienbarkeit einer Software darunter leiden, dass das Unternehmen eine Vielzahl von Erwartungen an sie stellt.

Hoffmann bezeichnet die Kriterien Transparenz, Übertragbarkeit, Wartbarkeit und Testbarkeit als *innere Qualitätsmerkmale*.[32] Daran angelehnt kann man die Kriterien Funktionalität, Laufzeit, Zuverlässigkeit und Benutzbarkeit als *äußere Qualitätsmerkmale* bezeichnen. Die äußeren Qualitätsmerkmale können vom Kunden direkt wahrgenommen werden, während die inneren Qualitätsmerkmale für den Kunden meist nicht sichtbar sind.

So diversifiziert die Software-Qualität auch ist, steht sie doch als Ganzes in einem weiteren Spannungsfeld. **Abbildung 5** zeigt ein Modell aus dem Projektmanagement, das die Beziehung zwischen Qualität, Kosten und Zeit verdeutlicht.

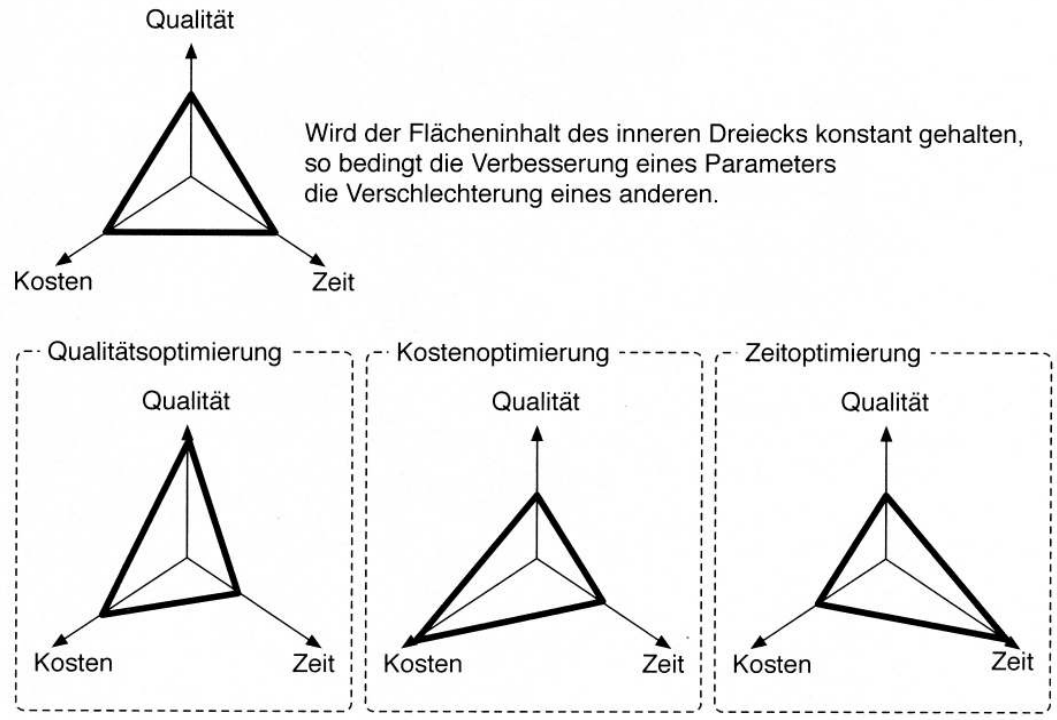

Abb. 5: Magisches Dreieck des Projektmanagements[33]

Der Flächeninhalt des Dreiecks wird durch die Leistungsfähigkeit des

[32] Hoffmann 2008, S. 9–10
[33] Hoffmann 2008, S. 12

Entwicklungsbetriebs bestimmt. Diese ist wiederum abhängig von der Organisation der Firma und von der Leistungsfähigkeit der Mitarbeiter. Eine geringe Ausdehnung des Dreiecks auf Kosten der Mitarbeiter mag in Ausnahmefällen möglich sein, aber bereits eine kurze Phase mit leicht erhöhter Leistungserbringung zieht unweigerlich eine Phase der Erholung nach sich, in der die Entwickler deutlich vermindert leistungsfähig sind. Anzustreben ist daher eine konstant hohe Leistung ohne Überforderung; der Flächeninhalt des Dreiecks bleibt konstant. Das Modell verdeutlicht, dass Qualität Zeit und Geld kostet. Eine Optimierung von Kosten *und* Entwicklungsdauer geht unweigerlich zu Lasten der Qualität.

Da in Softwareentwicklungsverträgen üblicherweise zusätzlich zur Funktionalität und den äußeren Qualitätsmerkmalen die Kosten und die Entwicklungsdauer der Software vereinbart werden, bleibt dem Hersteller der Software in diesem Fall nur eine einzige Variable: die innere Qualität. Um einen zu knappen Zeitplan und ein zu knappes Budget einzuhalten, werden also im Zweifel Codetransparenz, Testbarkeit, Wartbarkeit und Portierbarkeit eingeschränkt. Ein solches Produkt kann unter Umständen pünktlich und zur vollen Zufriedenheit des Kunden sein. Vernachlässigte innere Qualität ist jedoch der Hauptgrund für die so genannte Softwarealterung. Es ist daher auch möglich, dass sich durch mangelnde Codetransparenz und Testbarkeit die Entwicklungsgeschwindigkeit im weiteren Verlauf zunehmend verlangsamt, bis das Projekt schließlich komplett scheitert.

Der mangelnden Verlässlichkeit der Softwareentwicklung kann man mit der Qualität des Entwicklungsprozesses entgegenwirken. Verbindliche Standards für Design, Code und Tests schützen die Entwickler vor alterndem Code. Und um gar nicht erst in Versuchung zu kommen, an der Qualität zu sparen, sind beispielsweise genauere Schätzungen, bessere Planung, Transparenz gegenüber dem Kunden sowie flexible Verträge mit der Option von Teillieferungen und Nachverhandlungen hilfreich. Bei der Definition und Verbesserung von Entwicklungsprozessen setzen Methoden wie CMMI[34] oder SPICE[35] an. Sie sollen bei Herstellern und Kunden mit Hilfe von Zertifizierungen das Vertrauen schaffen, dass die getroffenen Zusagen eingehalten werden können.

Auch agile Methoden sollen Vertrauen zwischen Kunde und Hersteller schaffen: „Bei Scrum verpflichtet sich ein Team gemeinsam mit den Anforderern, den Kunden, dem Management, den Endanwendern und den Zulieferern darauf, ein bestimmtes Produkt innerhalb einer vorgegebenen Zeitspanne, die sehr überschaubar ist, zu liefern. Am Ende dieser Zeitspanne können alle Beteiligten erkennen, ob dieses Ziel erreicht worden ist.“[36] Da die Iterationen bei agilen Methoden sehr kurz sind, ist der durch den Kunden zu leistende Vertrauensvorschuss gering.

Agile Methoden stellen zudem hohe Anforderungen an die innere Qualität, denn wegen der kurzen Iterationen und des sukzessiven Designs sind Codetransparenz und Testbarkeit für das Funktionieren der Methode selbst deutlich wichtiger als bei plangesteuerten Methoden. Auch wenn ein Projekt in Verzug geraten sollte, ist die Verringerung der internen Qualität nicht zulässig. Im Hinblick auf das magische Dreieck bleiben mit Kosten, Zeit und externer Qualität immer noch drei mögliche Variablen, um Einfluss auf den Projektverlauf nehmen zu können. Agile Methoden verlangen Transparenz, enge Zusammenarbeit mit dem Kunden und flexible Verträge, um in solchen Situationen eine für alle Seiten akzeptable Lösung zu erreichen.

[34] Kneuper 2006
[35] Hörmann 2006
[36] Gloger 2008, S. 69

Bei der plangesteuerten Softwareentwicklung entstehen als Nebenprodukt zur Software viele Artefakte, die als Nachweis für getroffene Qualitätsmaßnahmen dienen können. Schneider weist darauf hin, dass solche Nachweise üblicherweise bei XP fehlen: „Die Qualität mag in XP hoch sein, das ist weniger das Problem; die Qualitätsmaßnahmen sind aber kaum dokumentiert und damit schlechter belegbar. Um das Projekt nicht zu bremsen, wurde auf Nachweise verzichtet.“[37] Dagegen muss man einwenden, dass bei XP sogar regelmäßig zwei Qualitätsnachweise erbracht werden: Die bestandenen automatisierten Tests und die lauffähige Software. Crystal Clear und Scrum bieten darüber hinaus auch die Freiheit, zusätzliche Qualitätsnachweise zu erstellen, wenn es als notwendig erachtet wird.

Agile Methoden eignen sich also offensichtlich zumindest grundsätzlich zur Sicherstellung von Qualität in der Softwareentwicklung. In Kapitel 3.2 wurde gezeigt, dass sich agile Methoden in einem hohen Maße anpassen und kombinieren lassen. Daher stellt sich die Frage, welche Faktoren die Qualität agiler Methoden beeinflussen und wie man mit ihrer Hilfe gezielt die Prozess- und Produktqualität erhöhen kann. Das folgende Unterkapitel wird diese Frage aufgreifen. Es wurde bereits erwähnt, dass Zertifizierungen eine Möglichkeit darstellen, um im eigenen Betrieb und bei Kunden Vertrauen zu schaffen. Da Zertifizierungen bei Unternehmen recht beliebt sind, werden in Kapitel 4.2 Zertifizierungsmöglichkeiten für agile Methoden vorgestellt.

4.1 Qualitätsfaktoren der agilen Softwareentwicklung

Unter „Qualitätsfaktoren“ werden hier Werte, Verhaltensweisen, Techniken und Praktiken verstanden, die bei der agilen Softwareentwicklung einen deutlichen Einfluss auf die Qualität des Produktes haben. Anhand dieser weitgefassten Definition wird bereits deutlich, dass das Qualitätsmanagement alle Ebenen der Softwareentwicklung durchdringt.

Da die untersuchten agilen Methoden recht unterschiedliche Ansätze verfolgen, wurden wichtige Qualitätsfaktoren ausgehend von den vier Schwerpunkten des Agilen Manifests aus allen drei Methoden zusammengetragen und in **Tabelle 4** zusammengefasst.

Schwerpunkt des agilen Manifests	Qualitätsfaktoren
Individuen und Interaktion	Kommunikation Erfahrung Einbindung des Kunden
funktionierende Software	Einbindung des Kunden Kommunikation Erfahrung Sukzessives Design und Refactoring Kontinuierliche Integration und Test Timeboxing
Zusammenarbeit mit dem Kunden	Einbindung des Kunden Kommunikation
sich Änderungen anpassen	Kommunikation Einbindung des Kunden Prozessverbesserung

Tab. 4: Qualitätsfaktoren von Crystal Clear, Scrum und Extreme Programming

Um Mehrfachnennungen bereinigt, ergeben sich aus obenstehender Tabelle sieben

[37] Schneider 2007, S. 186–187

Qualitätsfaktoren:

- Kommunikation
- Erfahrung
- Timeboxing
- Einbindung des Kunden
- Sukzessives Design und Refactoring
- Kontinuierliche Integration und Test
- Prozessverbesserung

Selbstverständlich fallen diese Qualitätsfaktoren nicht bei allen agilen Methoden gleichermaßen ins Gewicht. Wie bereits in Kapitel 3.2 dargestellt, handelt es sich bei Extreme Programming um eine sehr eng definierte Methode mit klar vorgeschriebenen Entwicklungspraktiken. Weil XP ganz allgemein keinen Spielraum für Anpassungen der Methode bietet, spielt die Prozessverbesserung hier auch keine nennenswerte Rolle. Scrum und Crystal Clear dagegen bieten größere Freiheiten bei der Anpassung. Erkenntnisse über Qualitätsfaktoren können daher sogar verwendet werden, um diese Methoden im Hinblick auf die Qualitätssicherung gezielt zu ergänzen oder zu modifizieren.

In den folgenden sieben Unterkapiteln wird betrachtet, wie sich diese Qualitätsfaktoren während der Entwicklung auf die Softwarequalität auswirken und welche Variablen bei jedem dieser Faktoren zur Einflussnahme auf die Qualität bestehen.

4.1.1 Kommunikation

Da agile Methoden wenig Wert auf Dokumentation legen, ist ein viel höherer Bedarf an persönlicher Kommunikation von Mensch zu Mensch nötig als bei plangesteuerten Methoden. Bei der Planung vor einer Iteration bespricht das Team die Anforderungen so lange mit dem Kunden, bis sich in den Köpfen des Teams ein Plan in Form von implizitem, interpersonellem Wissen herausgebildet hat.

Um diesem Plan bei der Entwicklung folgen zu können, ist ständige Kommunikation notwendig, denn er existiert ja nur im kollektiven Bewusstsein des Teams. Konkrete Entscheidungen und Designfragen werden zu einem großen Teil in die Entwicklungsphase verlagert. Es mag zwar je nach Projekt in verschiedenem Umfang Skizzen und Entwurfsdokumente geben, aber im Zweifelsfall gibt es keine Garantie dafür, dass die Antwort auf eine konkrete Frage aus diesen Dokumenten hervorgeht. Zur Klärung solcher Fragen ist oft eine Vielzahl von kurzen, unbürokratischen Abstimmungen notwendig. Die Büroräume und die Organisation des Teams sollten diese Arbeitsweise bestmöglich unterstützen und damit „osmotische Kommunikation“ nach Alistair Cockburn ermöglichen.

Aufwand und Qualität der Teamkommunikation werden in erster Linie durch die Anzahl der individuellen Kommunikationskanäle zwischen den Teammitgliedern bestimmt. **Abbildung 6** verdeutlicht, dass die Anzahl der Kanäle quadratisch wächst. Ein Team aus 10 Personen verfügt über 45 individuelle Kanäle, ein Team aus 15 Personen bereits über 105.

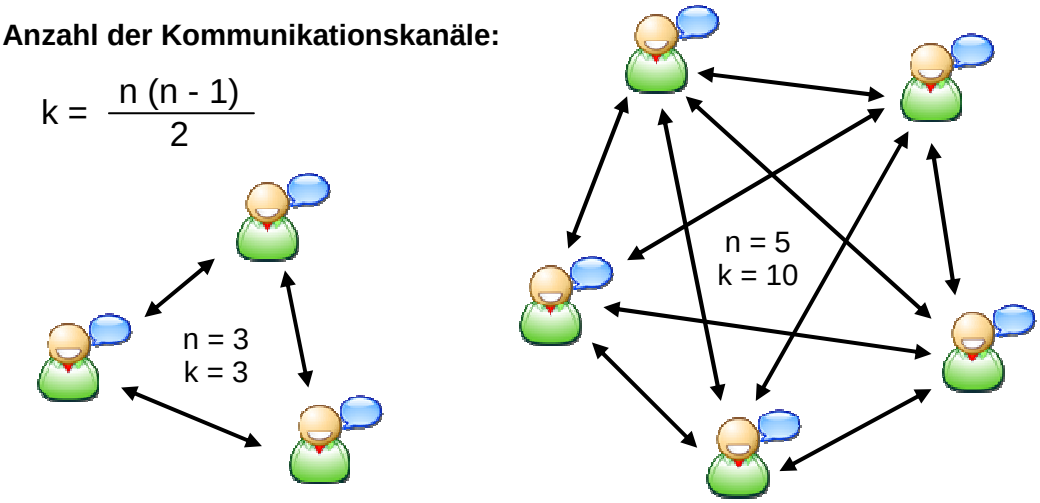

Abb. 6: Skalierung von Mensch-zu-Mensch-Kommunikation

Je größer das Team ist, umso größer wird auch der Aufwand zur Koordination. Während sich kleine Teams üblicherweise intuitiv koordinieren, werden bei großen Teams Hilfsmittel und Methoden notwendig, um die Anzahl der notwendigen individuellen Kommunikationsvorgänge zu verringern. Der Kommunikationsaufwand kann insbesondere dadurch gesenkt werden, dass die Einzelkommunikation soweit möglich durch Gruppenkommunikation ersetzt wird. Dieses Ziel verfolgen die täglichen Meetings aus Scrum oder XP, in denen reihum jedes Teammitglied seine aktuelle Arbeitssituation schildert. Durch diese effiziente Kanalbündelung genügt bereits eine der jeweiligen Teamgröße entsprechende Anzahl von Kommunikationsvorgängen, um ein gewisses Maß an Koordination zu erreichen.

Zusätzlich kann die Teamkoordination mit Mitteln wie gemeinsam geführten und für das gesamte Team stets sichtbaren Übersichtstafeln, Aufgabenlisten und Gedankenstützen erleichtert werden. Die persönliche Kommunikation von Mensch zu Mensch soll dadurch keinesfalls ersetzt, sondern auf einen praktikablen Umfang reduziert werden.

Bei der Kommunikation zwischen Menschen spielt nicht nur kühle fachliche Logik eine Rolle. Es gibt immer auch unterschwellige menschliche Faktoren, die in bestimmten Konstellationen die Teamkommunikation behindern können. Boris Gloger nennt sieben typische Konfliktfaktoren[38] in Teams:

- Machtkonflikte
- Finanzielle Ungerechtigkeiten
- Status-Konflikte
- Rassismus
- Sexismus
- Erotik
- Stress

In einem gleichberechtigten Team sollte es keine eindeutige Machtstruktur geben. Die Führung wechselt je nach Situation und Problemstellung zwischen den Mitgliedern. Machtkonflikte entstehen, wenn mehrere Teammitglieder die Führung in einer bestimmten Entscheidungssituation übernehmen wollen und den Machtanspruch der jeweils anderen nicht anerkennen. Machtkonflikte müssen sofort thematisiert und durch die gemeinsame Aufarbeitung von Argumenten beigelegt werden. Sollten Teammitglieder ihren Anspruch auf Macht auch gegen den Willen des restlichen

[38] vgl. Gloger 2008, S. 233–235

Teams nicht aufgeben, muss ernsthaft geprüft werden, ob sie für die Teamarbeit überhaupt geeignet sind.

Wenn Teammitglieder das Gefühl haben, von der Gruppe nicht anerkannt zu werden, kommt es zu Status-Konflikten. Rassismus und Sexismus fallen im Prinzip auch in diese Kategorie. Diese Konflikte lassen sich nur verhindern, indem eine Kultur der Anerkennung und Wertschätzung innerhalb der Gruppe etabliert und gelebt wird. Rationale und irrationale Probleme müssen jederzeit angesprochen werden können, doch der Status aller Mitglieder als wertgeschätzte Kollegen muss dabei unangetastet bleiben.

Der Status eines Mitarbeiters in der Firma wird auch durch das Gehalt verdeutlicht. Große Gehaltsunterschiede innerhalb eines Teams, in dem alle Mitglieder die gleiche Arbeit erledigen, können als ungerecht empfunden werden und Ursache für Konflikte sein. Es ist natürlich möglich, Teammitglieder unterschiedlich zu bezahlen, doch die Begründung dafür sollte transparent sein und allen Mitgliedern eine klare Perspektive zum Aufstieg eröffnen. So ist es denkbar, das Gehalt an bestimmte Ausbildungen und Zertifizierungen und an die Berufserfahrung zu knüpfen.

Erotik kann ein Team laut Gloger zwar beflügeln, führt aber besonders aufgrund des üblicherweise unausgewogenen Geschlechterverhältnisses in der Softwarebranche auch zu Konflikten. Da es sich bei Erotik um eine überwiegend unterbewusst wirkende Kraft und darüber hinaus um ein Tabuthema der Arbeitswelt handelt, können außer der Angleichung des Geschlechterverhältnisses kaum sinnvolle vorbeugende Maßnahmen ergriffen werden. Umso wichtiger ist es daher, tatsächliche Konflikte schnell zu erkennen, das Tabuthema in einem solchen Fall in einer angemessenen Form anzusprechen und den Konflikt unter Einbeziehung der Beteiligten individuell aufzulösen.

