entwickler.press

Agile Developer Skills

Effektives Arbeiten in einem Scrum-Team

Christoph Mathis und Andreas Wintersteiger

Christoph Mathis, Andreas Wintersteiger

Agile Developer Skills

Effektives Arbeiten in einem Scrum-Team

entwickler.press

Christoph Mathis, Andreas Wintersteiger
Agile Developer Skills
ISBN: 978-3-86802-247-6

Ein Imprint der Software & Support Media GmbH

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Geleitsstr. 14
60599 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 930089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Sebastian Burkart
Korrektorat: Robert Lippert, Büro für Projektleitung, Redaktion und Events
Satz: Pobporn Fischer
Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

1 Einführung

Im Projekt steht das Entwicklerteam im Zentrum: es schafft wertvolle Software in einem transparenten, empirischen Prozess.

Das Buch verfolgt das Anliegen, die Fähigkeiten und Kenntnisse zu beschreiben, die ein Teammitglied in einem agilen Projekt braucht. Diese gehen, wie wir sehen werden, weit über Programmierkenntnisse hinaus.

Unter Entwickler verstehen wir nicht nur die Teammitglieder, die Code produzieren, sondern alle, die an der Umsetzung von Anforderungen in neue Funktionalität mitarbeiten, also

- Programmierer
- Tester
- Architekten
- Dokumentierer

Am Anfang einer Entwicklung stehen sauberer Code und gute agile Engineering-Praktiken. Die Mitglieder des Teams dürfen und müssen darüber hinaus ihre Beziehungen untereinander und zu Außenstehenden gestalten. Erst als Team und nicht nur als Einzelne können sie wirklich produktiv werden, sich effektiv weiterentwickeln und den optimalen Beitrag zum Projekterfolg liefern.

Im Buch wollen wir die verschiedenen Aspekte abdecken und gliedern unsere Beschreibung in drei Teile:

- Einen Methodenteil, in dem wir begründen, warum die agile Entwicklung zur Hauptströmung geworden ist und beschreiben die wichtigen Methoden: Scrum, XP, Lean und Kanban.
- Einen Agile-Engineering-Teil, in dem wir die Techniken zur Softwareerstellung darstellen.
- Einen Anwendungsteil, in dem wir die praktische Umsetzung in der täglichen Arbeit, in Zusammenarbeit mit dem Product Owner und in der Organisation beschreiben.

Wir stellen uns explizit auf den Standpunkt eines Teammitglieds und versuchen, die Anforderungen an seine Qualifikation und Bedürfnisse herauszuarbeiten. Das kann Scrum Mastern, agilen Coaches und auch Managern helfen, einen schärferen Fokus darauf zu bekommen, was beim Aufstellen eines neuen Teams und bei der Betreuung wichtig ist. Es kann auch jedem Teammitglied helfen, die Arbeit seines Teams einzuschätzen und Ideen für Verbesserungsvorschläge zu erarbeiten.

1.1 Wissen, Techniken und Skills

Wir beschreiben damit Themen auf verschiedenen Ebenen: Wissen, Techniken und Skills. Mit Wissen meinen wir einen Überblick über ein Themengebiet. Wissen braucht man als Referenz, es muss aber nicht notwendig unmittelbar eine Grundlage für das Handeln sein. Wenn wir Techniken beschreiben, dann wollen wir auch die Richtung aufzeigen, wie sie praktisch angewendet werden können. Techniken, die man beherrscht, werden Skills. Dazu ist zusätzlich auch die richtige Einstellung nötig, das dauernde Suchen nach Verbesserung. Dieses Suchen ist eine der wesentlichen Voraussetzungen, zu einem agilen Entwickler zu werden. Wenn es uns gelingt, diese Aspekte für uns zu kombinieren, dann ist das der Weg zu dem, was wir persönliche Exzellenz bezeichnen: die Beherrschung meiner Arbeit und meine kontinuierliche Verbesserung.