Auch der Faktor Stress präsentiert sich als zweischneidiges Schwert: Einerseits kann gesunder Stress (sogenannter Eu-Stress) ein Team zur Kommunikation und zu effizienter Arbeit anregen. In aussichtslos erscheinenden Situationen entsteht jedoch negativer Stress (Di-Stress), der den genau gegenteiligen Effekt hat und die Mitarbeiter zudem gesundheitlich belastet. In erster Linie gilt es daher, negativen Stress organisatorisch zu verhindern, indem faire Arbeitszeiten eingehalten werden und den Mitarbeitern eine ausgewogene Work-Life-Balance ermöglicht wird. Bei Fehlplanungen muss der Plan korrigiert werden, nicht die Arbeitszeit der Mitarbeiter.

Konflikte werden oft nicht als solche wahrgenommen, wenn man selbst Teil des Geschehens ist. Es empfiehlt sich daher, einen externen Beobachter einzusetzen, der Störungen in der Teamkommunikation erkennen und ansprechen kann. Beispielsweise sieht Scrum für diese Aufgabe den Scrum Master vor, der genau deshalb auch kein Mitglied des Teams ist.

4.1.2 Erfahrung

In Kapitel 3.3.4 wurde bereits darauf hingewiesen, dass in agilen Teams grundsätzlich eine größere kritische Masse an kompetenten Mitarbeitern benötigt wird als in plangesteuerten Teams. Nun soll untersucht werden, in welchen konkreten Feldern das Personal über Erfahrungen verfügen muss, um ein qualitativ hochwertiges Produkt erstellen zu können.

Wenn ein Team ausnahmslos an die plangesteuerte Softwareentwicklung gewöhnt ist, kann es ohne eine Einführung in den vorgesehenen agilen Prozess und Unterstützung bei der Durchführung nicht agil entwickeln. Das Team mag aus noch so guten

Programmierern bestehen und noch so viele ähnliche Projekte erfolgreich bewältigt haben, doch der Mangel an Erfahrung im agilen *Prozessfeld* führt dazu, dass die Mitarbeiter in gewohnte Arbeitsweisen zurück fallen und fehlende Vorgaben und Pläne als störenden Mangel statt als Anreiz zur Arbeit wahrnehmen.

Ein Team mit Erfahrung in agiler Entwicklung kann trotzdem in große Schwierigkeiten geraten, wenn bisher unbekannte Technologien genutzt werden sollen. Dabei kann es sich um Programmiersprachen, Entwicklungsumgebungen, Frameworks, Komponenten, Test- und Buildsysteme oder um Software zur Unterstützung des Arbeitsprozesses handeln. Fehlende Erfahrung in diesem *Technologiefeld* kann zu falschen Annahmen, Schätzungen, Planungen und zu falschem oder schlechtem Design führen. Wenn während des Projektes nicht ausreichend Erfahrungen gesammelt werden können, kann es sogar notwendig werden, das Personal zusätzlich zu schulen. In jedem Fall kann pro Iteration aber weniger geliefert werden als vorgesehen; möglicherweise kann zu Beginn sogar überhaupt keine benutzbare Software produziert werden. Mit dem Kunden getroffenen Abmachungen müssen dann korrigiert werden. Da das Vertrauen des Kunden massiv enttäuscht wird, kann unter diesem neuen Vorzeichen das ganze Projekt gefährdet sein.

Doch es ist noch ein drittes Erfahrungsfeld entscheidend. Man stelle sich einfach vor, dass ein Team, das bisher erfolgreich Lohnbuchhaltungssysteme entwickelt hat, plötzlich eine Echtzeitsteuerung für Industrieanlagen programmieren soll. Selbst wenn die Entwicklung mit der gleichen Technologie stattfindet, fehlt dem Team das notwendige Wissen, um zuverlässige Annahmen zu treffen und ein geeignetes Design zu erstellen. Die Erfahrung im *Geschäftsfeld* beschränkt sich nicht nur darauf, den Kunden und seine Anforderungen zu verstehen, sondern beinhaltet auch Erfahrungen über den Aufwand und über typische Probleme solcher Projekte sowie die Kenntnis von Best Practices beim Design.

Das Sammeln von Erfahrungen im Prozessfeld ist während eines laufenden Projektes möglich, setzt aber voraus, dass mindestens ein sehr erfahrener Mitarbeiter als Coach zur Verfügung steht. Diese Person kann wie bei Scrum und XP in Form des Scrum Masters und des Coaches außerhalb des Teams stehen oder wie bei Crystal Clear selbst dem Team angehören, muss aber in jedem Fall streng über die Einhaltung des Prozesses wachen.

Optimalerweise sollte das gesamte Team Erfahrung im Technologiefeld mitbringen. Wenn eine eingesetzte Technologie für das Personal leicht zu erlernen ist – etwa, weil eine sehr ähnliche Technologie bereits etabliert ist – kann ein einzelner Experte im Team die restlichen Mitglieder einweisen und aufkommende Fragen beantworten. Es ist allerdings darauf zu achten, dass jede einzelne eingesetzte Technologie von mindestens einem Teammitglied beherrscht wird, damit es wirklich zu jedem Problem und bei jeder Frage mindestens einen kompetenten Ansprechpartner im Team gibt. Wenn der Aufwand zum Erlernen der eingesetzten Technologie höher ist, muss das Personal vor Beginn des Projektes geschult werden. Das Entwicklungsteam sollte allerdings immer mindestens einen Mitarbeiter beinhalten, der bereits über praktische Entwicklungserfahrung mit der Technologie verfügt.

Wenn das Team keine Erfahrung im Geschäftsfeld hat, ist es üblich, vor dem Projektstart einen Experten mit umfangreicher praktischer Erfahrung in der Realisierung entsprechender Software zu konsultieren. Dieser Experte prüft die Kenntnisse des Teams und schätzt ein, wie groß das Wissensdefizit ist. Auf dieser Basis kann entschieden werden, ob eine Fortbildung notwendig ist, ob die Einbindung

von Experten in das Team ausreicht, oder ob das Team in der aktuellen Konstellation für das Projekt geeignet ist.

4.1.3 Timeboxing

Vor jeder Iteration schätzen agile Teams, wie viel Arbeit während dieser Zeit gemeinsam bewältigt werden kann. Es liegt in der Natur von Schätzungen, dass sie notorisch ungenau sind. Eine genaue Vorhersage kann auch unmöglich getroffen werden, da zu diesem Zeitpunkt das Design höchstens grob umrissen ist und noch keine Erfahrungen aus der Implementierung vorliegen. Je nachdem, als wie zutreffend sich die Annahmen des Teams während der Entwicklung herausstellen, kann die geplante Arbeit rechtzeitig fertig gestellt oder eben mehr Zeit benötigt werden.

Wenn einzelne Aufgaben nicht rechtzeitig fertig werden, ist die naheliegende Reaktion, den Liefertermin Stück für Stück so lange in die Zukunft zu verlegen, bis ein Termin gefunden ist, zu dem alles fertig gestellt werden kann. Durch die Verlegung von zuvor dem Kunden versprochenen Terminen wächst jedoch der Druck auf die Entwickler und die Einhaltung des Lieferdatums wird zunehmend wichtiger als die Codequalität.

Dieser Verzerrung der Qualität durch zunehmenden Zeitdruck wirkt die Zeitmanagement-Strategie des *Timeboxing* entgegen. Jede Iteration wird dabei als Timebox angesehen, was nichts anderes heißt, als dass die Länge festgelegt ist und das Enddatum nicht verändert werden kann. Alle Programmteile, die zu diesem Zeitpunkt nicht fertig sind oder den festgelegten Qualitätskriterien nicht entsprechen, gelten als nicht fertig gestellt. Das schafft Transparenz und ermöglicht es, den Ursachen der falschen Schätzungen auf den Grund zu gehen. Aufgrund der Erfahrungen während der Iteration können bessere Schätzungen vorgenommen oder das Design verändert werden. Vor allem aber erhält der Kunde die Möglichkeit, seine Prioritäten für die nächste Iteration neu zu setzen. Er kann entscheiden, ob die Wichtigkeit der gewünschten Funktionalität den Mehraufwand rechtfertigt, ob die Anforderungen verändert werden können, oder ob die Implementierung der Funktionalität zunächst zurückgestellt werden kann.

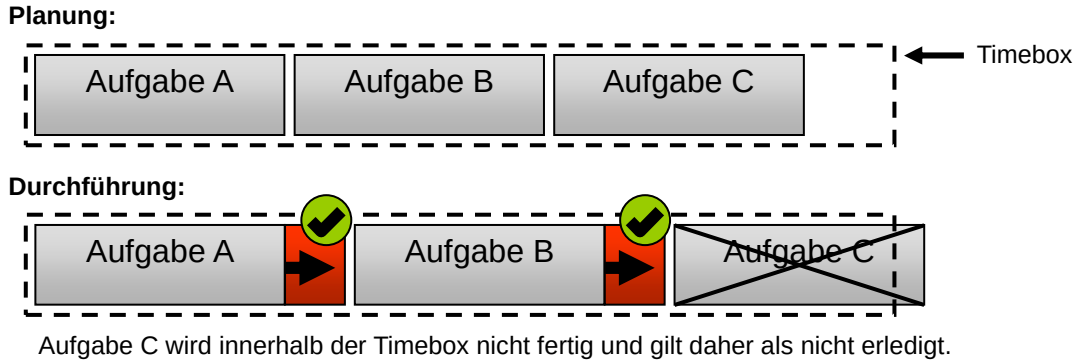

Abb. 7: Timeboxing

Timeboxing sorgt für eine Prioritätsverschiebung von der äußeren Qualität (Funktionalität) hin zur inneren Qualität (Codetransparenz, Testbarkeit). Um diese garantieren zu können, ist es notwendig, Qualitätsstandards für den Code zu etablieren und streng zu kontrollieren.

Wenn agile Teams nicht früh genug auf benutzbare Software hin arbeiten, besteht bei konsequenter Anwendung des Timeboxing die Gefahr, dass am Ende einer Iteration überhaupt kein benutzbares Programm vorliegt. Jeff Sutherland nennt in einem

Interview sogar eine alarmierende Zahl: „The biggest problem we have today is that 50% of the Scrum teams cannot get working software at the end of the sprint. It's a huge problem. […] The issue is how you actually get people implementing well. If 90% of the implementations are not done very well then that's the problem we need to address." [39]

Timeboxing kann also nicht verhindern, dass Teile der eigentlich geplanten Funktionalität nicht fertig werden. Die Technik sorgt aber dafür, dass in einem solchen Fall die Annahmen und Planungen kritisch überprüft werden können und die Situation zusammen mit dem Kunden neu bewertet wird. Außerdem sorgt Timeboxing für einen festen Rhythmus im Projekt, der die Iterationen untereinander vergleichbar macht und künftige Schätzungen erleichtert. Nebenbei wird die Zusammenarbeit mit dem Kunden dadurch erleichtert, dass fixe Termine für Meetings eine lange Zeit im Voraus festgelegt werden können.

4.1.4 Einbindung des Kunden

Bei der agilen Softwareentwicklung spielen Verhandlungen und Planungen zu Beginn eines Projektes nur eine untergeordnete Rolle. Daher haben der Grad und die Art und Weise der Einbindung des Kunden während der Entwicklung großen Einfluss auf die äußeren Qualitätsmerkmale der dabei entstehenden Software.

Betrachten wir zunächst einen fiktiven negativen Extremfall: Ein Kunde erteilt einen schriftlichen Auftrag. Bei Planung und Entwicklung der Software wird der Kunde nicht konsultiert. Über mehrere Iterationen wird eine Software erstellt, bis alle Anforderungen aus dem Auftrag umgesetzt sind. Nach der Lieferung der Software stellt der Kunde schließlich fest, dass die Funktionalität der Software dem vorgesehenen Einsatzzweck nicht gerecht wird, das Programm viel zu langsam reagiert und zu allem Überfluss von den Mitarbeitern nicht verstanden wird. Es überrascht nicht, wenn der Kunde die Qualität dieser Software als miserabel bezeichnet.

Eine enge Einbindung des Kunden in Planung und Entwicklung dagegen schafft Vertrauen, verhindert Missverständnisse und minimiert etwaige Verzögerungen. Der Kunde erhält so den größtmöglichen Einfluss auf die Planung und Umsetzung von Funktionalität. Wenn bei Schwierigkeiten das weitere Vorgehen sofort mit dem Kunden abgestimmt wird, kann dieser auch seine eigenen Planungen und Erwartungen korrigieren. Sobald benutzbare Software vorliegt, kann der Kunde Laufzeit, Zuverlässigkeit und Benutzbarkeit beurteilen und sein Feedback dazu geben. So werden mögliche Probleme und Missverständnisse früh erkannt und der Kurs kann rechtzeitig korrigiert werden.

Diese Form der Zusammenarbeit setzt allerdings persönliche Kommunikation von Angesicht zu Angesicht voraus, da Dokumente zu viel Spielraum für Interpretationen und Missverständnisse lassen. Nun ist „der Kunde" meist keine Einzelperson, sondern ein Unternehmen. Um die angestrebte enge Zusammenarbeit zu erreichen, müssen daher Vertreter des Kunden eingebunden werden.

Boehm und Turner haben bei einer Analyse von über 100 Projekten festgestellt, dass der Erfolg der Zusammenarbeit von fünf Eigenschaften[40] der Kundenvertreter abhängig ist:

[39] Moe et al. Juli 2009, S. 3
[40] vgl. Boehm, Turner 2008, S. 44–45

- **Bereitschaft zur Zusammenarbeit**: Verweigern oder behindern Kundenvertreter die Zusammenarbeit, kann dieses Defizit auch durch das Entwicklungsteam nicht ausgeglichen werden.
- **Repräsentativität**: Kundenvertreter müssen alle Interessen des Kunden vertreten. Da die Ansichten innerhalb des Unternehmens möglicherweise auseinander gehen, müssen Kundenvertreter einen dem Projekt förderlichen Mittelweg finden.
- **Entscheidungsbefugnis**: Fehlt Kundenvertretern die Befugnis, Entscheidungen bis zu einer gewissen Tragweite eigenständig zu treffen, besteht die Gefahr, dass sie das Projekt ausbremsen oder durch unbefugte Entscheidungen in die Irre leiten.
- **Verpflichtung**: Kundenvertreter müssen sich dem Projekt verpflichtet fühlen, damit sie auch die Entwicklerseite beim Kunden angemessen vertreten, ihre Aufgaben zuverlässig erledigen und anwesend sind, wenn das Entwicklungsteam sie braucht.
- **Sachkunde**: Sie müssen das Projekt, den Kunden und die Entwickler verstehen. Fehlende Sachkunde führt zu Verzögerungen, einem Produkt von schlechter Qualität oder zu beidem.

Diese Aufzählung verdeutlicht, dass die Erwartungen an Kundenvertreter sehr hoch sind. Mitarbeiter dieses Kalibers sind bei Kunden üblicherweise selten, ausgelastet und teuer. Kaum ein Kunde wird es sich daher leisten können, für ein einzelnes Projekt einen solchen Vertreter in Vollzeit zur Verfügung zu stellen.

Als Ausweg aus dieser Problematik bieten sich mehrere Wege an. Zum einen kann die zeitliche Belastung auf mehrere Schultern verteilt werden, zum anderen kann die Intensität der Zusammenarbeit von Vollzeit auf kürzere, aber regelmäßige Intervalle verringert werden. So wäre es zum Beispiel möglich, den Kundenvertreter im Regelfall nur über ein kurzes tägliches Meeting einzubeziehen und bei Bedarf zusätzliche Endanwender mit dem benötigten Spezialwissen ins Team zu holen. Idealerweise sollte der Kundenvertreter jedoch für wichtige Rückfragen telefonisch kurzfristig zur Verfügung stehen.

Eine weitere Möglichkeit besteht darin, dass der Kundenvertreter nicht vom Kunden, sondern vom Auftragnehmer gestellt wird. Dem Kunden wird dadurch die möglicherweise ungewohnte und aufwändige Entsendung von Mitarbeitern erspart. Die Erfüllung der oben genannten fünf kritischen Eigenschaften fällt dadurch allerdings schwerer. Während man davon ausgehen kann, dass sich ein solcher Mitarbeiter dem Projekt verpflichtet fühlt, wird die Bereitschaft zur Zusammenarbeit lediglich auf seine Kontakte beim Kunden verschoben. Die geforderte Repräsentativität mag der Mitarbeiter durch seine externe Sichtweise auf den Kunden erreichen können, aber genau diese Perspektive geht auch zu Lasten der Sachkunde. Zu klären ist schließlich das Problem der Entscheidungsbefugnis: Wenn ein solcher Kundenvertreter alle Entscheidungen selbst trifft, wird die Entwicklung maßgeblich von der Entwicklerseite gelenkt und nicht mehr vom Kunden. Ein Entscheider beim Kunden müsste jedoch selbst wieder den Anforderungen an einen Kundenvertreter genügen. Denkbar wäre daher zum Beispiel ein Modell mit zwei Kundenvertretern, die jeweils auf Kunden- und Entwicklerseite als Schnittstellen zu ihren Kontakten fungieren und eng zusammen arbeiten.

In jedem Fall sollte der Kontakt zum Kunden nicht auf einen Kundenvertreter beschränkt bleiben. Um sicherzugehen, dass das Produkt den tatsächlichen Anforde-

rungen genügt, ist es notwendig, lauffähige Software mit echten Endanwendern zu testen. Die Reaktionen der Anwender geben dann Aufschluss darüber, wie gut sie der Kundenvertreter verstanden und vertreten hat. Sollte es grundsätzliche oder anhaltende Probleme geben, müssen die Art und Weise und die personelle Besetzung der Kundenvertretung angepasst werden.

4.1.5 Sukzessives Design und Refactoring

Planung und Design komplett „Up-front" zu Beginn eines Projektes durchzuführen bringt einige signifikante Probleme mit sich:

- **Umfang und Komplexität des Plans**: Da der Plan zu umfangreich ist um im Kopf behalten zu werden, ist man auf eine genaue schriftliche Fixierung aller Punkte angewiesen. Das kostet viel Zeit und kann trotzdem missverständlich sein.
- **Verifikation des Plans**: Wenn Kunden- und Entwicklerseite bei der Einigung auf einen Plan unwissentlich aneinander vorbei arbeiten, wird die resultierende Software den Anforderungen des Kunden nicht genügen, obwohl sie dem Plan aus Sicht der Entwickler entspricht. Große Änderungen sind dann nicht mehr möglich.
- **Änderungen**: Während der laufenden Entwicklung können sich die Anforderungen ändern. Diese Änderungen in den Plan und in die Entwicklung einfließen zu lassen ist kompliziert. Manche Änderungswünsche sind schlichtweg nicht mehr umsetzbar.
- **Erfahrung**: Das Design muss komplett ohne praktische Erfahrungswerte aus der Entwicklung des Projektes erstellt werden. Spätere Erfahrungen machen eventuell wieder komplizierte Änderungen notwendig.

Auf der anderen Seite ist ein reines „Ad-hoc-Design", bei dem die komplette Planung erst unmittelbar im Moment der Umsetzung durch den Programmierer stattfindet, auch nicht erstrebenswert. Ein solches Vorgehen erschwert jede Form von Steuerung und Teamarbeit sehr. Man stelle sich noch vor, ein etwas unentschlossener Kundenvertreter hätte außerdem jederzeit vollen Einfluss auf alle Entscheidungen – das Projekt würde hin- und hersteuern, bald im Chaos versinken und möglicherweise niemals benutzbare Software hervorbringen.

Es ist also sinnvoll, mit sukzessivem Design einen Mittelweg zu wählen: Die Entwicklung wird dabei in kurzen Iterationen vorangetrieben, wobei stets zuvor ein planerischer Rahmen abgesteckt wird, der den Entwicklern die nötige Sicherheit und Orientierung bietet. Eine solche Vorgehensweise schafft drei Ebenen für das Design der Software:

- **Iterationsübergreifendes Design**: Kritische Programmteile oder sehr wichtige Schnittstellen können vor Beginn der ersten Iteration für das gesamte Projekt geplant werden. Diese Ebene bietet die Planungssicherheit des plangesteuerten Ansatzes, bringt aber auch seine Nachteile mit. Da in einem agilen Projekt das iterationsübergreifende Design auf ein Minimum reduziert wird, kann die Komplexität jedoch überschaubar sein. Verifikation und Änderungen können damit erleichtert werden.
- **Iterationsdesign**: In enger Zusammenarbeit mit dem Kunden werden auf dieser Ebene die Anforderungen in ein Design umgesetzt. Wegen der kurzen Iterationen ist der Umfang des Plans gering, weshalb er in Form von

implizitem Wissen in den Köpfen des Teams bestehen kann und nur mit wenigen Notizen und Skizzen umrissen werden muss.

- **Ad-hoc-Design**: Alle Design-Entscheidungen, die zusätzlich zur Vorbereitung einer konkreten Implementierung getroffen werden müssen, das restliche Team aber nicht oder nur in geringem Maße beeinträchtigen, können vom einzelnen Programmierer unmittelbar vor und während der Programmierung getroffen werden. Sind andere Teammitglieder von der Ausgestaltung des Designs betroffen, kann das Vorgehen sofort persönlich abgestimmt werden.

Dieser Ansatz vereint die Vorteile der langfristigen Planung mit denen überschaubarer Pläne, während gleichzeitig die Eigenverantwortung der Programmierer gestärkt wird.

Aus Sicht der Qualität bieten dieses Vorgehen zwei Stellschrauben: Die Länge der Iterationen und die Verteilung der Designaufgaben auf die drei Ebenen. Sind die Iterationen zu lang, wird die Planung unübersichtlich, sind sie zu kurz, kann in der knappen Zeit keine benutzbare Software erstellt werden. Bei einer falschen Verteilung der Designaufgaben mehren sich die negativen Symptome des Up-front-Designs, des Ad-hoc-Designs oder beider Ansätze.

Unter dem sukzessiven Design leidet jedoch die Designqualität immens. Da beim Iterationsdesign nur die unmittelbar bevorstehenden Aufgaben von Interesse sind, ist klar, dass das Design im Hinblick auf das Gesamtprojekt schnell zu kurz greift. Innerhalb einiger Iterationen ohne Anpassungen des bestehenden Codes würde damit ein Stückwerk entstehen, das weder effizient noch übersichtlich ist. Ein Projekt in einem solchen Zustand wäre von minderwertiger Codequalität und könnte zudem nicht mehr agil weiterentwickelt werden. Aus diesem Grund ist es von zentraler Bedeutung, in jeder Designrunde auch das bestehende Design an die neuen Anforderungen anzupassen und den vorhandenen Code wenn nötig einem Refactoring zu unterziehen. Damit stellt die Abwägung, wie viel Zeit in einer Iteration für die Anpassung von bestehendem Code und wie viel Zeit für das Erstellen von neuem Code aufgewendet wird, die letzte Qualitätsvariable in diesem Bereich dar.

4.1.6 Kontinuierliche Integration und Test

Auch bei iterativer Entwicklung kann man verschiedene Integrationsstrategien verfolgen. Es geht dabei um die Frage, wann und wie Module oder Programmteile verschiedener Programmierer zu einem Gesamtprojekt vereinigt werden. Dirk W. Hoffmann unterscheidet drei grundsätzliche Strategien: Die Big-Bang-Integration, die strukturorientierte Integration und die funktionsorientierte Integration.[41]

Beim Big-Bang werden erst die vollständig fertig entwickelten Module und Programmteile auf einen Schlag integriert. Dadurch werden jedoch Fehler im Programm und Probleme in der Architektur erst sehr spät sichtbar und Änderungen stark erschwert. Außerdem lassen sich Fehler, die sich aus der Wechselwirkung vieler Programmteile ergeben, nur sehr schwer zurückverfolgen und beseitigen. Bei dieser Strategie besteht ein extrem hohes Risiko, dass am Ende einer Iteration keine benutzbare Software vorliegt, sie ist daher ungeeignet.

Struktur- und funktionsorientiertes Vorgehen sehen dagegen kontinuierliche Integration vor und machen damit auch kontinuierliches Testen möglich. Die innere Qualität der Software wird so ständig geprüft; die Entwicklung schreitet erst fort,

[41] vgl. Hoffmann 2008, S. 163–166

wenn der Code den Qualitätsanforderungen genügt.

Betrachtet man Software als eine Ansammlung teils voneinander abhängiger Module mit Funktionalität, die gemeinsam eine Gesamtfunktionalität bereitstellen, so sieht die strukturorientierte Integration vor, dass die Module nacheinander vollständig entwickelt und integriert werden. Bei der funktionsorientierten Vorgehensweise wird Code dagegen von einer angestrebten Gesamtfunktionalität ausgehend in alle betroffenen Module integriert. Wird eine Software komplett strukturorientiert entwickelt und integriert, besteht die Gefahr, dass wichtige Module zum Ende der Iteration nicht fertig werden und man nur wenig oder gar keine benutzbare Funktionalität vorweisen kann. Da benutzbare Software ein zentrales Ziel der agilen Softwareentwicklung ist, bietet sich also hauptsächlich die funktionsorientierte Integration an. Wenn dabei nicht alle Programmieraufgaben fertig gestellt werden können, fehlt am Ende nur die entsprechende Teilmenge der geplanten Funktionalität. Der Rest ist bei voller innerer Qualität vorhanden.

Allerdings stellt die funktionsorientierte Vorgehensweise hohe Anforderungen an die Teamdisziplin und an die technische Ausstattung:

- **Versionsverwaltung**: Da das Team gleichzeitig an vielen Teilen des Codes arbeitet, ist ein leistungsfähiges Versionsverwaltungssystem notwendig.
- **Refactoring-Funktionalität**: Um schnell und problemlos notwendige größere Änderungen an bestehendem Code vornehmen zu können, sollte die Entwicklungsumgebung über möglichst mächtige Refactoring-Funktionen verfügen.
- **Automatisierte Testumgebung**: Automatisierte Tests werden benötigt, damit durch gleichzeitige Codebearbeitung oder Refactoring direkt und indirekt entstandene Fehler sofort bei der Integration entdeckt und behoben werden können.

Innerhalb eines Projektes können struktur- und funktionsorientierte Entwicklung und Integration mit angemessener Planung auch vermischt werden. Die Anforderungen an die technische Ausstattung lassen sich zumindest etwas reduzieren, wenn zentrale Module zuerst implementiert werden und anschließend funktionsorientiert weiterer Code hinzugefügt und bestehender Code angepasst wird.

4.1.7 Prozessverbesserung

Die bisherigen Ausführungen machen deutlich, dass bei der agilen Softwareentwicklung allein zur Beeinflussung der Qualität zahlreiche Stellschrauben vorhanden sind. Wenn die Prioritäten, Regeln und Methoden eines Prozesses nicht zum Projekt passen, leidet die Qualität des Ergebnisses darunter. „Den Prozess“, der für alle Projekte uneingeschränkt geeignet ist, gibt es nicht. Es ist vielmehr notwendig, den Entwicklungsprozess für jedes Projekt an die konkreten Umstände und Anforderungen anzupassen.

Für ein Team gibt es kein deprimierenderes Erlebnis, als mit einem Projekt sehenden Auges zu scheitern. Wenn die Vorschriften des Prozesses nicht im Einklang mit einer effizienten, zielorientierten Arbeit stehen, unnötige Arbeit verursachen oder den Anforderungen des Projektes widersprechen, sinkt die Motivation der Teammitglieder und sie verlieren das Gefühl der Verpflichtung gegenüber dem Projekt. Agile Softwareentwicklung ist unter diesen Umständen nicht mehr möglich. Genauso kann es natürlich vorkommen, dass ein Projekt „blind“ gegen die Wand gefahren wird. Ein

solches Projekt scheitert deshalb, weil die Mitarbeiter unwissentlich einem ungeeigneten Prozess nachgehen.

Wie ein passender Prozess genau ausgestaltet sein muss, lässt sich vor einem Projektstart allenfalls schätzen. Es ist daher notwendig, auch während der laufenden Entwicklung die eigenen Regeln und Vorgehensweisen ständig im Blick zu haben, kritisch zu prüfen und nötigenfalls anzupassen. Eine solche kontinuierliche Prozessverbesserung wird zum Beispiel durch den Deming-Zyklus beschrieben, der aus den vier sich stets wiederholenden Phasen *Plan*, *Do*, *Check* und *Act* besteht:[42]

- **Plan**: In dieser Phase werden die bisherige Vorgehensweise analysiert und Probleme herausgearbeitet. Daraufhin werden konkrete Verbesserungsvorschläge gesammelt, untersucht und ausgewählt.
- **Do**: Anschließend werden die in der Planungsphase ausgewählten Maßnahmen in der Praxis eingesetzt und beobachtet, welche Wirkung sie tatsächlich zeigen.
- **Check**: Nach der Praxisphase werden die Erfahrungen gesammelt, Auswirkungen der Verbesserungen analysiert und überprüft, ob die gewünschten Ziele erreicht werden konnten.
- **Act**: Wenn die Änderungen tatsächlich Verbesserungen hervorgebracht haben, wird die Vorgehensweise beibehalten und der Prozess angepasst. Ansonsten wird die Idee verworfen oder in der nächsten Planungsphase abgeändert.

Agile Methoden eignen sich sehr gut für eine solche kontinuierliche Prozessverbesserung, da meist in sehr kurzen Iterationen gearbeitet wird und daher sehr schnell Erfahrungen aus einem Prozess gewonnen werden können. Diese Erfahrungen werden sofort sichtbar, egal ob es sich um gute oder schlechte handelt.

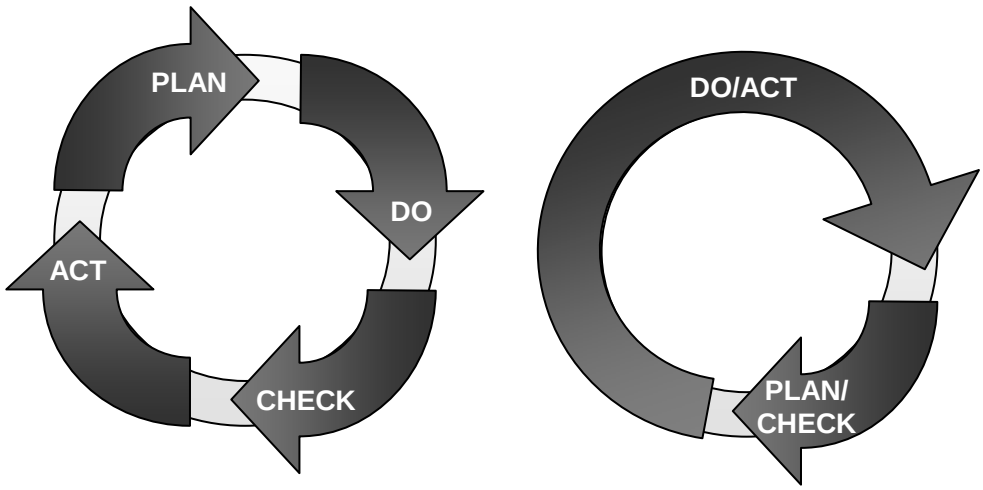

Abb. 8: Deming-Zyklus zur kontinuierlichen Prozessverbesserung und Abwandlung durch Verschränkung

Um den Deming-Zyklus dem Rhythmus agiler Methoden anzupassen, kann man ihn in sich verschränken und auf zwei kombinierte Phasen reduzieren: Die *Do/Act-Phase* entspricht dann einer Iteration. In dieser Phase wird der aktuelle Prozess umgesetzt, zusätzlich werden zuvor ausgewählte Änderungen ausprobiert. An jede Iteration schließt sich dann eine sehr kurze *Plan/Check-Phase* an, in der sowohl die zuvor getesteten Änderungen, als auch das sonstige Vorgehen evaluiert werden und entschieden wird, welche Verbesserungen beibehalten werden und welche neuen

[42] vgl. Wallmüller 2001, S. 110–111

Ideen in der nächsten Iteration ausprobiert werden sollen. Das Führen einer Beibehalten/Ausprobieren-Liste ist auch genau die Vorgehensweise, die Alistair Cockburn für den sogenannten Reflexions-Workshop vorschlägt.[43]

Die Prozessverbesserung ist also ein empirischer Vorgang; durch Beobachtung und schrittweise Anpassung wird die Qualität in kleinen Schritten erhöht. Die Qualität der Vorschläge und Verbesserungen ist allerdings in hohem Maße davon abhängig, wie gut und genau die Erfahrungen und Messungen sind, auf deren Basis die Verbesserungen geplant werden. Insbesondere bieten sich die Dokumentation von Schätzungen und Hindernissen und die Messung der Entwicklungsgeschwindigkeit an. Wenn zu wenig Wert auf Messungen und Dokumentation der Vorgehensweise gelegt wird, fehlen möglicherweise belastbare Grundlagen für den Entwurf konkreter Verbesserungsvorschläge. Auf der anderen Seite sollte nur so viel dokumentiert werden, wie unbedingt notwendig, damit die Entwicklungsgeschwindigkeit nicht zu sehr unter den Maßnahmen der Prozessverbesserung leidet.

4.2 Zertifizierung

Allein die Vielzahl der Variablen mit Einfluss auf die Qualität macht deutlich, dass sich Fehler und Versäumnisse an sehr vielen verschiedenen Stellen negativ auf die Softwarequalität auswirken können. Man kann also viel falsch machen. Firmen, die agile Methoden für kommerzielle Projekte einsetzen möchten, müssen daher volles Vertrauen in ihre Prozesse und in ihr Personal haben. Auch Kunden wählen bei der Auftragsvergabe oft nicht nur günstige, sondern auch vertrauenswürdige Angebote.

Neben Referenzprojekten spielen insbesondere Zertifizierungen eine wichtige Rolle bei der Vertrauensbildung innerhalb von Unternehmen und gegenüber Kunden. In den folgenden zwei Kapiteln sollen daher kurz Möglichkeiten zur Zertifizierung von Personal und agilen Prozessen dargestellt werden.

4.2.1 Zertifizierung des Personals

Bei der Zertifizierung von Personal werden Kenntnisse und Fähigkeiten in einem bestimmten Feld durch eine unabhängige Stelle geprüft und durch diese bestätigt. Die Zertifizierung ist umso wertvoller, je vertrauenswürdiger die zertifizierende Stelle ist. In Kapitel 4.1.2 wurden das Prozessfeld, das Technologiefeld und das Geschäftsfeld als die drei wichtigsten Erfahrungsfelder für das Personal agiler Entwicklungsteams vorgestellt. Da für einen reibungslosen Projektablauf mindestens ein ausgewiesener Experte für das Prozessfeld, das Geschäftsfeld und für jede eingesetzte Technologie im Team sein muss, ist die Zertifizierung des Personals in diesen Feldern natürlich besonders interessant. Daher soll nun untersucht werden, welche Zertifizierungsmöglichkeiten dort bestehen.

Im Prozessfeld zeichnet sich vor allem Scrum durch ein organisiertes Zertifizierungsprogramm aus. Die unter Beteiligung von Ken Schwaber gegründete *Scrum Alliance* bietet Zertifizierungen zum *Certified Scrum Master (CSM)*, zum *Certified Scrum Product Owner (CSPO)* und zum *Certified Scrum Practitioner (CSP)* an. Die Trainer der Scrum Alliance haben diese Zertifizierungen selbst durchlaufen und zusätzlich die Qualifikation zum Trainer erworben. Für XP werden von unterschiedlichen Anbietern Schulungen und Kurse angeboten. Diese Angebote mögen das notwendige Prozesswissen vermitteln, es fehlt jedoch ein Mechanismus zur Zertifizierung von Anbietern

[43] vgl. Cockburn, Giesecke 2005, S. 98

solcher Kurse. Die Auswahl geeigneter Anbieter fällt daher schwerer. Für Crystal Clear werden bisher augenscheinlich überhaupt keine Schulungen oder Zertifizierungen angeboten.

Im Technologiefeld bieten die Hersteller von Programmiersprachen, Entwicklungsumgebungen, Frameworks, Off-The-Shelf-Komponenten und prozessunterstützender Software überwiegend Zertifizierungsprogramme an. Je nach eingesetzter Technologie können Unternehmen ihren Entwicklern beispielsweise eine Zertifizierung als *Microsoft Certified Technology Spezialist (MCTS)*, als *Sun Certified Java Developer (SCJD)*, oder auch als *Zend Certified PHP Engineer (ZCE)* ermöglichen. Borland bietet etwa Zertifizierungen für seine Application Lifecycle Management Software an, und über Oracle können zumindest Kurse zu den verschiedenen Oracle-Technologien besucht werden.

Auch für das Geschäftsfeld gibt es zum Teil Zertifizierungen durch Technologieanbieter. Ein *Microsoft Certified Professional Developer (MCPD)* kann sich zum Beispiel als *Enterprise Application Developer* oder als *Web Developer* zertifizieren lassen. Sun bietet unter anderem eine Zertifizierung zum *Sun Certified Mobile Application Developer (SMAD)* an. Besonders viele individuelle Zertifizierungen für verschiedenste Geschäftsfelder bietet SAP. Das in solchen Programmen zertifizierte Wissen ist allerdings in hohem Maße von der jeweiligen Technologie abhängig. Unabhängige Erfahrung in einem Geschäftsfeld lässt sich wohl nur durch lange Berufserfahrung mit verschiedenen Technologien erlangen.

Beim Umstieg auf agile Softwareentwicklung empfiehlt es sich also insbesondere, das beteiligte Personal für die gewählte agile Methode zu schulen und gegebenenfalls zertifizieren zu lassen. Auch wenn das Team bereits Erfahrung im Technologiefeld hat, sollten passende Zertifizierungen angestrebt werden, um zu jeder eingesetzten Technologie auf mindestens einen zertifizierten Experten zurückgreifen zu können. Unternehmen sollten zunächst Geschäftsfeldern treu bleiben, in denen die Mitarbeiter bereits Erfahrung haben, da Zertifizierungen meist nur in Verbindungen mit bestimmten Technologien angeboten werden.

4.2.2 Zertifizierung von Prozessen

Zertifizierungen von agilen Prozessen sind derzeit nicht verbreitet. Für Crystal Clear, Scrum und Extreme Programming werden keine direkten Zertifizierungen angeboten. Daher stellt sich die Frage, inwiefern verbreitete Reifegradmodelle wie CMM/CMMI (entwickelt vom *Software Engineering Institute* der Carnegie Mellon University) oder SPICE (internationaler Standard) zur Ausgestaltung und Zertifizierung agiler Prozesse geeignet sind.

Hoffmann betont, dass Reifegradmodelle in erster Linie der Prozessverbesserung dienen: „Reifegradmodelle verfolgen das Ziel, die Arbeitsabläufe eines Unternehmens zu analysieren und zu optimieren. Hierzu werden die Prozesse einer Organisation anhand einer definierten Systematik auf Projekt- und Unternehmensebene bewertet und die Ergebnisse anschließend in Maßnahmen zur Prozessverbesserung umgesetzt. Neben der Schaffung produktiver Entwicklungsprozesse zielen Reifegradmodelle vor allem auf die Optimierung der Organisationsstrukturen ab. Hierzu zählen unter anderem die eindeutige Verteilung von Rollen und Verantwortlichkeiten, die Etablierung transparenter Kommunikationswege und die Definition klar geregelter Eskalationsmechanismen.“[44] In erster Linie sollen die Modelle Unternehmen also

[44] Hoffmann 2008, S. 492

dabei unterstützen, sukzessive einen immer höheren Reifegrad zu erreichen. CMM und CMMI sehen fünf Stufen vor, SPICE kennt sechs Stufen. Nun besteht die Möglichkeit, sich diesen Reifegrad zertifizieren zu lassen. Dabei wird im Rahmen eines sogenannten „Appraisal" oder „Assessment" geprüft, ob alle geforderten Eigenschaften der jeweiligen Stufe erfüllt werden.