1.2 Agile Engineering Skills

Seit Anfang 2009 hat sich eine lebhafte Diskussion in der Scrum-Community über die Qualifikation von Teammitgliedern und gute Engineering-Praktiken entwickelt. Angestoßen wurde diese Diskussion durch einen Blog-Eintrag von Martin Fowler über „Flaccid Scrum“ [3]. Darin beklagt er, dass es immer wieder Scrum-Teams gibt, die trotz guten Scrum-Prozesses keine gute Software abliefern. Er nennt das „flaccid“, d.h. schlappes Scrum. Daraus hat sich eine produktive Diskussion entwickelt. Die Scrum Alliance hat aus dieser Diskussion heraus einen Katalog von grundlegenden Entwicklerqualifikationen destilliert. Diese umfassen:

- Agile Architektur: Softwarearchitektur in einer agilen Umgebung ist emergent, sie wächst organisch, ohne dabei ihre konzeptionelle Integrität einzubüßen, und ihre Struktur ist selten endgültig.
- Testgetriebene Entwicklung (TDD): Sie baut darauf auf, dass man den Test vor dem Code schreibt. Das wirkt zunächst künstlich – warum ist die Reihenfolge wichtig – aber geht weit darüber hinaus: der Test wird nicht nur zuerst geschrieben, sondern bestimmt auch, wie die Geschäftslogik implementiert wird.
- Refactoring: TDD beinhaltet als untrennbaren Bestandteil den Refactoring-Schritt. Wir kennen das als Kent Becks zwei Hüte: im ersten Schritt wird die minimale Funktionalität implementiert, die der Test verlangt. Dann wird im Refactoring-Schritt der Code verbessert, restrukturiert, ohne dass neue Funktionalität entsteht.
- Versionsverwaltung: Sie ermöglicht es, einen Zustand gefahrlos wiederherzustellen und ist ein unverzichtbares Sicherheitsnetz für eine berechenbare agile Entwicklung.
- Kontinuierliche Integration: Nach jeder Änderung wird das System neu gebaut und automatisch getestet. Damit ergibt sich ein kurzer Feedback-Zyklus über den wahren Stand an korrekter, getesteter und integrierter Software.

- Pair Programming: Dies ist eine Technik, bei der zwei Entwickler gemeinsam vor einem Bildschirm entwickeln. Pair Programming hilft bei der Herstellung von qualitativ hochwertiger Software und bei der schnellen Ausbreitung von Wissen in einem Team.

1.3 Skills zur Zusammenarbeit

In der agilen Entwicklung arbeitet ein Teammitglied mehr mit anderen zusammen als in der konventionellen Entwicklung. Dies umfasst

- Zusammenarbeit als ein Team
- Einbeziehen des Kunden in den Prozess
- Mitarbeit an Features des Produkts

Damit ergibt sich neben einer effektiveren Arbeit auch mehr Transparenz für alle Beteiligten. Es ergeben sich neue und höhere Anforderungen an die Fähigkeit zur Zusammenarbeit und spezifische notwendige Skills wie:

- Aktives Zuhören: Aktives Zuhören ist eine Fähigkeit, sich auf die Kommunikation mit einer anderen Person zu fokussieren – verbal und nichtverbal.
- Effektives Fragen: Die Fähigkeit, gute Fragen effektiv zu formulieren hilft uns, neues Wissen in unser mentales Modell der Welt zu integrieren – oder auch, unser Modell zu verändern.
- Logisch Argumentieren: Beim Präsentieren einer Idee oder einer Meinung ist es wichtig, sie logisch abzuleiten, um die Essenz zu finden und abzusichern.
- Respektieren der Kollegen: Respekt heißt, einen Schritt von den eigenen Überzeugungen zurückzutreten und zuzugestehen, dass die Meinung des anderen ebenfalls legitim ist.
- Hilfe anbieten: Hilfe anbieten und dann auch zu leisten, gehören zusammen.
- Teilen und mitteilen: Teilen heißt, wir teilen unser Wissen und Kenntnisse oder wir teilen Ressourcen. Im Team ist Teilen darauf gerichtet, dass es dem Team hilft, die gemeinsamen Ziele zu erreichen.
- Partizipieren und teilnehmen

1.4 Skills zur erfolgreichen Arbeit in der Organisation

Unter dieser Kategorie fassen wir einen weiten Bogen von Skills zusammen:

- Lerntechniken
- Allgemeine Arbeitstechniken
- Techniken zur Arbeit in der Firma und zur Organisation von Communities