Da Reifegradmodelle die Ausgestaltung von Prozessen nicht konkret vorschreiben, sondern für jeden Reifegrad bestimmte Eigenschaften fordern, ist die Anwendung auf agile Prozesse zumindest grundsätzlich möglich. Theoretische Untersuchungen und praktische Erfahrungen zur Kompatibilität von Reifegradmodellen und agilen Methoden finden sich vor allem im Umfeld von Scrum. So untersuchte Ken Schwaber bereits 2002 gemeinsam mit Mark Paulk vom Software Engineering Institute, ob Scrum die *Key Process Areas (KPA)* der CMM-Stufen 2 und 3 erfüllt. **Tabelle 5** zeigt das Ergebnis dieser Untersuchung, wobei ein Haken „größtenteils konform" und zwei Haken „vollständig konform" bedeuten.[45]

Stufe	KPA (Key Practice Area) [sic!]	Bewertung
2	Anforderungsverwaltung	✓✓
2	Software-Projektplanung	✓✓
2	Softwareprojekt-Verfolgung und -Aufsicht	✓✓
2	Software-Zuliefererverwaltung	
2	Software-Qualitätssicherung	✓✓
2	Software-Konfigurationsverwaltung	✓
3	Unternehmensprozess-Fokus	✓
3	Unternehmensprozess-Definition	✓
3	Schulungsprogramm	✓
3	Integrierte Softwareverwaltung	✓
3	Software-Produktentwicklung	✓✓
3	Gruppenübergreifende Kommunikation	✓✓
3	Durchsicht durch Programmierkollegen	✓✓

Tab. 5: Scrum und CMM [46]

Schwaber betont, dass Scrum sämtliche KPAs der Stufe 2 (die Software-Zuliefererverwaltung lässt er offenbar außen vor) und seit Beginn des Zertifizierungsprogramms der Scrum Alliance im Jahr 2003 auch alle KPAs der Stufe 3 erfüllt.

Im Paper „Scrum and CMMI Level 5: The Magic Potion for Code Warriors" beschreiben die Autoren das Fallbeispiel der Firma Systematic, die nach dem Appraisal zur CMMI-Stufe 5 Scrum eingeführt hat. Tatsächlich war die Anpassung der Prozesse möglich und sorgte insbesondere bei großen Projekten für eine deutliche Produktivitätssteigerung. Im Fazit des Papers heißt es: „Our recommendation to the Agile community is to use the CMMI generic practices from CMMI Level 3 to amplify the benefits from Agile methods. Our recommendation to the CMMI community is that Agile methods can fit into your CMMI framework and will provide exciting improvements to your organization."[47]

Als Fazit lässt sich also festhalten, dass Reifegradmodelle im Prinzip auf agile Prozesse angewendet werden können. Die oben genannten Untersuchungen legen nahe, dass zumindest bei der Anwendung von Scrum ein Appraisal zur CMMI-Stufe 3 vergleichsweise leicht möglich sein sollte.

[45] vgl. Schwaber 2007, S. 155
[46] Schwaber 2007, S. 155
[47] Sutherland et al. 2007, S. 6

5 Entwicklung eines qualitätsorientierten agilen Prozesses

Auf Basis der Untersuchung der Qualitätsfaktoren und -variablen in Kapitel 4 soll nun ein konkreter agiler Prozess vorgeschlagen werden, dessen Schwerpunkt auf der Sicherstellung der Qualität und auf der Dokumentation dieser Qualitätsmaßnahmen liegt. Der Prozess soll möglichst kompatibel zum bestehenden Entwicklungsprozess der Firma GIGATRONIK sein, damit bei Bedarf auch bereits bestehende Maßnahmen und Checklisten der Qualitätssicherung in den agilen Prozess eingebracht werden können.

Aus diesem Grund wird im nächsten Unterkapitel zunächst kurz der Entwicklungsprozess der Firma mit seinen Maßnahmen zur Qualitätssicherstellung vorgestellt. Im Anschluss daran wird die Ausgestaltung des agilen Prozesses erläutert.

5.1 GIGATRONIK-Entwicklungsprozess

Die Firma GIGATRONIK betreibt ein nach ISO 9001:2000 zertifiziertes Qualitätsmanagement, das die interne Organisation und alle Aktivitäten der Firma betrifft. In Bezug auf den Software-Entwicklungsprozess besagt das Qualitätsmanagement-Handbuch nur, dass bei der Softwareentwicklung nach einem definierten Prozess, dem „GT-Entwicklungsprozess (GT-EP)“ vorgegangen wird, und dass dieser Prozess „kontinuierlicher Überprüfung und Weiterentwicklung im Rahmen eines ständigen Verbesserungsprozesses“[48] unterliegt.

Bei der Betrachtung des Entwicklungsprozesses fällt allerdings auf, dass diese kontinuierliche Prozessverbesserung nicht selbst Teil des Prozesses ist. GIGATRONIK hat sich bei der Ausgestaltung des Prozesses zwar am Reifegradmodell CMMI orientiert, beschränkt sich jedoch auf die Umsetzung der Anforderungen des Reifegrades 2. Im Klartext heißt das, dass Projekte auf die dokumentierte Art und Weise bearbeitet und geführt werden. Erst CMMI-Level 3 fordert eine kontinuierliche Prozessverbesserung.

Der Entwicklungsprozess ist in fünf Hauptphasen (*Envisioning, Planning, Developing, Stabilizing* und *Deploying & Maintenance*) gegliedert, die dem Microsoft Solutions Framework (MSF) entnommen sind. Außer dieser Aufteilung und Namensgebung finden sich im Prozess allerdings keine weiteren Anleihen aus dem MSF wieder. Die Hauptphasen sind wie in **Abbildung 9** ersichtlich in acht operative Phasen unterteilt: *Projektbeginn, Angebot, Beauftragung, Design, Realisierung, Stabilisierung, Anlauf* und *Projektende*. Diese Phasen werden im Rahmen eines Projektes linear durchlaufen.

Abb. 9: Phasen des GIGATRONIK-Entwicklungsprozesses[49]

Durch den linearen Ablauf und die deutliche Aufteilung des Prozesses in Planung und Entwicklung wird deutlich, dass der GIGATRONIK-Entwicklungsprozess ein plan-

[48] GIGATRONIK GmbH September 2008, S. 22
[49] GIGATRONIK GmbH Juni 2007, S. 7

gesteuerter Prozess ist. Beim Vorgehensmodell handelt es sich im Prinzip um nichts anderes als das klassische Wasserfallmodell, das hier zum Vergleich in **Abbildung 10** dargestellt ist. Auch wenn die Phasen im GIGATRONIK-Entwicklungsprozess anders benannt sind, bleibt das Prinzip der Abkapselung und sequentiellen Abarbeitung der Phasen erhalten.

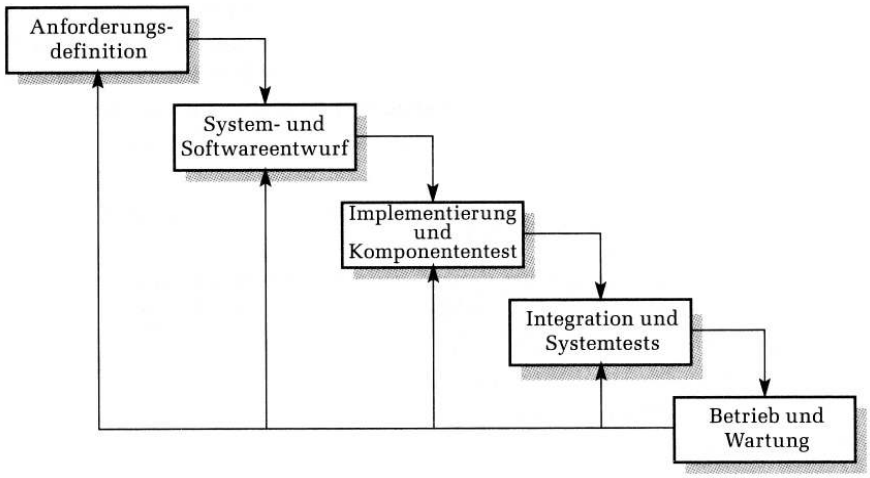

Abb. 10: Wasserfall-Modell[50]

Zur Einsteuerung von Änderungen während der laufenden Entwicklung sieht der GIGATRONIK-Entwicklungsprozess die parallele Abarbeitung eines „Change Request"-Prozesses vor. In diesem Prozess wird zunächst der mögliche Einfluss auf die laufende Entwicklung geprüft. Wenn der Change Request genehmigt wird, folgt die Änderung des Plans und der Software wieder Schritt für Schritt den Phasen des Prozesses.

Zur Qualitätssicherung sieht der GIGATRONIK-Prozess nach jeder operativen Phase (außer der Phase „Anlauf") ein sogenanntes *Quality Gate* vor. An jedem Gate wird zunächst überprüft, ob der Entwicklungsprozess gewissenhaft befolgt wurde. Anschließend wird die Qualität des Ergebnisses der vorangegangenen Phase mit Hilfe von Checklisten begutachtet. Ein Quality Gate darf nur durchschritten werden, wenn die Qualitätskriterien erfüllt werden. Werden die Kriterien nicht erfüllt, muss nachgebessert werden, bevor die nächste Phase der Entwicklung beginnen kann. Während der Phasen Design, Realisierung und Stabilisierung sollen periodisch Qualitätskontrollen durchgeführt werden, die jeweils mit dem Quality Gate „Kontrolle" enden.

Quer durch alle Prozessphasen definiert der GIGATRONIK-Entwicklungsprozess konkrete Aufgaben für die folgenden Prozessgebiete des CMMI-Level 2: *Konfigurationsmanagement, Messung und Analyse, Projektverfolgung und -steuerung, Projektplanung, Qualitätssicherung* sowie *Anforderungsmanagement*. Für das Prozessgebiet Qualitätssicherung werden vier konkrete Aufgaben definiert:

- Soll-Qualität / Style Guide definieren (Phase: Angebot)
- Qualitätsplan erstellen (Phase: Design)
- Testaktivitäten überprüfen (Phase: Stabilisierung)
- Endkontrolle: Qualität prüfen (Phase: Stabilisierung)

Insgesamt stellen sich die Quality Gates mit den ihnen zugeordneten Checklisten als Hauptinstrument der Qualitätssicherung im GIGATRONIK-Entwicklungsprozess dar.

[50] Sommerville 2007, S. 97

5.2 Agiler Prozess

Ein qualitätsorientierter agiler Softwareentwicklungsprozess muss so ausgestaltet sein, dass während eines Projektes die Aufmerksamkeit zu jeder Zeit auf die gerade relevanten Qualitätsfaktoren gelenkt wird. Die Idee ist, eine bestehende agile Methode so zu erweitern, dass durch die kontinuierliche Beobachtung dieser Faktoren mögliche Qualitätsprobleme frühzeitig erkannt werden und Handlungsbedarf deutlich gemacht wird. Die Erweiterung wird zudem so gestaltet, dass der resultierende Prozess kompatibel zu den Qualitätssicherungsmechanismen des GIGATRONIK-Entwicklungsprozesses ist. Bestehende Checklisten können dann – wenn gewünscht – auch im agilen Prozess eingesetzt werden.

Die Überprüfung der Qualitätsfaktoren und -variablen wird mit Hilfe von neuen, speziell für die agile Entwicklung konzipierten Checklisten realisiert. Wenn diese Checklisten archiviert werden, kann für ein Projekt ein lückenloser Qualitätsnachweis erbracht werden. Außerdem bilden die archivierten Qualitätseinschätzungen und Maßnahmen eine wertvolle Grundlage für die Prozessverbesserung.

5.2.1 Die Basis: Scrum

Als zugrundeliegende agile Methode für den Prozess wurde Scrum gewählt. Bereits in Kapitel 3.2 wurde deutlich, dass die Methode einen Mittelweg zwischen hohem Konkretisierungsgrad und hoher Flexibilität bietet. Die vorgeschriebenen Techniken für das Projektmanagement bilden eine gute Grundlage für die Qualitätssicherstellung, während gleichzeitig genug Freiheiten zur Anpassung der Methode bestehen. Für Scrum sprachen außerdem das Personal-Zertifizierungsprogramm und – für die Firma GIGATRONIK als Zulieferer der Automobilindustrie besonders interessant – die in Kapitel 4.2.2 beschriebene Kompatibilität zu CMMI.

Der prinzipielle Rhythmus des Prozesses folgt damit dem von Scrum: Auf das Sprint Planning folgt ein Sprint von maximal 30 Tagen und fester Dauer. Das tägliche Daily Scrum dient der Synchronisation des Teams und der Identifikation von Hindernissen. Nach jedem Sprint werden im Sprint Review die fertige Funktionalität präsentiert und in der Sprint Retrospective Prozessverbesserungen eingebracht.

Das Hauptdokument zur Steuerung der Funktionalität und der Prioritäten ist das vom Product Owner geführte Product Backlog. Die wichtigsten Items stehen darin ganz oben. In jedem Sprint Planning übernimmt das Team eine selbst gewählte Menge der wichtigsten Items in das Sprint Backlog und verpflichtet sich zur Lieferung der dazugehörigen Funktionalität am Ende des Sprints. Der Scrum Master überwacht die Einhaltung des Prozesses und moderiert die Meetings.

Um eine Übersicht über bestehende und beseitigte Hindernisse zu erhalten, führt das Team zusätzlich ein Impediment Backlog, wie von Boris Gloger beschrieben.[51] Dabei handelt es sich schlicht um eine öffentlich geführte und für alle Teammitglieder sichtbar aufgehängte Liste, in die alle Hindernisse eingetragen werden.

5.2.2 Rollen

Bei der Betrachtung der Qualitätsfaktoren wurde deutlich, dass in Softwareprojekten viele Rollen Einfluss auf die Qualität haben. Scrum kennt nur die Rollen des Product Owners, des Teams und des Scrum Masters. Da aus Sicht der Qualitätssicherung die

[51] vgl. Gloger 2008, S. 17

Überwachung dieser Rollen nicht ausreicht, sind im qualitätsorientierten agilen Prozess drei zusätzliche Rollen definiert:

- **Kundenvertreter**: Durch die Rolle des Product Owners entfällt zwar die Notwendigkeit zur ständigen Anwesenheit von Kundenvertretern, doch ohne den Kontakt zu geeigneten Kundenvertretern kann der Product Owner das Product Backlog nicht pflegen und priorisieren. Wenn die Kundenvertreter die Zusammenarbeit verweigern, nicht autorisiert oder nicht fachkundig sind, ist das Projekt gefährdet. Die direkte Interaktion mit den Kundenvertretern ist außerdem wichtig, um zu verifizieren, dass der Product Owner den Kunden auf der Seite des Entwicklungsteams richtig vertritt.
- **Anwender**: Beim Testen von fertiger Funktionalität beim Sprint Review ist die Verfügbarkeit von echten Anwendern von großer Bedeutung für die Qualität. Durch die direkte Rückmeldung wird die Ebene der Kundenvertreter übersprungen, was nebenbei wiederum Rückschlüsse auf die Qualität der Arbeit selbiger ermöglicht.
- **Management**: Bei der Zusammenstellung von Entwicklungsteams und bei der Beseitigung von Hindernissen auf Organisationsebene ist das Management involviert. Es ist wichtig, zu erkennen, dass sich aus der Qualitätssicherung und der Prozessverbesserung dieses agilen Prozesses auch Pflichten für das Management ableiten.

Management, Scrum Master, Team und Product Owner sind Rollen des Dienstleisters, während Kundenvertreter und Anwender Rollen des Kunden sind. Kunden können je nach Projekt und Situation einen oder mehrere Kundenvertreter sowie einen oder mehrere Anwender stellen.

Während eines laufenden Sprints muss der Product Owner dem Team jederzeit kurzfristig persönlich für Rückfragen zur Verfügung stehen. Bei Bedarf muss der Kontakt zu Kundenvertretern ebenso kurzfristig hergestellt werden können.

5.2.3 Prozessablauf

Das Konzept der Quality Gates lässt sich nicht direkt auf einen agilen Prozess übertragen, da dort keine abgeschlossenen Phasen existieren, die man mit einem solchen „Tor“ strikt voneinander trennen könnte. Da Planung, Design, Integration und Test über den gesamten Prozess verteilt und auch Teil der täglichen Arbeit sind, muss folglich auch die Qualität fortlaufend kontrolliert werden.

Zur Überwachung und Bewertung der Qualitätsfaktoren werden daher zu markanten Zeitpunkten im Prozessablauf *Quality Meetings* durchgeführt. Diese Meetings zeichnen sich dadurch aus, dass zusätzlich zu den grundsätzlich anfallenden Aufgaben mit Hilfe einer individuellen Checkliste die jeweils relevanten Qualitätsfaktoren untersucht werden. Am Ende erfolgt durch die Anwesenden gemeinsam eine Bewertung, ob die Qualität sichergestellt werden kann oder ob dazu noch spezielle Maßnahmen erforderlich werden.

Wie **Abbildung 11** zeigt, haben Sprint Planning, Daily Scrum, Sprint Review und Sprint Retrospective den Status eines Quality Meetings erhalten. Zusätzlich wurden die Quality Meetings *Angebot*, *Beauftragung* und *Project Retrospective* definiert.

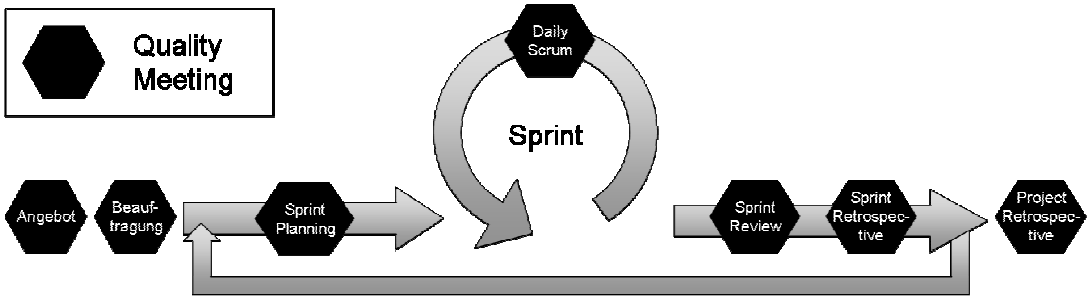

Abb. 11: Prozessablauf mit Quality Meetings

Tabelle 6 gibt eine Übersicht darüber, zu welchen Zeitpunkten die Quality Meetings anzusetzen sind und welche Rollen daran teilnehmen. Die Anwesenheit der in Klammern angegebenen Rollen ist zwar nicht zwingend erforderlich, erhöht aber die Transparenz des Prozesses.

Quality Meeting	Zeitpunkt	Anwesende Rollen
Angebot	Vor Abgabe des Angebotes	Management Product Owner
Beauftragung	Vor Annahme der Beauftragung	Management Product Owner Scrum Master
Sprint Planning	Vor jedem Sprint	Product Owner Kundenvertreter Team Scrum Master
Daily Scrum	Täglich während des Sprints	(Product Owner) Team Scrum Master
Sprint Review	Nach jedem Sprint	Product Owner Kundenvertreter Anwender Team Scrum Master
Sprint Retrospective	Nach jedem Sprint Review, auch nach abgebrochenem Sprint	(Management) Team Scrum Master
Project Retrospective	Nach dem Projekt, auch nach Abbruch	Management Product Owner Scrum Master

Tab. 6: Organisation der Quality Meetings

Wenn in den Quality Meetings Angebot, Beauftragung oder Sprint Planning festgestellt wird, dass unter den aktuellen Umständen die Qualität nicht sichergestellt werden kann, besteht die Möglichkeit, ein neues Meeting anzusetzen und bis dahin die bestehenden Probleme auszuräumen. In der Zwischenzeit ist der normale Prozessablauf blockiert. Wenn die Beseitigung der Probleme nicht möglich ist, kann das Projekt auch abgebrochen werden.

In den restlichen Quality Meetings besteht keine Möglichkeit zur Blockade des Prozesses. Hier kann lediglich entschieden werden, ob wie bisher fortgefahren werden soll, bestimmte Maßnahmen notwendig werden oder Sprint oder Projekt abgebrochen werden müssen. Qualitätsprobleme sollen damit möglichst parallel zum laufenden Betrieb ausgeräumt werden.

5.2.4 Kompatibilität zum GIGATRONIK-Prozess

In den Quality Meetings werden primär die neuen, speziell auf den agilen Prozess zugeschnittenen Checklisten verwendet. Durch die Konzeption der Quality Meetings wurde allerdings sichergestellt, dass auch die bestehenden Checklisten der Firma GIGATRONIK zur zusätzlichen Beurteilung der Qualität weiter verwendet werden können. Ausgenommen davon sind selbstverständlich Checklisten und Punkte, die sich explizit auf Eigenschaften des GIGATRONIK-Entwicklungsprozesses beziehen.

Tabelle 7 zeigt, in welchen Quality Meetings die bestehenden Checklisten verwendet werden können.

Quality Meeting	Checklisten des/der Quality Gates
Angebot	Projektbeginn, Angebot
Beauftragung	Beauftragung
Sprint Planning	Design
Daily Scrum	Design, Realisierung, Stabilisierung
Sprint Review	Stabilisierung, Kontrolle
Sprint Retrospective	-
Project Retrospective	Projektende

Tab. 7: Kompatibilität zum GIGATRONIK-Prozess

5.3 Checklisten

In der Kopfzeile jeder Checkliste ist angegeben, für welches Quality Meeting sie vorgesehen ist. Direkt darunter findet sich auf jeder Liste ein Bereich, in dem das Datum sowie Informationen zum Projekt und zum Bearbeiter eingetragen werden. Bei allen Quality Meetings, die einem bestimmten Sprint zugeordnet sind, wird zusätzlich die laufende Nummer des Sprints notiert (siehe **Abbildung 12**). Durch diese Systematik wird die Sortierung und Archivierung der Checklisten erleichtert.

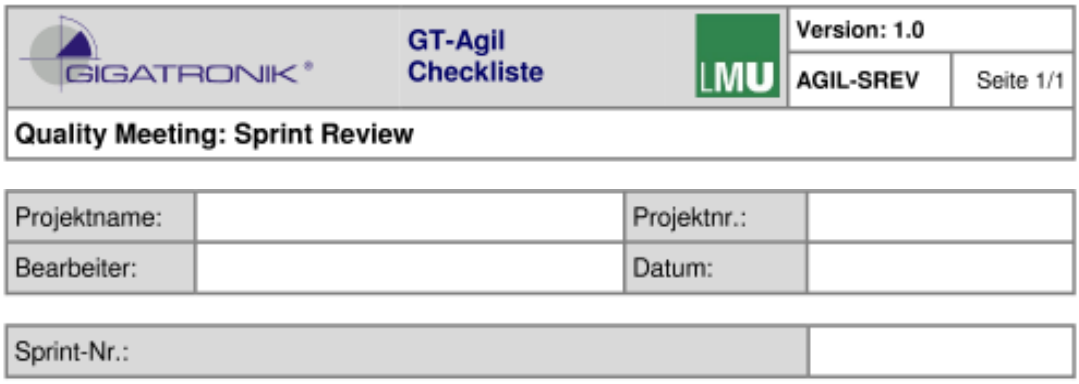

Abb. 12: Kopfbereich einer Checkliste

In jeder Checkliste werden gezielt die in Kapitel 4.1 identifizierten Qualitätsvariablen abgefragt, die zum jeweiligen Zeitpunkt beeinflusst werden können oder von großer Bedeutung für den Zeitraum bis zum nächsten Quality Meeting sind. **Tabelle 8** gibt eine Übersicht darüber, in welche Checklisten die Qualitätsfaktoren eingeflossen sind.

	Kommunikation	Erfahrung	Timeboxing	Einbindung des Kunden	Sukzessives Design und Refactoring	Kontinuierliche Integration und Test	Prozessverbesserung
Angebot		x					
Beauftragung	x	x		x			
Sprint Planning	x		x	x			
Daily Scrum	x			x	x	x	x
Sprint Review			x	x			
Sprint Retrospective	x	x		x			x
Project Retrospective		x		x			x

Tab. 8: Einfluss der Qualitätsfaktoren auf die Checklisten

Alle Fragen in den Checklisten sind so gestaltet, dass sie mit „ja“ oder „nein“ beantwortet werden können. Ein „ja“ bedeutet dabei immer, dass die Qualität in diesem Punkt gewährleistet werden kann. Ein „nein“ deutet dagegen stets auf ein mögliches Qualitätsproblem hin. Allein aus der Betrachtung der Antworten lässt sich damit sehr schnell erkennen, wie es um die Qualität steht.

Alle mit „nein“ beantworteten Fragen müssen näher erläutert werden. Hierzu steht am Ende jeder Checkliste ein Feld für Anmerkungen zur Verfügung. Darin ist anzugeben, ob das Problem toleriert werden kann oder ob es ausgeräumt werden muss. Im letzteren Fall ist anzugeben, welche Maßnahmen zur Beseitigung getroffen werden und wer dafür verantwortlich ist.

Zum Schluss jeder Checkliste wird das Fazit des Quality Meetings notiert. Dabei gibt es wie in **Abbildung 13** ersichtlich jeweils eine Variante für die blockierenden und für die nicht blockierenden Quality Meetings.

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Angebot erstellen	☐ nein, neues Meeting am: _____	☐ nein, Abbruch

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Sprint fortsetzen	☐ ja, aber bestimmte Maßnahmen erforderlich	☐ nein, Abbruch

Abb. 13: Fußbereich zweier Checklisten für ein blockierendes und ein nicht blockierendes Quality Meeting

Auf den folgenden Seiten sind die Checklisten aller Quality Meetings abgedruckt. Die Listen passen jeweils auf eine DIN A4-Seite, im praktischen Gebrauch kann es jedoch unter Umständen notwendig werden, zusätzliche Zeilen hinzuzufügen.

5.3.1 Angebot

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-ANG	Seite 1/1
Quality Meeting: Angebot				

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Geschäftsfeld / Softwaretyp (z. B. Web-Geschäftsanwendung, Steuergeräte-Software)		
Liegen Erfahrungen mit diesem Geschäftsfeld vor?	☐ ja	☐ nein
Referenzprojekte oder Mitarbeiter mit Erfahrung hier benennen:		

Eingesetzte Technologien (z. B. Programmiersprache, Frameworks, IDE)	Liegt *praktische* Erfahrung im Hause vor?	
1.	☐ ja	☐ nein
2.	☐ ja	☐ nein
3.	☐ ja	☐ nein
4.	☐ ja	☐ nein
Bieten diese Technologien:		
Versionsverwaltung?	☐ ja	☐ nein
Refactoring-Funktionalität?	☐ ja	☐ nein
Automatisierte Tests?	☐ ja	☐ nein
Können alle bekannten nichtfunktionalen Anforderungen mit diesen Technologien erfüllt werden?	☐ ja	☐ nein
Bekannte nichtfunktionale Anforderungen:		

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Angebot erstellen	☐ nein, neues Meeting am: _____	☐ nein, Abbruch

5.3.2 Beauftragung

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-BEAUF	Seite 1/1

Quality Meeting: Beauftragung

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Rollen			
Product Owner		Scrum Master	
Team			

Beurteilung der Erfahrung		
Ist der Product Owner zertifiziert oder ausreichend erfahren?	☐ ja	☐ nein
Ist der Scrum Master zertifiziert oder ausreichend erfahren?	☐ ja	☐ nein
Ist mindestens ein Teammitglied Experte im Geschäftsfeld?	☐ ja	☐ nein
Ist für jede eingesetzte Technologie ein Experte im Team?	☐ ja	☐ nein
Hat jedes Teammitglied Grundkenntnisse im agilen Prozess?	☐ ja	☐ nein

Kundenvertreter	Position	Erreichbarkeit	
1.			
2.			
3.			
Sind die Kundenvertreter zur Zusammenarbeit bereit?		☐ ja	☐ nein
Sind die Kundenvertreter repräsentativ?		☐ ja	☐ nein
Sind die Kundenvertreter entscheidungsbefugt?		☐ ja	☐ nein
Sind die Kundenvertreter bezüglich des Projektes sachkundig?		☐ ja	☐ nein
Fühlen sich die Kundenvertreter dem Projekt verpflichtet?		☐ ja	☐ nein
Stehen die Kundenvertreter kurzfristig für Rückfragen bereit?		☐ ja	☐ nein
Kann über die Kundenvertreter Kontakt zu Anwendern hergestellt werden?		☐ ja	☐ nein

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Entwicklung starten	☐ nein, neues Meeting am: _____	☐ nein, Abbruch

5.3.3 Sprint Planning

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-SPL	Seite 1/1

Quality Meeting: Sprint Planning

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Sprint-Nr.:		Sprint-Dauer:	

Product Backlog		
War der Kunde in die Erstellung und Priorisierung einbezogen?	☐ ja	☐ nein
Ist das Product Backlog auf dem aktuellen Stand?	☐ ja	☐ nein

Sprint Backlog (wird vom Team nach Priorität aus dem Product Backlog ausgewählt)

ID	Name	Aufwand

Besteht Einigkeit zwischen Kunde, Product Owner und Team, wie diese Backlog Items umgesetzt werden sollen?	☐ ja	☐ nein
Verpflichtet sich das Team zur Lieferung dieser Funktionalität?	☐ ja	☐ nein

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Sprint starten	☐ nein, neues Meeting am: _____	☐ nein, Abbruch

5.3.4 Daily Scrum

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-DSCR	Seite 1/1

Quality Meeting: Daily Scrum

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Sprint-Nr.:		Tag-Nr.:	

Kommunikation / Kultur		
Herrscht eine Kultur der Anerkennung und Wertschätzung?	☐ ja	☐ nein
Hat sich das Team über Aktivitäten/Probleme ausgetauscht?	☐ ja	☐ nein
Steht der Product Owner, wenn nötig der Kunde, für Rückfragen kurzfristig zur Verfügung?	☐ ja	☐ nein

Integration und Test		
Ist sämtlicher Code integriert?	☐ ja	☐ nein
Wurden alle Integrations- und Regressionstests bestanden?	☐ ja	☐ nein

Design- und Codequalität		
Wurde das Design sukzessive an das wachsende Projekt angepasst und im Code durch Refactoring umgesetzt?	☐ ja	☐ nein
Entspricht sämtlicher Code den Qualitätskriterien?	☐ ja	☐ nein
Ist die Software benutzbar (lauffähig, Benutzerschnittstelle)?	☐ ja	☐ nein

Dokumentation		
Wurden alle Hindernisse im Impediment Backlog dokumentiert?	☐ ja	☐ nein
Wurde die Entwicklungsgeschwindigkeit gemessen und dokumentiert?	☐ ja	☐ nein

Erfahrungen mit getesteten Verbesserungen

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Sprint fortsetzen	☐ ja, aber bestimmte Maßnahmen erforderlich	☐ nein, Abbruch

5.3.5 Sprint Review

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-SREV	Seite 1/1
Quality Meeting: Sprint Review				

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Sprint-Nr.:	

Sprint Backlog			
ID	Name	Fertig und benutzbar?	
		☐ ja	☐ nein
		☐ ja	☐ nein
		☐ ja	☐ nein
		☐ ja	☐ nein
		☐ ja	☐ nein
		☐ ja	☐ nein
		☐ ja	☐ nein
Entsprechen alle fertigen Items den Qualitätsrichtlinien?		☐ ja	☐ nein

Review		
Wurden alle fertigen Items dem Kunden präsentiert?	☐ ja	☐ nein
Haben repräsentative Endanwender das Produkt getestet?	☐ ja	☐ nein

Product Backlog		
Wurden unfertige Sprint Backlog Items neu priorisiert?	☐ ja	☐ nein
Wurden die Änderungswünsche des Kunden / der Anwender in das Product Backlog eingebracht und priorisiert?	☐ ja	☐ nein

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Projekt fortsetzen	☐ ja, aber bestimmte Maßnahmen erforderlich	☐ nein, Abbruch

5.3.6 Sprint Retrospective

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-SRET	Seite 1/1

Quality Meeting: Sprint Retrospective

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Sprint-Nr.:	

Kommunikation / Kultur		
Herrscht zwischen allen Projektbeteiligten eine Kultur der Anerkennung und Wertschätzung?	☐ ja	☐ nein
Waren alle Teammitglieder stets über die Aktivitäten und Probleme der anderen informiert?	☐ ja	☐ nein
Wurden die regulären Arbeitszeiten eingehalten?	☐ ja	☐ nein
Ermöglicht die aktuelle Sprintdauer angenehmes Arbeiten?	☐ ja	☐ nein

Erfahrung		
Ist die Erfahrung des Teams im Prozess-, Technologie- und Geschäftsfeld ausreichend?	☐ ja	☐ nein

Unbeseitigte Impediments / Hindernisse: (aus dem Impediment Backlog)		
1.		
2.		
Erklären sich allein daraus eventuelle Nicht-Lieferungen?	☐ ja	☐ nein
Verpflichten sich die Verantwortlichen, die Hindernisse auszuräumen?	☐ ja	☐ nein

Verbesserungen / Hilfsmittel (Im Daily Scrum dokumentierte Erfahrungen einfließen lassen)		
Ausprobieren (im nächsten Sprint)	Beibehalten (haben sich bewährt)	
1.	1.	
2.	2.	
Verpflichtet sich das Team zur Umsetzung?	☐ ja	☐ nein

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

Fazit: Kann die Qualität sichergestellt werden?		
☐ ja, Projekt fortsetzen	☐ ja, aber bestimmte Maßnahmen erforderlich	☐ nein, Abbruch

5.3.7 Project Retrospective

GIGATRONIK®	GT-Agil Checkliste	LMU	Version: 1.0	
			AGIL-PRET	Seite 1/1

Quality Meeting: Project Retrospective

Projektname:		Projektnr.:	
Bearbeiter:		Datum:	

Kundenzufriedenheit		
Ist der Kunde mit der agilen Zusammenarbeit zufrieden?	☐ ja	☐ nein
Ist der Kunde mit dem Endprodukt zufrieden?	☐ ja	☐ nein
Sind repräsentative Endanwender mit dem Produkt zufrieden?	☐ ja	☐ nein

Erfahrung		
War die Erfahrung des Teams im Prozess-, Technologie- und Geschäftsfeld ausreichend?	☐ ja	☐ nein

Unbeseitigte Impediments / Hindernisse		
1.		
2.		
3.		
Können diese Hindernisse zukünftig ausgeräumt werden?	☐ ja	☐ nein
Maßnahmen		

Bewährte und beibehaltene Verbesserungen / Hilfsmittel		
1.		
2.		
3.		
4.		
5.		
Werden diese Verbesserungen institutionalisiert?	☐ ja	☐ nein

Anmerkungen (mit „nein" beantwortete Fragen müssen hier begründet werden!)

5.4 Ergebnis

Es ist gelungen, mit den Quality Meetings ein agiles Äquivalent zu den Quality Gates des GIGATRONIK-Entwicklungsprozesses zu konstruieren. Durch sie unterstehen die gerade relevanten Qualitätsfaktoren zu jeder Zeit einer genauen Beobachtung. So können eventuelle Qualitätsprobleme rechtzeitig erkannt und gegebenenfalls entsprechende Korrekturmaßnahmen eingeleitet werden.

Während die Quality Gates im Prozessablauf bei Qualitätsproblemen grundsätzlich als Blockade wirken, wird bei den Quality Meetings zwischen blockierenden und nicht blockierenden Meetings unterschieden. Bei Angebot, Beauftragung und Sprint Planning, also vor dem Beginn der iterativen Entwicklung, führen ernste Probleme zu einer Aufschiebung des weiteren Prozessablaufes. Die Probleme können anschließend iterativ – durch die Abarbeitung der anfallenden Aufgaben und die Wiederholung des Quality Meetings – gelöst werden. Durch die Blockade wird sichergestellt, dass die notwendigen Voraussetzungen für eine qualitativ hochwertige, produktive, ungestörte und ununterbrochene Arbeit geschaffen sind, bevor ein Sprint begonnen wird. Die Quality Meetings während und nach einem Sprint wirken im Gegensatz dazu nicht blockierend, da mit den Sprints selbst bereits ein iterativer Rahmen besteht, innerhalb dessen aufgetretene Probleme beseitigt werden können: Parallel zur regulären Arbeit im laufenden Sprint oder in den folgenden Sprints.

Die blockierende Sonderrolle von Angebot, Beauftragung und Sprint Planning ergibt sich daraus, dass in diesen Meetings Qualitätsfaktoren überprüft werden, die für die korrekte Funktion des agilen Prozesses an sich von entscheidender Wichtigkeit sind. Zur Sicherstellung der Qualität ist also auch in der agilen Softwareentwicklung eine gewisse Vorausplanung notwendig; überwiegend konnte die Qualitätssicherung in diesem agilen Prozess aber wesentlich dynamischer gestaltet werden als es in schwerfälligen plangesteuerten Prozessen üblich ist.

Zum Schluss soll noch darauf hingewiesen werden, dass der entwickelte agile Prozess Techniken zur Prozessverbesserung beinhaltet, wodurch selbstverständlich auch eine evolutionäre Verbesserung der Checklisten ermöglicht wird. Insgesamt stellt sich der Prozess damit als guter Ausgangspunkt für einen Start in die qualitätsorientierte agile Softwareentwicklung dar.

6 Praktisches Projekt: GIGATRONIK-Gefahrstoffmanagement

Im Rahmen dieser Arbeit wurde bei der Firma GIGATRONIK ein praktisches Softwareentwicklungsprojekt durchgeführt. Leider standen nicht genügend Personalressourcen zur Verfügung, um die Rollen des qualitätsorientierten agilen Prozesses tatsächlich zu besetzen. Als einziges Teammitglied konnte der Autor dieser Arbeit somit weder echte Meetings durchführen noch Kompromisse mit anderen eingehen oder überstimmt werden.

Durch das Projekt sollte hauptsächlich sondiert werden, ob die agile Entwicklung in der Praxis problemlos eingesetzt werden kann, ob die von GIGATRONIK favorisierte Technologie für die agile Entwicklung geeignet ist und ob im Laufe der Entwicklung unvorhergesehene Qualitätsprobleme auftreten. Die Auswertung dieser Probleme ermöglicht aber zumindest einen Rückschluss darauf, ob die Probleme mit dem qualitätsorientierten agilen Prozess frühzeitig erkannt worden wären.

6.1 Projektbeschreibung

Die Abteilung *Umweltsysteme* der Firma GIGATRONIK München GmbH plant die Entwicklung einer Gefahrstoffmanagement-Software für mittelständische produzierende Unternehmen. In diesem Geschäftsfeld verfügt GIGATRONIK über langjährige Erfahrung aus der Entwicklung und dem Betrieb eines umfangreichen Systems (ZEUS, Zentrale Erfassung umweltrelevanter Stoffe) für die Firma BMW (siehe **Abbildung 14**) und der Software myMDS (siehe **Abbildung 15**).

Abb. 14: ZEUS der Firma BMW, entwickelt von GIGATRONIK

Die Firma GIGATRONIK erwägt, zur Realisierung dieser Software agile Methoden und die Technologien Oracle ADF und JDeveloper einzusetzen. Dazu liegen jedoch bisher keine praktischen Erfahrungen vor.