1.5 Sieben Leitlinien für agile Entwickler

Es ist immer problematisch, einen so komplizierten Sachverhalt auf ein paar Schlagworte zu reduzieren. Es hilft aber vielleicht, den richtigen Mindset zu setzen. Die folgenden konkreten Punkte sind uns in unseren Softwareprojekten immer wieder untergekommen:

- Entwickle das richtige System
- Beweise es kontinuierlich
- Liefere zielgerichtet neue Funktionalität
- Liefere bei jedem Sprint
- Trage zur richtigen Architektur und zum richtigen Design bei
- Arbeite kooperativ
- Bilde dich weiter und hilf anderen zu lernen

Entwickle das richtige System

Ein alter Kalauer sagt, das Zweitwichtigste sei, etwas richtig zu machen. Das Wichtigste ist, das Richtige zu machen.

Das hat eine ganz reale Anwendung in der Softwareentwicklung: jeder Schritt in der Entwicklung muss von den Anforderungen her motiviert sein. Viel zu oft werden spekulative Verallgemeinerungen oder gar spekulative Features realisiert, für die es letztlich gar keinen „Business Case“, keine geschäftliche Motivation gibt. Das ist einer weit verbreiteten Lücke in der Kette der Übergabe von Anforderungen geschuldet: Kunden spezifizieren Anforderungen in ihrer Welt, ihrer Begrifflichkeit und Granularität. Das reicht oft nicht aus, um die Realisierung hinreichend genau zu leiten.

Was hilft, ist eine Diskussion der Anforderungen. Dabei entsteht ein gegenseitiges Verständnis und evtl. verändern sich dabei auch die Anforderungen: Arbeite mit dem Product Owner am Verständnis. Nutze User Stories als Werkzeug und achte darauf, dass das gemeinsame Verständnis durch Akzeptanztests gefestigt wird.

Übrigens: Technische Anforderungen können oft nur vom Entwicklerteam erkannt oder sinnvoll formuliert werden. Dann muss ich das auch tun und so dazu beitragen, dass das richtige und das beste System entwickelt wird.

Beweise es kontinuierlich

Oft gehen Akzeptanzkriterien direkt in die Formulierung von Akzeptanztests ein, die damit die Rolle einer ausführbaren Spezifikation bekommen. Das bietet eine riesige Chance, kooperativ alle Fähigkeiten im Team einzubinden. Insbesondere die Rolle der Tester verändert sich massiv, sie stehen nicht am Ende eines Miniwasserfalls, sondern sind mitten in der Entwicklung dabei. Manchmal übernehmen sie teilweise die Rolle eines Analysten, der Geschäftszusammenhänge für die Umsetzung in Software aufbereitet.

Am besten funktioniert das mit Werkzeugen, die es erlauben, automatisch ablaufende Testskripte schon vor der dazugehörigen Realisierung zu schreiben.

Das heißt: Setze Akzeptanzkriterien direkt und nachvollziehbar in Tests um.

Liefere zielgerichtet neue Funktionalität

Die Zielsetzung heißt: gemeinsam wandelt das Team effektiv und zuverlässig Anforderungen in neue Funktionalität um. Die Mittel der Wahl, die den Stand der Kunst repräsentieren, heißen testgetriebene Entwicklung und Pair Programming.

Für eine testgetriebene Entwicklung gibt es ausgezeichnete Gründe:

- Jede Zeile Code ist durch einen Test, d.h. ein Stück ausführbare Spezifikation begründet. Das stellt sicher, dass keine „goldenen Wasserhähne" eingebaut werden, die niemand braucht.
- Die Software ist unendlich viel flexibler. Das wird allein schon dadurch erreicht, dass man bei der Entwicklung jedes Stück Code auf Verbesserungsmöglichkeiten abklopft: das Refactoring bei jedem Schritt.
- Ich hänge nicht mehr fest: Durch die Aufteilung der Entwicklung in kleine, reversible Einzelschritte kann ich jederzeit einen Schritt zurückgehen und neu aufsetzen – im Gegensatz zum Fall, in dem ich mit dem Debugger einen Fehler in der Arbeit von mehreren Tagen suchen muss.

Es gibt keine Möglichkeit, Wissen und Fähigkeiten in einem Team schneller zu verbreiten als mit Pair Programming.

- Neue Techniken lassen sich ohne Risiko im laufenden Projekt erarbeiten.
- Gefährliche Wissensmonopole werde aufgelöst oder verhindert, das Team erhöht seine Handlungsfähigkeit.

Das heißt: Um zielgerichtet und zuverlässig liefern zu können, arbeite testgetrieben und mit Pair Programming.

Liefere nach jedem Sprint

Ein Sprint, nach dem keine fertige Software ausgeliefert wird, wird zum puren Ritual. Seine Funktion zum Inspizieren des Entwicklungsstands, die Gelegenheit zum Nachsteuern, die genaue Aussage über Stärken und Defizite geht verloren. Die Motivation und das Interesse gehen verloren, das Commitment wird wertlos.

Das Team wird in einem negativen Feedback-Kreis gefangen: die Menge der unfertigen Arbeiten nimmt zu, man kann sich nicht auf Arbeitsergebnisse verlassen, die Entwicklungsgeschwindigkeit sinkt.

Deshalb: Plane realistisch, beende angefangene Arbeiten und liefere nach jedem Sprint getestete Software.

Trage zur richtigen Architektur und zum richtigen Design bei

Zwei Aspekte einer guten Softwarelösung sind von außen nur schwer zu beurteilen:

- Die innere Qualität, das heißt, die Einhaltung von Strukturierungsprinzipien
- Manche nichtfunktionalen Eigenschaften der Software

Manche der nichtfunktionalen Eigenschaften kann man durch Qualitätsmetriken erfassen – das sollte man auch tun. Andere erschließen sich nicht so leicht: wenn über die Zeit viele Änderungen stattgefunden haben, stellt sich manchmal die Frage, ob die Gesamtstruktur der Software noch adäquat ist oder ob tiefgreifende Änderungen durchgeführt werden müssen, um die Entwicklung zu unterstützen – agile Architektur ist emergent, d.h. die Strukturen entwickeln sich tendenziell weiter.

Um das zu erkennen und ggf. zu thematisieren, muss man einen Blick über den Tellerrand wagen, sich mit dem Gesamtbild beschäftigen und nicht nur mit dem aktuellen Arbeitsschritt. Man muss auch einen Blick dafür entwickeln und gezielt weiterpflegen. Mit anderen Worten: arbeite daran, ein Systemarchitekt zu werden.

Arbeite kooperativ

Softwareentwicklung ist eine Teamarbeit. Wir arbeiten gemeinsam an einem Resultat und wir lernen voneinander. Von Alistair Cockburn stammt die Gleichung „Entwicklungstempo ist gleich Lerntempo“. Das funktioniert nur dann, wenn ich kooperativ im Team arbeite und wenn wir voneinander lernen.

Kooperation geht aber über das Team hinaus. Ich muss mit dem Product Owner und meinen Kunden zusammenarbeiten, ich muss ihre Probleme und Ziele verstehen, um den besten Beitrag zur Lösung zu erbringen.

In keinem Fall funktioniert diese Kooperation, wenn ich mein Gegenüber nicht respektiere und versuche, es zu verstehen.

Das heißt dann vollständig: arbeite kooperativ im Team, mit dem Product Owner und deinen Kunden, respektiere deine Kollegen.

Bilde dich weiter und hilf anderen zu lernen

Bob Martin startete eine Diskussion auf der Konferenz Agile 2008 mit einem Beitrag „Software Craftsmanship over Crap“[1]. Das bringt eine Diskussion auf den Punkt, in der deutlich wird: ich bin für die Resultate meiner Arbeit verantwortlich. Gute Software entsteht nur, wenn ich selbst dazu motiviert bin – Außenmotivation funktioniert nicht. Ich kann nur dann auf der Höhe der Zeit bleiben, wenn ich selbst die Initiative zu kontinuierlicher Weiterbildung ergreife. Und der Stand unseres Handwerks (meinetwegen: unserer Kunst) wird sich nur dann signifikant verbessern, wenn wir selbst daran arbeiten.

1 TODO: Quelle

2 Agile Grundlagen

Agile Strukturen und Entwicklungen haben sich durchgesetzt, weil sie nachhaltig einige Vorteile für Unternehmen und Entwickler bieten. Das ist eine Win-Win-Situation, und wenn wir diese nutzen wollen, um unsere Kollegen und unsere Firma von agilen Arbeitsweisen zu überzeugen, sollten wir die Vorteile für beide Seiten parat haben.