Daher sollte in diesem Projekt mit agilen Methoden und unter der Verwendung dieser Technologien ein Prototyp erstellt werden. Die Ergebnisse sollen in die Entscheidung einfließen, wie und mit welcher Technologie die eigentliche Software entwickelt wird.

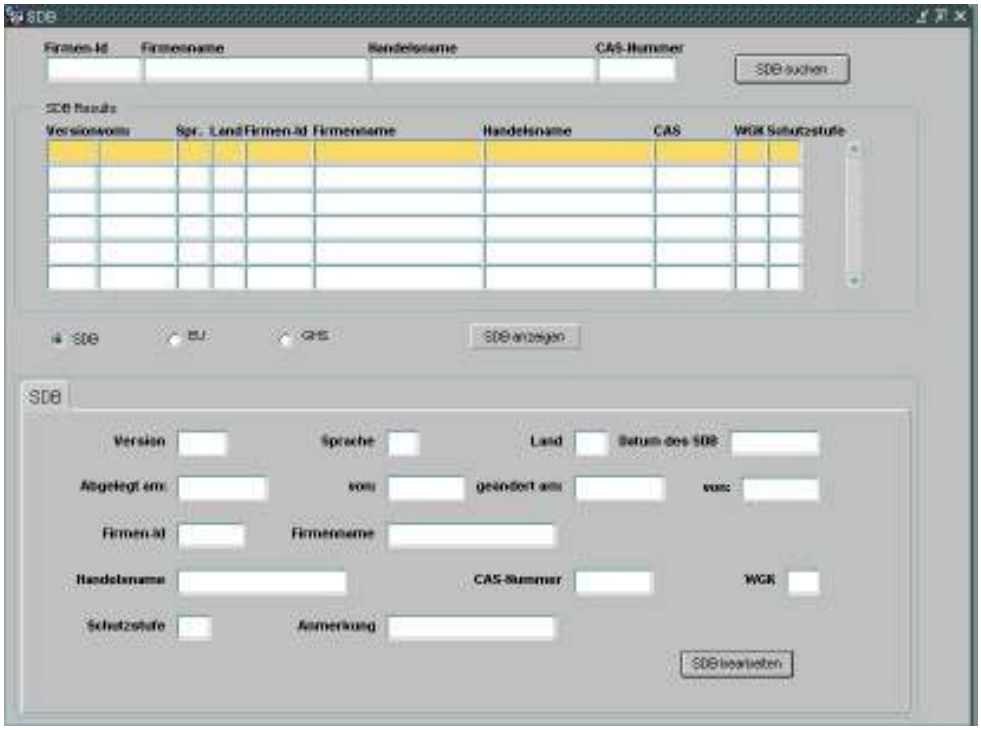

Abb. 15: myMDS von GIGATRONIK

6.2 Technologie

Oracle ADF ist ein Java EE-basiertes Model-View-Controller-Metaframework, das in den einzelnen Schichten die Verwendung unterschiedlicher Technologien ermöglicht. **Abbildung 16** zeigt die Architektur des Frameworks. Oracle wirbt damit, dass bei der Verwendung von Oracle-Technologien in allen Schichten in Verbindung mit der Oracle-Entwicklungsumgebung JDeveloper eine einfache, komponentenbasierte Web-Entwicklung möglich ist und datengebundene Benutzeroberflächen direkt aus dem Modell generiert werden können.

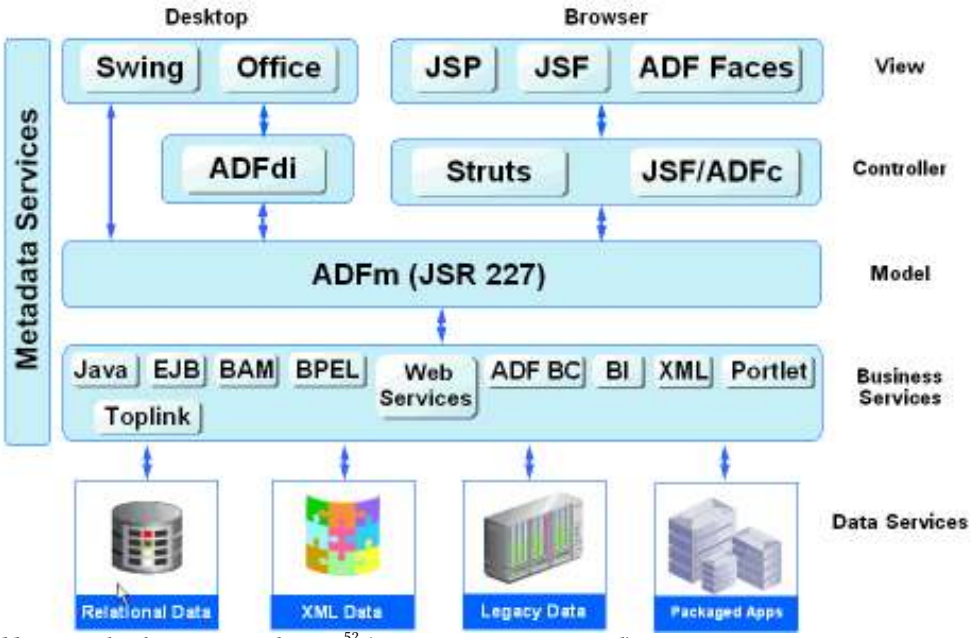

Abb. 16: Architektur von Oracle ADF[52] (Mauszeiger ist im Original)

[52] Oracle Corporation 2009

Um diese vielversprechenden Vorteile voll ausschöpfen zu können, wurden für das Projekt in allen Schichten Oracle-Technologien verwendet:

- **Data Services**: Oracle Database 10g
- **Business Services**: ADF Business Components
- **Model**: ADFm (ADF Model)
- **Controller**: ADFc (ADF Controller)
- **View**: ADF Faces

ADF und JDeveloper kamen in der Version 11g zum Einsatz.

6.3 Durchführung

Zu Beginn des Projektes wurde zunächst die Funktionalität des Prototyps umrissen. Aus dem Use-Case-Diagramm in **Abbildung 17** ist ersichtlich, dass die Software Workflows der Rollen „Gefahrstoffbeauftragter“ und „Mitarbeiter“ unterstützen und die Kommunikation im Bereich der Auskunft über Gefahrstoffe und der Beantragung von Freigaben vereinfachen soll.

Das Diagramm impliziert, dass die Software zwischen den beiden genannten Rollen unterscheidet und je nach Rolle eine unterschiedliche Funktionalität bereitstellt. Aus diesem Grund muss zusätzlich eine Benutzeranmeldung implementiert werden.

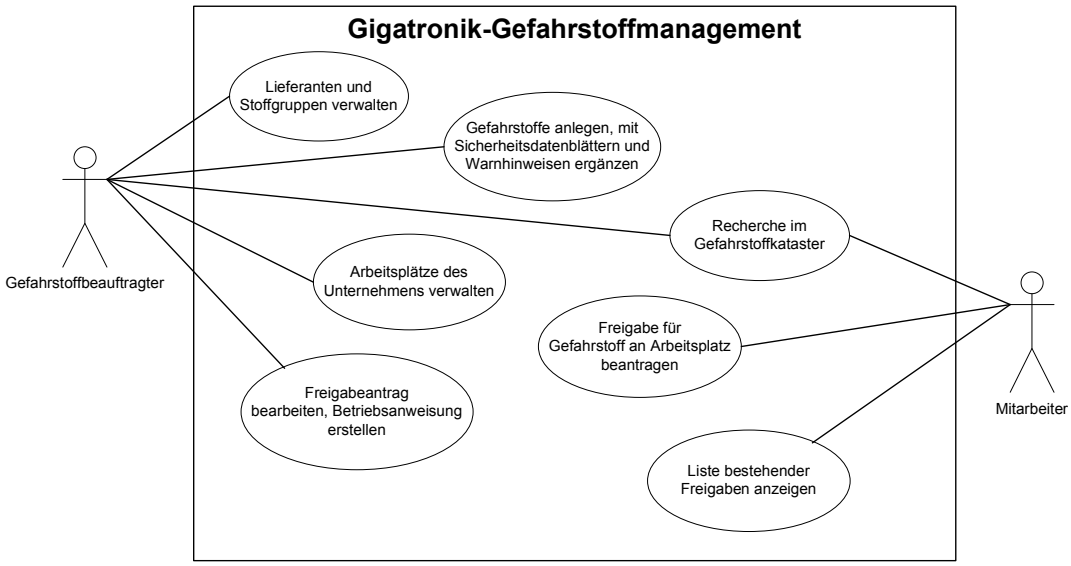

Abb. 17: Use-Case-Diagramm zum Gefahrstoffmanagement-Prototypen

Aufgrund des kleinen Projektumfangs und der Personalsituation erschien eine kurze Sprint-Dauer sinnvoll. Die Dauer wurde auf 10 Arbeitstage festgesetzt und während des Projektes beibehalten.

Die Qualitätsrichtlinien für den Code wurden mündlich formuliert: Jeglicher Code muss übersichtlich und kommentiert sein und durch Unit-Tests abgedeckt werden.

Das Product Backlog wurde wie in **Tabelle 9** dargestellt initial befüllt. Aus den **Tabellen 9-14** ist ersichtlich, welche Items in den jeweiligen Sprints zur Umsetzung vorgesehen waren und wann sie tatsächlich realisiert werden konnten. Insgesamt vier Items konnten nicht rechtzeitig fertig gestellt werden. Die Items 3 und 13 wurden daraufhin in der Priorität abgestuft, die Items 8 und 9 wurden wieder in den nächsten Sprint aufgenommen.

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1
2	Benutzeranmeldung	1
3	Benutzerverwaltung	1
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	
5	Hersteller verwalten	
6	Stoffgruppen verwalten	
7	Gefahrstoffe anlegen	
8	Warnhinweise zu Stoffen ergänzen	
9	Sicherheitsdatenblätter zu Stoffen ergänzen	
10	Arbeitsplätze des Unternehmens verwalten	
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	
12	Freigabeanträge bearbeiten	
13	Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen	

Tab. 9: Initiales Product Backlog

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1 ✓
2	Benutzeranmeldung	1 ✓
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	2
5	Hersteller verwalten	2
6	Stoffgruppen verwalten	2
7	Gefahrstoffe anlegen	
8	Warnhinweise zu Stoffen ergänzen	
9	Sicherheitsdatenblätter zu Stoffen ergänzen	
10	Arbeitsplätze des Unternehmens verwalten	
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	
12	Freigabeanträge bearbeiten	
13	Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen	
3	Benutzerverwaltung	~~1~~

Tab. 10: Product Backlog vor Sprint 2

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1 ✓
2	Benutzeranmeldung	1 ✓
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	2 ✓
5	Hersteller verwalten	2 ✓
6	Stoffgruppen verwalten	2 ✓
7	Gefahrstoffe anlegen	3
8	Warnhinweise zu Stoffen ergänzen	3
9	Sicherheitsdatenblätter zu Stoffen ergänzen	3
10	Arbeitsplätze des Unternehmens verwalten	
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	
12	Freigabeanträge bearbeiten	
13	Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen	
3	Benutzerverwaltung	~~1~~

Tab. 11: Product Backlog vor Sprint 3

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1 ✓
2	Benutzeranmeldung	1 ✓
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	2 ✓
5	Hersteller verwalten	2 ✓
6	Stoffgruppen verwalten	2 ✓
7	Gefahrstoffe anlegen	3 ✓
8	Warnhinweise zu Stoffen ergänzen	~~3~~ 4
9	Sicherheitsdatenblätter zu Stoffen ergänzen	~~3~~ 4
10	Arbeitsplätze des Unternehmens verwalten	
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	
12	Freigabeanträge bearbeiten	
13	Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen	
3	Benutzerverwaltung	~~1~~

Tab. 12: Product Backlog vor Sprint 4

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1 ✓
2	Benutzeranmeldung	1 ✓
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	2 ✓
5	Hersteller verwalten	2 ✓
6	Stoffgruppen verwalten	2 ✓
7	Gefahrstoffe anlegen	3 ✓
8	Warnhinweise zu Stoffen ergänzen	~~3~~ 4 ✓
9	Sicherheitsdatenblätter zu Stoffen ergänzen	~~3~~ 4 ✓
10	Arbeitsplätze des Unternehmens verwalten	5
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	5
12	Freigabeanträge bearbeiten	5
13	Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen	5
3	Benutzerverwaltung	~~1~~

Tab. 13: Product Backlog vor Sprint 5

Nr	Name der Story	Sprint
1	Einheitliches Look&Feel der Oberfläche mit Navigation	1 ✓
2	Benutzeranmeldung	1 ✓
4	Oberfläche soll auf Deutsch und Englisch bedienbar sein	2 ✓
5	Hersteller verwalten	2 ✓
6	Stoffgruppen verwalten	2 ✓
7	Gefahrstoffe anlegen	3 ✓
8	Warnhinweise zu Stoffen ergänzen	~~3~~ 4 ✓
9	Sicherheitsdatenblätter zu Stoffen ergänzen	~~3~~ 4 ✓
10	Arbeitsplätze des Unternehmens verwalten	5 ✓
11	Freigabe für Gefahrstoff an Arbeitsplatz beantragen	5 ✓
12	Freigabeanträge bearbeiten	5 ✓
~~13~~	~~Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen~~	~~5~~
~~3~~	~~Benutzerverwaltung~~	~~1~~

Tab. 14: Product Backlog bei Projektende

6.4 Ergebnis

Einheitliches Look&Feel der Oberfläche mit Navigation:

Um ein einheitliches „Look & Feel“ für alle Seiten zu erreichen, wurden mehrere *Java Server Faces Templates* angelegt. Die Templates unterscheiden sich lediglich in der angezeigten Menü-Tiefe.

Die Struktur des *ADF Menu* wird über eine XML-Datei konfiguriert.

Abb. 18: Menüstruktur

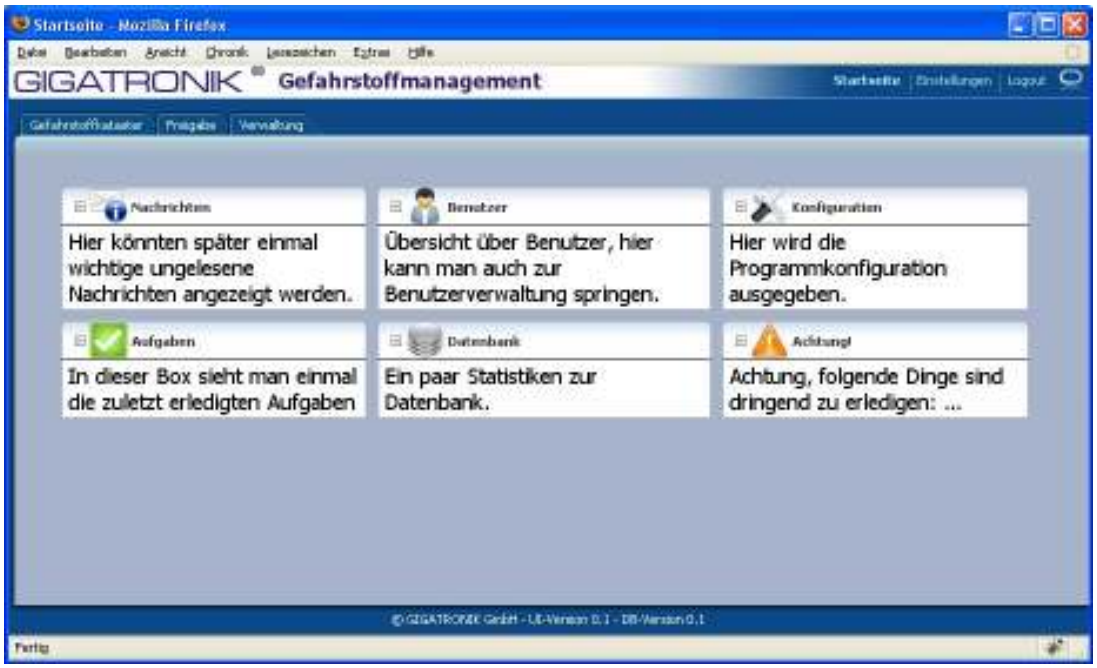

Abb. 19: Seitenlayout

Benutzeranmeldung:

ADF setzt auf *Container Managed Security*, das heißt: Für die Authentifizierung ist der Application Server zuständig. Im Server wird konfiguriert, ob die Benutzer gegen ein LDAP-Verzeichnis, eine XML-Datei oder eine Datenbank geprüft werden. Um einen Benutzer zu authentifizieren, muss die Anwendung die Login-Daten an den Application Server übertragen. Der Server gibt dann in einer Antwort an die Anwendung zurück, ob der Benutzer bekannt ist und über welche Rechte er verfügt. Es besteht also keine direkte Verbindung zwischen Benutzerdatenbank und Anwendung!

Das Problem dieses Konzeptes ist, dass die Schnittstelle zwischen Anwendung und Application Server nicht standardisiert ist und damit abhängig vom eingesetzten Application Server ist.

Für den Prototyp wurde unter Verwendung der proprietären WebLogic-API die Übergabe von Benutzername und Kennwort an den WebLogic-Server implementiert. Sofern der Benutzer beim Server bekannt ist, erfolgt eine Weiterleitung auf die Startseite, ansonsten erscheint eine Fehlermeldung. Der Prototyp ist nun allerdings an den WebLogic-Server gebunden.

Abb. 20: Anmeldemaske

Benutzerverwaltung:

Ursprünglich war es vorgesehen, die Benutzer in einer Datenbank zu halten und über die Anwendung zu verwalten. Wegen des Konzepts Container Managed Security hätte dazu jedoch der WebLogic-Server so konfiguriert werden müssen, dass die Datenbank als Speicher für die Zugangsdaten verwendet wird. Diese Konfiguration war mit vertretbarem Aufwand und dem vorhandenen Wissen nicht umsetzbar.

Die Benutzer wurden daher vorerst in einer XML-Datei vorgehalten. Das geplante Backlog Item „Benutzerverwaltung“ wurde zurückgestellt.

Oberfläche soll auf Deutsch und Englisch bedienbar sein:

Die Benutzeroberfläche von ADF-Anwendungen kann mittels Java Resource Bundles mehrsprachig gestaltet werden. Dabei werden für Texte Platzhalter vergeben, die zur Laufzeit mit den Strings aus dem Resource Bundle gefüllt werden.

Es wurde je ein Resource Bundle für die Sprachen Deutsch und Englisch angelegt. Durch die ausgelagerten Strings kann eine Anwendung leicht in weitere Sprachen übersetzt werden. Der Umgang mit den Platzhaltern ist während der Entwicklung jedoch relativ aufwändig. Hier wäre eine bessere Integration des Konzeptes in die Entwicklungsumgebung äußerst hilfreich.