In diesem Kapitel wollen wir über diese Vorteile reden und die allgemeinen Hintergründe etwas ausleuchten.

2.1 Agiler Nutzen für Entwickler

Wer in einem agilen Team gearbeitet hat, hat schon nach kurzer Zeit viele Vorteile durch diese Vorgehensweise erfahren:

- Effektiv und zuverlässig liefern: Mit einer klar abgestimmten Aufgabenstellung können wir souverän den Arbeitsfortschritt steuern und wissen, dass wir das richtige Problem lösen.
- Transparenz schaffen: Wir arbeiten in nachhaltiger Geschwindigkeit, ohne unnötigen Stress und können Erwartungen frühzeitig mit der Realität zusammenführen.
- Selbstorganisiertes Arbeiten im Team: Entscheidungen über die Arbeitsorganisation fällen wir gemeinsam und direkt im Team, ohne Flaschenhals und Verzögerungen.
- Kontinuierliche Verbesserung der eigenen Arbeitsweise: Im Team decken wir Möglichkeiten zur Optimierung der Arbeit auf und setzen sie sofort um.
- Maximal lernen: Durch Zusammenarbeit helfen wir uns, auf dem Stand der Technik zu bleiben und neue Techniken zu erproben und zu praktizieren.
- Mehr Spaß: Wir sind Entwickler, weil wir komplizierte Probleme lösen wollen; eine agile Umgebung räumt unnötige Hindernisse beiseite und gibt uns die beste Möglichkeit dazu.

2.2 Agiler Nutzen für Unternehmen

Agile Strukturen und Entwicklungen haben sich durchgesetzt, weil sie nachhaltig einige Vorteile für Unternehmen bieten:[1]

- Verbessern des ROI[2] durch den Fokus auf kontinuierlichen Fluss der Wertschöpfung
- Verlässliche Lieferung durch Einbeziehen der Kunden in häufige Interaktionen und gemeinsame Verantwortung
- Managen von Unsicherheit und Veränderung durch kontinuierliches Inspizieren und Adaptieren
- Freisetzen von Kreativität durch die Anerkennung der Tatsache, dass Individuen die ultimative Quelle der Wertschöpfung darstellen, und schaffen einer Umgebung, in der sie einen Unterschied machen können
- Erhöhen der Performance durch Gruppenverantwortlichkeit für die Ergebnisse und geteilte Verantwortung für die Effektivität des Teams
- Verbessern der Effektivität durch adaptierte Strategien, Prozesse und Praktiken

Das bedeutet, dass agile Prinzipien in den verschiedensten Kontexten auftreten und Nutzen stiften können. Wir interessieren uns in erster Linie für den Einsatz in Softwareprojekten, aber auch hier hat Agil eine Menge verschiedener Facetten.

Es lohnt sich aber, wenn wir zunächst über ein paar allgemeine Prinzipien von Agil reden.

1 Nach der „Declaration of Interdependence“, siehe *http://pmdoi.org/*

2 Return on Invest oder wirtschaftlicher Ertrag aus einer Investition

2.3 Woher kommt Agil

In den 1980ern hieß der wichtigste Trend in der Softwareprojektorganisation „Software-Engineering“. Man hatte mehr und mehr damit zu kämpfen, dass die Softwareentwicklungen immer größer und komplexer wurden und dass die erzielten Ergebnisse immer weniger vorhersehbar waren.

Die bevorzugte Lösung war es, einen Prozess zu definieren, nach dem alle Projektbeteiligten vorgehen sollten. Zuerst waren das Handbücher, diese wurden bald durch Software unterstützt, die eine Einhaltung dieser Prozesse unterstützen und überwachen sollte. Diese Prozesse wurden mit der Zeit sehr umfangreich, ohne dass die Ergebnisse wirklich überzeugend waren: sie wurden nur teilweise eingehalten und oft hatte man den Eindruck, dass ein Ergebnis trotz und nicht wegen des Prozesses erzielt wurde.

Diese Prozesse waren alle sehr reich an Vorschriften und stark an langfristiger Planbarkeit und weniger an flexibler Anpassung ausgerichtet.[3] Gleichzeitig gab es in vielfältigen Formen den Trend, dass sich Dinge weniger planen ließen als zuvor: Globalisierung, mehr Konkurrenz, mehr Entwicklungen in kleinen Firmen (das waren die Jahre des frühen Siegeszugs des PC) machten es ständig dringender, sich kontinuierlich auf neue Veränderungen einzustellen.

Als Reaktion darauf entwickelte sich Anfang bis Mitte der 1990er Jahre eine Gegenbewegung, die auf weniger statt mehr Festlegungen setzte, die „Light Weight Processes“, die leichtgewichtigen Prozesse. Diese Methoden waren Scrum, Extreme Programming, Crystal und andere.

Mehrere dieser Methodenentwickler trafen sich im Januar 2001 und verabschiedeten eine gemeinsame Sicht auf die Methoden und die Prinzipien dahinter. Sie nannten dieses Papier (frei von Bescheidenheit) das Agile Manifest. Dieses Manifest hat inzwischen eine zentrale Rolle in der Diskussion der grundlegenden Prinzipien der Softwareentwicklung gewonnen.

Agile Vorgehensweise bietet sich insbesondere für komplexe Probleme an, bei denen nicht alle Parameter zu Beginn eines Vorhabens hinreichend bekannt sind. Dann spielt auch agile Softwareentwicklung ihre Stärke aus. Wichtig ist es auch, das Konzept von Agil umfassend zu verstehen, sonst funktioniert es nicht:

- Agile Softwareentwicklung ist nicht nur eine Menge von Praktiken. Sie basiert auf dem Agilen Manifest, dessen Werten und Prinzipien. Diese zu verstehen ist essenziell.
- Die Einführung agiler Softwareentwicklung erfordert eine Änderung der Einstellung. Dazu ist es oft nötig, die Unternehmenskultur zu verändern.

3 Um nur die (in unserer Region) einflussreichsten zu nennen: das V-Modell [VMo2010] und Rational Unified Method, RUP [Jac1999]

2.4 Das Agile Manifest

Das „Agile Manifest", das im Jahr 2001 von 17 prominenten Personen aus der Softwareentwicklung geschrieben wurde, beschreibt folgende grundlegenden Werte und Prinzipien:

Agile Werte

Die wesentlichen Werte des Agilen Manifests sind

Abbildung 2.1: Werte im Manifest

Im Agilen Manifest sind folgende Prinzipien als grundlegend aufgeführt:

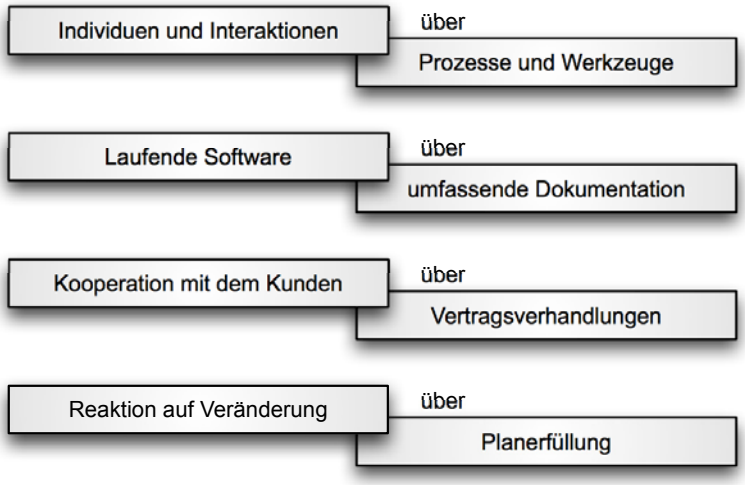

Abbildung 2.2: Prinzipien im Manifest

Ausführlicher heißt es:

- Unsere erste Priorität ist es, den Kunden durch frühe und kontinuierliche Lieferung werthaltiger Software zufriedenzustellen.
- Wir begrüßen veränderte Anforderungen, auch in einer späten Entwicklungsphase. Agile Prozesse nutzen Veränderungen als Konkurrenzvorteil.