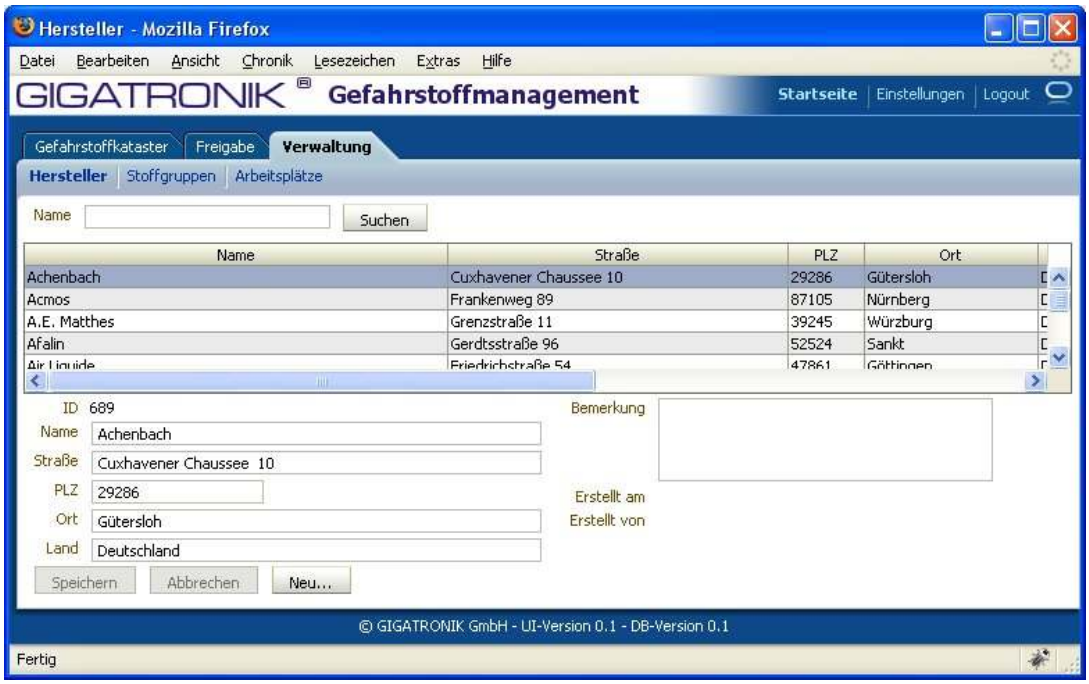

Abb. 21: Oberfläche auf Deutsch

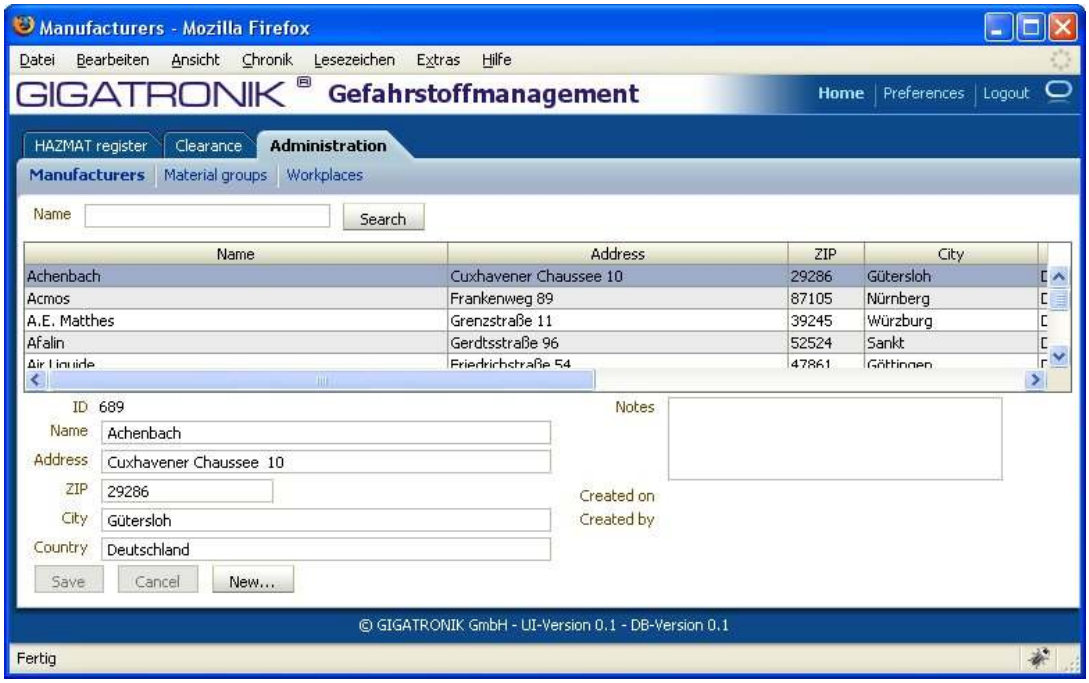

Abb. 22: Oberfläche auf Englisch

Hersteller verwalten:

Alle Hersteller werden in einer durchsuchbaren Liste angezeigt. Neue Datensätze können erfasst werden, Daten von bestehenden Herstellern können angepasst werden.

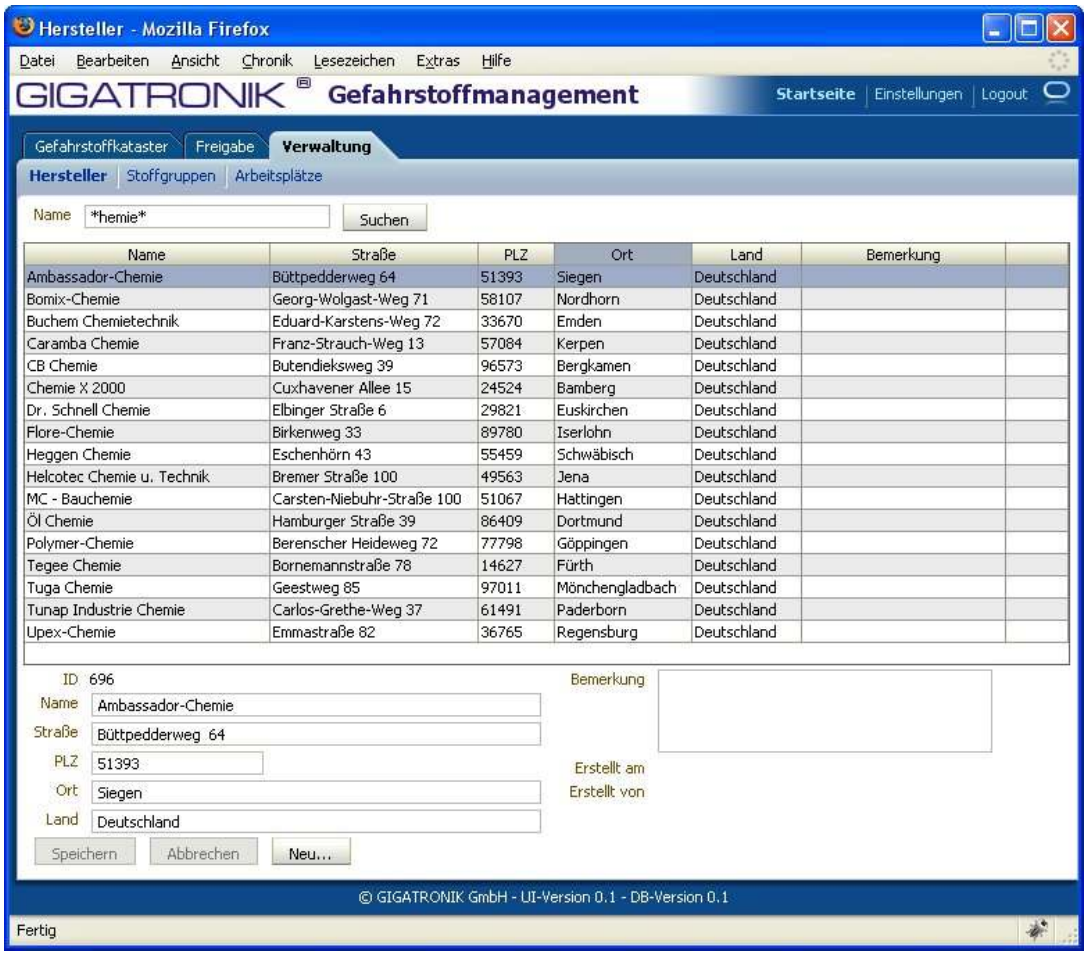

Abb. 23: Herstellerverwaltung

Stoffgruppen verwalten:

Übergeordnete Stoffgruppen können angelegt und bearbeitet werden. Die Stoffgruppen ermöglichen später eine Gruppierung der Stoffe, zum Beispiel bei der Suche oder auf Reports.

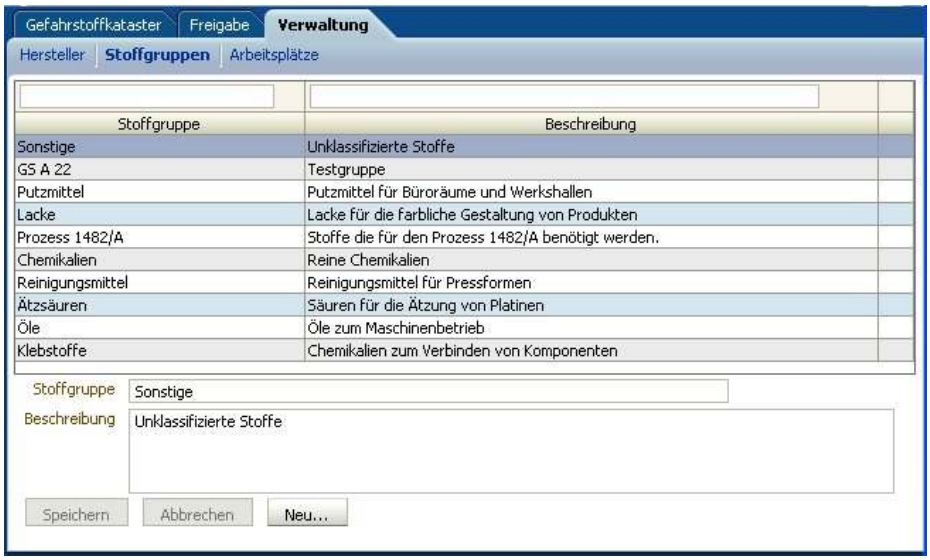

Abb. 24: Stoffgruppenverwaltung

Gefahrstoffe anlegen:

Für die Stoffe wurden drei verschiedene Ansichten erstellt: eine tabellarische Ansicht mit Suchfunktion, eine Detailansicht, und eine Bearbeitungsansicht. Dank der Model-View-Controller-Architektur von ADF verwenden alle drei Ansichten dabei dasselbe Datenmodell.

Stoffe können gesucht und neu angelegt werden, Stammdaten von bestehenden Stoffen können bearbeitet werden.

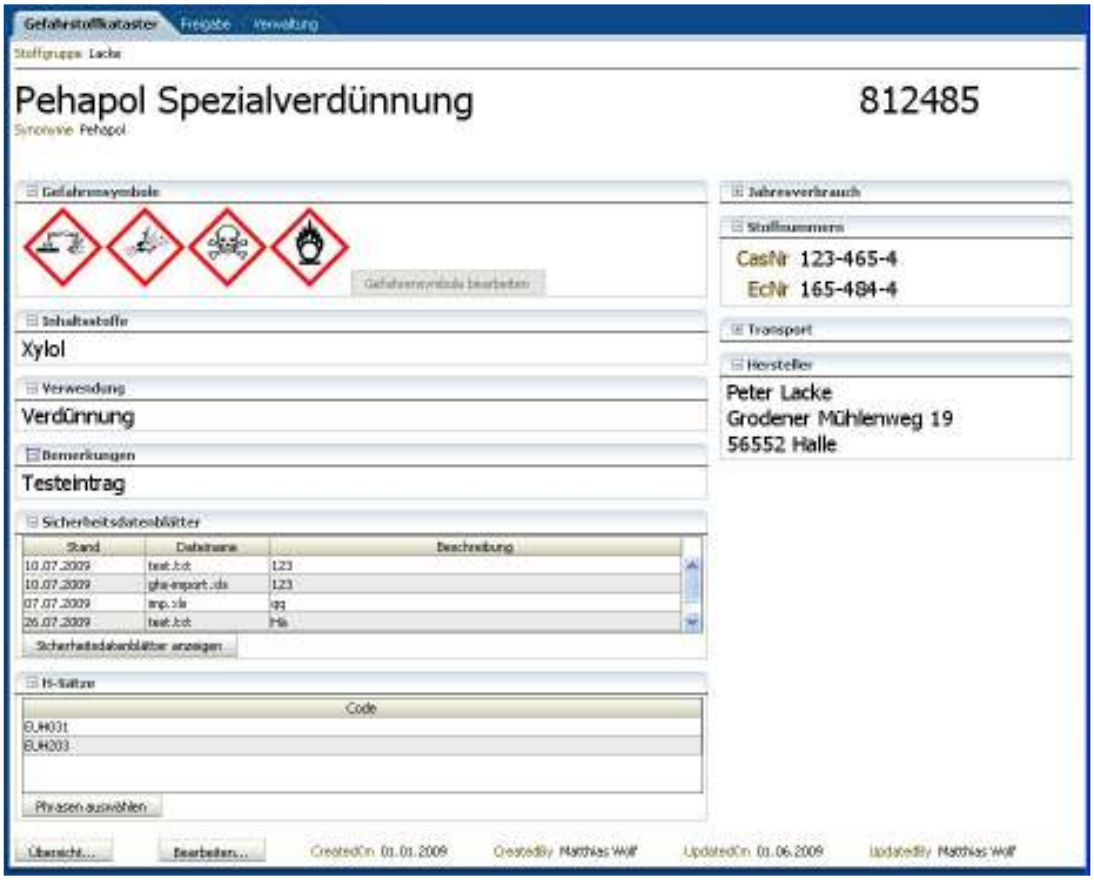

Abb. 25: Detailansicht eines Gefahrstoffes

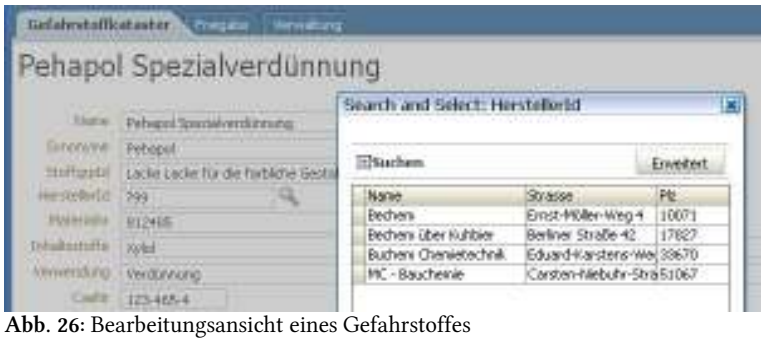

Abb. 26: Bearbeitungsansicht eines Gefahrstoffes

Warnhinweise zu Stoffen ergänzen:

Phrasen aus einem Katalog in der Datenbank können in die linke „Leading List“ geladen werden. Eine beliebige Kombination kann vom Benutzer in die rechte „Trailing List“ verschoben werden und beliebig sortiert werden. Die gewählten Phrasen werden in einer Zuordnungstabelle incl. Sortierung gespeichert.

Die hierzu verwendete ADF-Faces-Komponente *SelectOrderShuttle* besitzt keine Logik zur Datenbindung außer dem Laden der linken Liste. Die gesamte oben genannte Funktionalität musste daher komplett über eigenen Code realisiert werden. Da für die Kommunikation zwischen den Schichten des ADF-Frameworks die proprietären und außerodentlich schlecht dokumentierten ADF-APIs verwendet werden mussten, war dies die schwerste und langwierigste Aufgabe des gesamten Projektes.

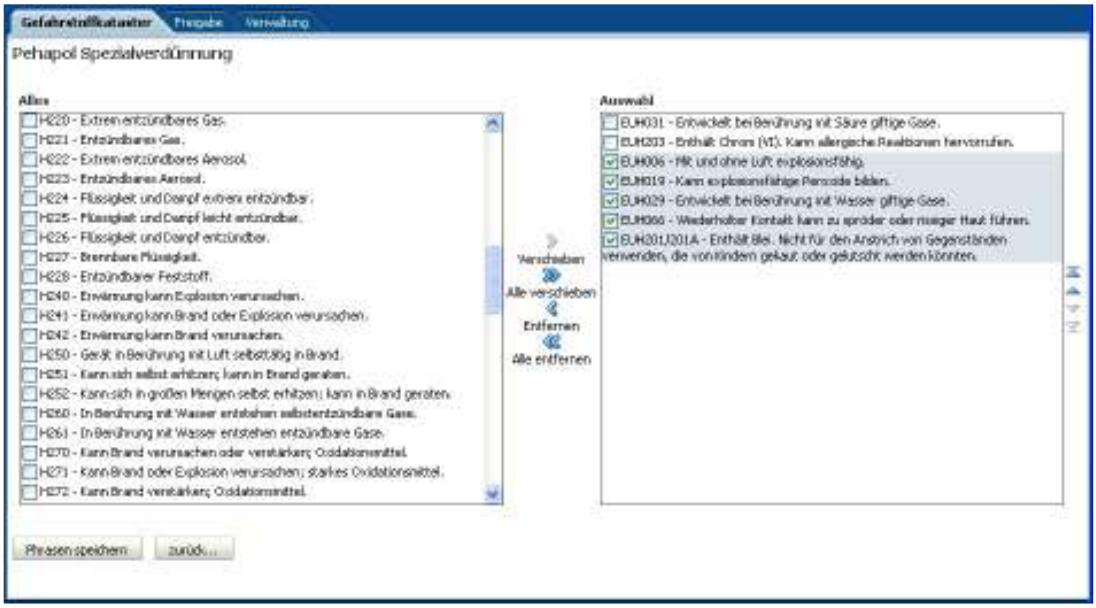

Abb. 27: Zuweisung von Warnhinweisen

Sicherheitsdatenblätter zu Stoffen ergänzen:

Zu jedem Stoff können mit Bemerkungen versehene Dateien hochgeladen werden. Sie werden unter Verwendung der ADF-APIs von der View an das Application Module durchgereicht und von dann als BLOB in der Datenbank gespeichert. Diese Funktionalität musste in jeder Schicht eigens implementiert werden.

Der Download einer Datei ist durch Anklicken eines Links möglich. Dabei werden die Schichten des Frameworks in der umgekehrten Reihenfolge durchlaufen und die Datei schließlich an den Browser gesendet.

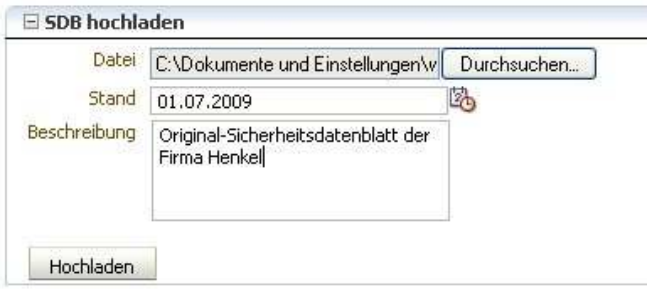

Abb. 28: Datei-Upload

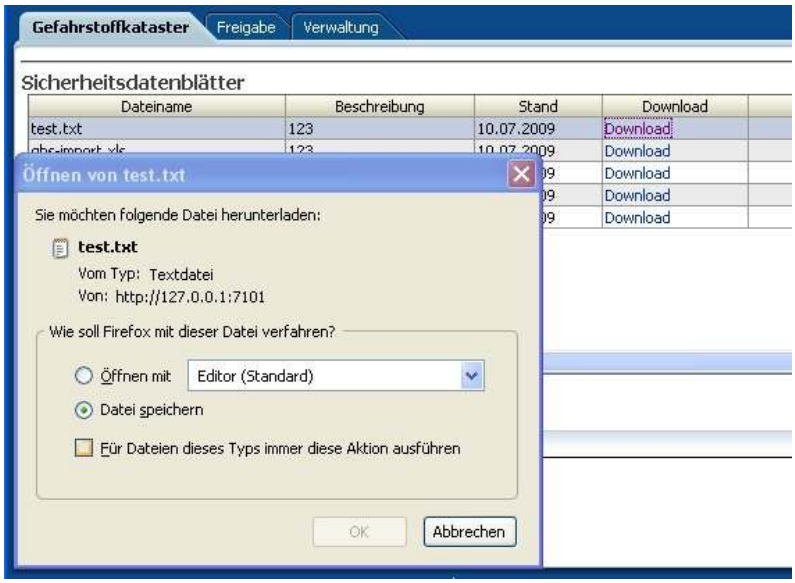

Abb. 29: Datei-Download

Arbeitsplätze des Unternehmens verwalten:

Arbeitsplätze können angelegt und bestehende Einträge können bearbeitet werden. Die Arbeitsplätze können sowohl generisch als auch speziell befüllt werden. Für die hier eingerichteten Arbeitsplätze können von Mitarbeitern später Materialfreigaben beantragt werden.

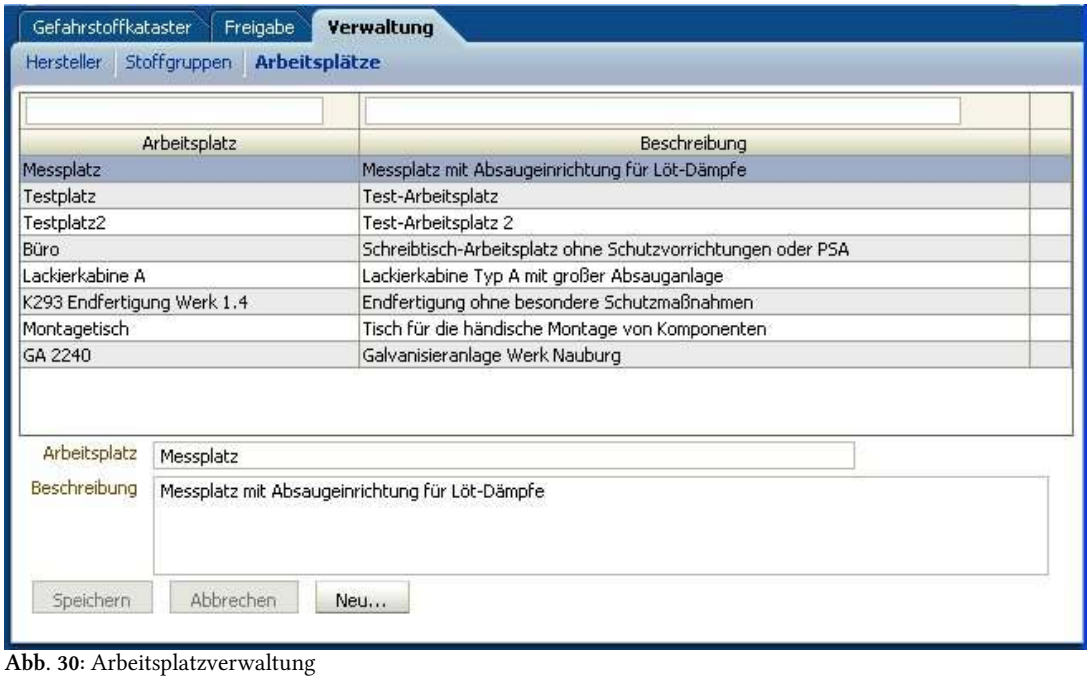

Abb. 30: Arbeitsplatzverwaltung

Freigabe für Gefahrstoff an Arbeitsplatz beantragen:

Benutzer können für einen Stoff und einen Arbeitsplatz zu einem bestimmten Zweck eine Materialfreigabe beantragen. Der Status aller Anträge wird ihnen auf einer Übersichtsseite dargestellt.

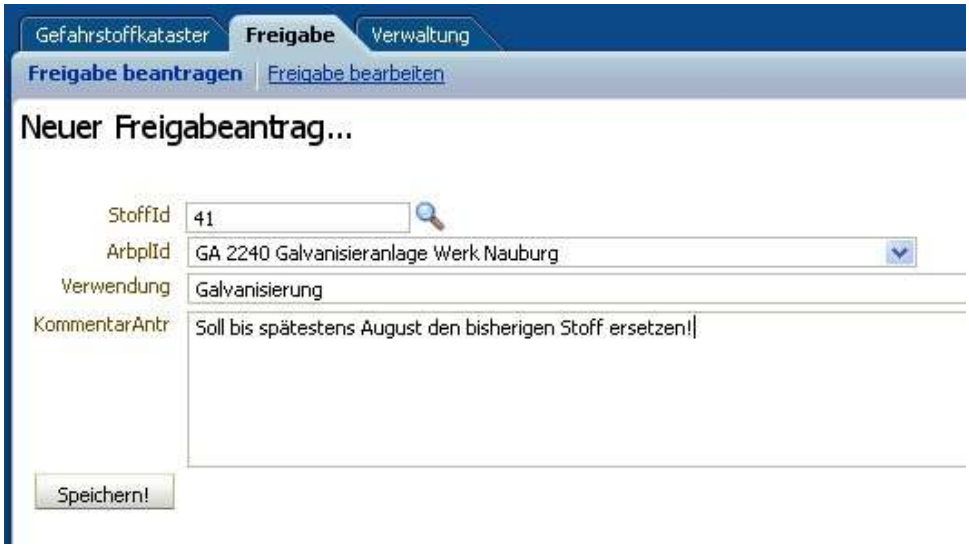

Abb. 31: Freigabeantrag stellen

Freigabeanträge bearbeiten:

Offene Freigabeanträge können schließlich kommentiert und daraufhin genehmigt oder abgelehnt werden.

Abb. 32: Freigabeantrag bearbeiten

Betriebsanweisung (Word-Dokumente) zu Freigaben ergänzen:

Es war vorgesehen, zu jeder genehmigten Freigabe eines Stoffes an einem Arbeitsplatz eine Betriebsanweisung zu ergänzen. Diese Funktionalität konnte aus Zeitmangel nicht mehr realisiert werden.

6.5 Auswertung

Bei der Betrachtung des Ergebnisses kann festgestellt werden: Es wurde tatsächlich ein benutzbarer Prototyp erstellt. Der an Scrum angelehnten Vorgehensweise ist es zu verdanken, dass bei Abweichungen von der Sprintplanung eine Neubewertung der Prioritäten erfolgte. Zwei Backlog Items wurden während des Projektes auf später verschoben und letztendlich nicht mehr implementiert. Die zur Verfügung stehende Zeit konnte so für die Implementierung von Items mit einem besseren Verhältnis von Geschäftswert zu Aufwand verwendet werden.

Allerdings muss auch angemerkt werden, dass in agilen Projekten unbedingt alle vorgesehenen Rollen durch verschiedene Personen besetzt werden müssen. Die in diesem Projekt durch die Zusammenlegung von Scrum Master, Product Owner und Team entstandenen Interessenkonflikte verursachten eine hohe Belastung und bedeuteten eine große Gefahr für das Projekt. Insbesondere die Einhaltung der Timeboxen und die Priorisierung der Backlog Items dürfen auf keinen Fall von Interessen des Teams beeinflusst werden. Qualitätsrichtlinien sollten zudem stets vor Projektbeginn schriftlich fixiert und später anhand dieser Vereinbarung kontrolliert werden, damit sich für das Team auch in zeitlich knappen Situationen keine Möglichkeit zur Verringerung der Qualität bietet.

Im Laufe des Projektes hat sich das Technologiewissen des Teams als wichtiger Qualitätsfaktor herausgestellt. Vor Projektbeginn absolvierte Tutorials bestätigten den von Oracle suggerierten Eindruck, dass mit dieser Technologie sehr schnell komplexe datenbankbasierte Webanwendungen erstellt werden können. Doch gute Java-Kenntnisse sind eben nicht gleichbedeutend mit Detailwissen zu ADF-APIs und praktischer Erfahrung mit JDeveloper. Während der Entwicklung zeigte sich nämlich, dass das Framework und die Oberflächenkomponenten schnell an ihre Grenzen geraten und das Schreiben von eigenem Code unter Verwendung der ADF APIs notwendig wird. Aufgrund des Charakters von ADF als Meta-Framework sind diese APIs jedoch sehr generisch gehalten, der Lernaufwand ist ausgesprochen hoch. Die APIs sind inkompatibel zur Vorgängerversion des Frameworks, und zu allem Überfluss gab es zur aktuellen Version 11g noch keine Literatur außer der spärlichen Dokumentation im Internet. Durch diese Unsicherheiten waren die Sprints im Vorhinein ausgesprochen schlecht planbar.

Ein weiterer Grund für unvorhergesehene Verzögerungen waren verschiedene Bugs in der Entwicklungsumgebung JDeveloper. Beispielsweise erhöhte sich der Speicherbedarf von JDeveloper mit jedem Deployment der Webanwendung, immer nach etwa 10 Deployments stürzte der integrierte Application Server schließlich ab und musste etwa zwei Minuten lang neu gestartet werden. Es handelte sich dabei laut Oracle um einen bekannten Bug. Ein weiteres Beispiel: Bei der Anwendung der Refactoring-Funktion zur Umbenennung einer Konfigurationsdatei überschrieb JDeveloper ohne Rückfrage eine interne, vor dem Benutzer versteckte Datei mit gleichem Namen. In der Folge war die Entwicklungsumgebung bis zur Wiederherstellung einer alten Codeversion nicht mehr benutzbar.

Wirklich problematisch war, dass bei Fragen zur Technologie kein kompetenter Ansprechpartner zur Verfügung stand. Mit einem besseren Technologiewissen über ADF und JDeveloper hätte sich der Zeitbedarf für die Entwicklung sehr wahrscheinlich verringern lassen können, auf jeden Fall wäre aber zumindest im Sprint Planning eine fundiertere Einschätzung des Aufwands und des Zeitbedarfs möglich gewesen.

Als positiv ist die Integration von JUnit in JDeveloper zu bewerten. Unit-Tests für eigene Klassen konnten damit problemlos ausgeführt werden (siehe **Abbildung 33**).

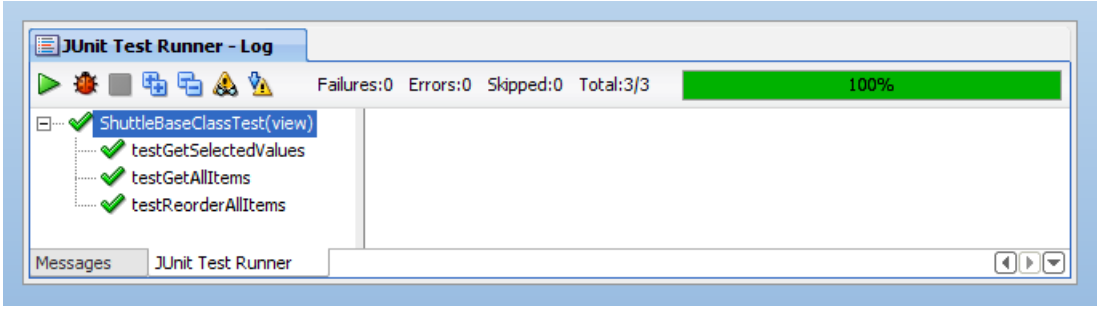

Abb. 33: JUnit-Integration

Allerdings offenbarte sich im Projekt auch ein großer „blinder Fleck“, der mit Tests innerhalb von JDeveloper nicht abgedeckt werden konnte: Das Framework ADF erzeugt einen großen Teil der letztendlichen Funktionalität aus einer Vielzahl von XML-Konfigurationsdateien. Jede Änderung an einer Konfigurationsdatei kann Auswirkungen auf die korrekte Funktion des Programms haben. Eine solche Auswirkung kann mit Unit-Tests allerdings nicht festgestellt werden. Gerade in einem Projekt mit mehreren Entwicklern könnte es zu gegenseitigen Beeinflussungen kommen, die möglicherweise erst spät entdeckt werden.

Die Verwendung des Versionsverwaltungssystems Subversion (SVN) mit JDeveloper funktionierte problemlos. Alle Änderungen an Geschäftsschicht, Model, Controller und View wurden versioniert. Das Datenbankschema, das sich im Laufe des Projektes natürlich auch entwickelte, konnte jedoch nicht in die Versionsverwaltung aufgenommen werden. Da dies jedoch grundsätzlich sinnvoll erscheint, wird vorgeschlagen, das Datenbankschema und Testdaten mittels Java-Code zu erstellen und diesen Code in die Versionsverwaltung aufzunehmen.

7 Gesamtbewertung / Ausblick

Es ist natürlich schade, dass im Rahmen dieser Arbeit kein echter Test des qualitätsorientierten agilen Prozesses stattfinden konnte. Die Auswertung des praktischen Projekts lässt aber dennoch den Schluss zu, dass der Prozess die aufgetretenen Qualitätsprobleme minimiert hätte. Sehr wichtig waren die Erfahrungen, dass die Rollen Scrum Master, Team und Product Owner durch verschiedene Personen besetzt werden müssen und dass Qualitätsrichtlinien vor dem Projekt schriftlich fixiert werden müssen.

Die Technologie Oracle ADF/JDeveloper hat nach diesem Projekt deutlich an Marketingglanz verloren. Sie präsentierte sich zwar als prinzipiell geeignet für die agile Softwareentwicklung, offenbarte jedoch auch einige Schwächen und ernste Probleme. Auf jeden Fall wurde deutlich, dass ein Einsatz dieser Technologie bei GIGATRONIK einen immensen Schulungsaufwand für das Personal bedeuten würde.

Für welche Art von Projekten agile Methoden grundsätzlich geeignet sind, wurde in Kapitel 3.3 erörtert, indem Stärken und Grenzen aufgezeigt wurden.

Im Zentrum dieser Arbeit standen jedoch die Herausarbeitung von Qualitätsfaktoren der agilen Softwareentwicklung und die anschließende Entwicklung eines qualitätsorientierten agilen Prozesses. Zunächst wurden exemplarisch drei sehr unterschiedliche agile Methoden betrachtet. Obwohl diese Methoden drei verschiedene Ansätze verkörperten, war es ohne größere Probleme möglich, zu abstrahieren und allgemeingültige Qualitätsfaktoren zu bestimmen.

Als Ergebnis dieser theoretischen Überlegungen liegen nun eine konkrete Beschreibung des Prozesses und eine erste Version der dazugehörigen Checklisten vor. Für zukünftige Verbesserungen im Praxiseinsatz ist der Prozess dank der integrierten Prozessverbesserungsmechanismen bestens gerüstet.

Abbildungsverzeichnis

* Erstellt mit Grafiken aus dem Nuvola Icon Set von David Vignoni (Lizenziert unter der GNU Lesser General Public License, Version 2.1), Quelle: http://www.icon-king.com/files/nuvola-1.0.tar.gz

Tabellenverzeichnis

Literaturverzeichnis

Beck, Kent (2003): Extreme programming explained. Embrace change. 8th. print. Boston: Addison-Wesley.

Beck, Kent; Beedle, Mike; van Bennekum, Arie; Cockburn, Alistair; Cunningham, Ward; Fowler, Martin et al. (2001): Manifesto for Agile Software Development. Online verfügbar unter http://agilemanifesto.org/, zuletzt geprüft am 27.08.2009.

Boehm, Barry; Turner, Richard (2008): Balancing agility and discipline. A guide for the perplexed. 6. print. Boston, Mass.: Addison-Wesley.

Brodbeck, Felix C. (1994): Produktivität und Qualität in Software-Projekten. Psychologische Analyse und Optimierung von Arbeitsprozessen in der Software-Entwicklung. München: Oldenbourg.

Cockburn, Alistair; Giesecke, Jobst (2005): Crystal Clear. Agile Software-Entwicklung für kleine Teams; [Strategien, Rollen und Prozesse; typische Fehler erkennen und vermeiden; zahlreiche Praxisbeispiele und eine detaillierte Fallstudie]. 1. Aufl. Bonn: mitp-Verl.

Dietl, Wilhelm (2003): Das Informationssystem der deutschen Polizei. In: Hirschmann, Kai; Leggemann, Christian (Hg.): Der Kampf gegen den Terrorismus. Strategien und Handlungserfordernisse in Deutschland. Berlin: BWV Berliner Wiss.-Verl., S. 191–200.

Dumke, Reiner R.; Lother, Mathias; Wille, Cornelius; Zbrog, Fritz (2003): Web Engineering. München: Pearson-Studium.

Gernert, Christiane (2003): Agiles Projektmanagement. Risikogesteuerte Softwareentwicklung. München: Hanser.

GIGATRONIK GmbH (Juni 2007): GIGATRONIK Entwicklungsprozess (GT-EP). Projekthandbuch.

GIGATRONIK GmbH (September 2008): Qualitätsmanagement-Handbuch.

Gloger, Boris (2008): Scrum. Produkte zuverlässig und schnell entwickeln. München: Hanser.

Grässlin, Jürgen (2005): Das Daimler-Desaster. Vom Vorzeigekonzern zum Sanierungsfall? München: Droemer.

Hoffmann, Dirk W. (2008): Software-Qualität. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg.

Hörmann, Klaus (2006): SPICE in der Praxis. Interpretationshilfe für Anwender und Assessoren ; basierend auf ISO/IEC 15504 (Stand 2006). 1. Aufl. Heidelberg: dpunkt-Verl.

Hunt, Andrew; Thomas, Dave (2004): Unit-Tests mit JUnit. München: Hanser (Pragmatisch programmieren).

Kneuper, Ralf (2006): CMMI. Verbesserung von Softwareprozessen mit Capability Maturity Model Integration. 2., überarb. und erw. Aufl. Heidelberg: dpunkt-Verl.

Koletzke, Peter; Mills, Duncan (2007): Oracle JDeveloper 10g for forms & PL/SQL developers. A guide to Web development with Oracle ADF; covers Oracle JDeveloper 10g (10.1.3). New York, NY: McGraw-Hill.

Liebhart, Daniel (2007): Architecture Blueprints. Ein Leitfaden zur Konstruktion von Softwaresystemen mit Java Spring, .NET, ADF, Forms und SOA. 2., aktual. und erw. Aufl. München: Hanser.

Moe, Nils Brede; Stålhane, Tor; Dingsøyr, Torgeir (Hg.) (Juli 2009): Improve Newsletter 2/2009. Online verfügbar unter http://www.sintef.no/improve/downloads/2009/Improve_2_2009.pdf, zuletzt geprüft am 26.07.2009.

Naur, Peter; Randell, Brian (Hg.): SOFTWARE ENGINEERING. Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968. Online verfügbar unter http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF, zuletzt geprüft am 11.08.2009.

Oracle Corporation (2009): Introduction to Building Fusion Web Applications with Oracle ADF. Online verfügbar unter http://download.oracle.com/docs/cd/E12839_01/web.1111/b31974/intro.htm, zuletzt geprüft am 25.09.2009.

Pichler, Roman (2008): Scrum - agiles Projektmanagement erfolgreich einsetzen. 1. Aufl., korrigierter Nachdr. Heidelberg: dpunkt-Verl.

Schneider, Kurt (2007): Abenteuer Softwarequalität. Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. 1. Aufl. Heidelberg: dpunkt-Verl.

Schwaber, Ken (2007): Agiles Projektmanagement mit Scrum. Unterschleißheim: Microsoft Press.

Sommerville, Ian (2007): Software engineering. 8., aktualisierte Aufl. München: Pearson Studium.

Stephen C. Johnson (1994): Objecting to Objects. In: USENIX Association (Hg.): Proceedings of the USENIX Winter 1994 Technical Conference. San Francisco .

Sutherland, Jeff; Jakobsen, Carsten Ruseng; Johnson, Kent (2007): Scrum and CMMI Level 5: The Magic Potion for Code Warriors. Herausgegeben von IEEE Computer Society. (AGILE 2007). Online verfügbar unter http://www.agile2007.org/agile2007/downloads/proceedings/057_Scrum%20and%20CMMI%20Level%205_425.pdf, zuletzt geprüft am 25.08.2009.

Takeuchi, Hirotaka; Nonaka, Ikujiro (1986): The New New Product Development Game. In: Harvard Business Review (Reprint 86116). Online verfügbar unter http://apln-richmond.pbworks.com/f/New+New+Prod+Devel+Game.pdf, zuletzt geprüft am 24.08.2009.

Wallmüller, Ernest (2001): Software-Qualitätsmanagement in der Praxis. Software-Qualität durch Führung und Verbesserung von Software-Prozessen. 2., völlig überarb. Aufl. München: Hanser.

Anhang

1. Projekthandbuch zum GIGATRONIK-Entwicklungsprozess (24 Seiten)

2. Gefahrstoffmanagement-Prototyp (auf CD-ROM)