- Häufige Auslieferung laufender Software alle paar Wochen bis alle paar Monate, mit einer Präferenz zu kürzeren Intervallen.
- Anwender und Entwickler arbeiten das ganze Projekt über täglich zusammen.
- Projekte werden um motivierte Personen aufgebaut. Gib ihnen die nötige Umgebung und Unterstützung und vertraue ihnen, ihre Arbeit zu machen.
- Die effizienteste und effektivste Methode zur Übermittlung von Informationen mit und innerhalb eines Entwicklungsteams ist die direkte Kommunikation (face-to-face).
- Laufende Software ist das primäre Maß für den Arbeitsfortschritt.
- Agile Prozesse fördern nachhaltige Entwicklung. Sponsoren, Entwickler und Anwender sollten kontinuierlich ein konstantes Entwicklungstempo beibehalten.
- Kontinuierliche Aufmerksamkeit für technische Exzellenz und gutes Design stützt agile Entwicklung.
- Einfachheit – die Kunst, unnötige Arbeit zu vermeiden – ist essenziell.
- Die besten Architekturen, Anforderungen und Designs entwickeln sich in selbstorganisierenden Teams.
- Das Team reflektiert in regelmäßigen Abständen über die Verbesserung seiner Arbeitsweise, verbessert sie und passt sie an.

Agile Aktivitäten und Praktiken

Auf dieser Grundlage bauen eine Reihe von Aktivitäten und Praktiken auf, u.a.:

- Agiles Schätzen
- Testgetriebene Entwicklung
- Kontinuierliche Integration
- Kollektives Eigentum an Code
- Inkrementelles Design
- Pair Programming
- Nachhaltige Arbeitsgeschwindigkeit
- Enge Beteiligung des Kunden im Projektverlauf

Diese Prinzipien sind zunächst abstrakt – wir sind aber erst bei den grundlegenden Elementen. In den folgenden Kapiteln werden wir konkreter zeigen, wie eine Umsetzung aussehen kann. Vielleicht lohnt es sich, wenn ihr danach noch einmal hierher zurückkommt und diese Einordnung aus einer anderen Perspektive betrachtet.

2.5 Wann kann man Agil einsetzen

Die Nutzung Agiler Prinzipien und Vorgehensweisen ist an die geeigneten Bedingungen gebunden. Dazu zählen

- Die Aufgabenstellung und die Umgebung. Wir nennen das den Arbeitskontext.
- Die Art und Weise, wie die Beteiligten ihre Zusammenarbeit organisieren. Das umfasst Teamarbeit, Management, Kultur und vieles mehr.

Wir starten hier mit einer Beschreibung verschiedener Arbeitskontexte, die exemplarisch deutlich machen, in welchen sich Agiles Arbeiten aufdrängt.

In einem Artikel im Harvard Business Review hat 2007 Snowden[4] verschiedene Typen von Arbeitskontexten untersucht. Er unterscheidet dabei einfache, komplizierte und komplexe Kontexte.

Einfache Arbeitskontexte

Einfache (Arbeits-)Kontexte können durch eine überschaubare Menge von Anweisungen strukturiert werden. Ein Beispiel dafür ist die Zubereitung von Hamburger in einer Hamburger-Filiale: es gibt ein Handbuch, in dem die genauen Anweisungen zum Braten eines Hamburgers hinterlegt sind. Die Mitarbeiter des Hamburger Imbiss haben keinen Einfluss auf die Inhalte des Handbuchs, sie sollen lediglich die Anweisungen ausführen. Es gibt eine Menge ähnlicher Arbeitskontexte auch in der Softwareentwicklung, auf die diese einfache Struktur anwendbar ist.

- Die Schadensbearbeitung einer Versicherung: Jeder Kunde wird darauf bestehen, dass sein Schaden nach nachvollziehbaren Regeln erstattet wird – das kann aber nur dann funktionieren, wenn es dafür klar definierte Regeln gibt, an die sich jeder Sachbearbeiter hält.
- Arbeiten von Behörden: Nachvollziehbare und zuverlässige Regeln sind einer der Grundpfeiler unserer Gesellschaft.

Weil hier Anweisungen im Wesentlichen in eine Richtung fließen, bieten agile Organisationsprinzipien wenig Vorteile.

Komplizierte Arbeitskontexte

Komplizierte Kontexte können nicht mehr durch einen einfachen Satz von Anweisungen erfasst werden. Oft ist eine lange Ausbildung, ein tiefes Verständnis der Materie oder sogar ein ganzes Team nötig, um komplizierte Probleme zu lösen: Die Verkabelung eines Flugzeugs ist kompliziert. Es dauert lange herauszufinden, wie alles verbunden ist. Aber wenn man sie lange genug untersucht, kann man mit (annähernder) Sicherheit sagen, was jeder Schaltkreis bewirkt und wie man ihn steuert.

4 [Sno2007]

Abbildung 2.3: Kompliziert

Das System ist letztlich verstehbar. Wenn es wichtig ist, es zu verstehen, lohnt sich der Aufwand es zu studieren und ein detailliertes Diagramm zu erstellen. Und für die Aufgabenstellung dabei gibt es letztlich eine beste Lösung. Zusammengefasst kann man sagen, kompliziert bedeutet: nicht einfach, aber letztlich beherrschbar.

Ein Teamleiter ist hier gut beraten, wenn er vor einer Entscheidung den Rat der Teammitglieder einholt: er weiß nicht alles und könnte verhängnisvolle Fehler machen. Daher ist das angemessene Leitungssystem nicht „Command and Control", sondern ein kooperativer Stil, im dem das Wissen und die Meinungen der Teammitglieder gehört werden.

Komplexe Systeme

In unserem Beispiel für „komplexe Systeme" sitzen Besatzung und Passagiere im Flugzeug und versuchen vorherzusagen, was während des Flugs passieren wird. Plötzlich wird aus „kompliziert" *komplex*. Man kann das Leben all dieser Menschen über Jahre studieren ohne vorhersagen zu können, wie sie interagieren. Man kann Vermutungen anstellen, aber man ist nie sicher. Kein noch so hoher Aufwand kann je diese Sicherheit geben. Es gibt einfach zu viele Variablen.

Komplex heißt also: nicht einfach und nicht vollkommen verstehbar. Es gibt keine Garantie dafür, dass eine Aktion, die heute richtig ist, auch morgen zu gleichen Ergebnissen führt. Und es heißt auch, dass man nie für alle Eventualitäten vorausplanen kann, sondern dass ständig von allen Beteiligten Entscheidungen gefordert sind.

Es kann also kein Handbuch geben, das zu befolgen ist wie in einfachen Kontexten. Die Situation ist, nun ja, komplex.

Abbildung 2.4: Komplex

Es reicht auch nicht aus, wenn die Teammitglieder den Leiter beraten und dieser dann über die beste Vorgehensweise entscheidet. Das würde ihn sofort zu einem hoffnungslosen Flaschenhals für Informationen und Entscheidungen machen. Jedes Teammitglied muss stattdessen die Initiative ergreifen, den anderen nach Bedarf helfen und ständig Entscheidungen treffen. Mit anderen Worten: in komplexen Situationen brauchen wir ein selbstorganisiertes Team.

Definierte und empirische Prozesse

Die bekannteste Form eines Prozesses ist dadurch charakterisiert, dass man versucht, alle Eventualitäten vorab zu beschreiben und bei der Ausführung so wenig wie möglich Spielraum übrig zu lassen. Wir reden hier von definierten Prozessen. Das funktioniert bei einfachen Kontexten ganz gut und bei komplizierten noch einigermaßen, versagt aber völlig bei komplexen Situationen.

Das heißt nicht, dass wir in komplexen Situationen ohne Prozess arbeiten müssen. In einem empirischen Prozess plant man auch zu Beginn, weiß aber von vornherein, dass dieser Plan nicht vollständig oder unveränderlich ist. Man legt ein Hauptaugenmerk darauf, den Ablauf ständig nachzusteuern. Statt also den vergeblichen Versuch zu unternehmen, alles im Vorhinein festzuzurren, steuert man ständig während des Prozessverlaufs nach.

Die wesentlichen Unterschiede der beiden Prozesstypen liegen in der Vorhersagbarkeit bzw. Nichtvorhersagbarkeit des Planungsraums.

Definierte Prozesse:

- Wir kennen alle Anfangsbedingungen
- Wir können jede Handlung präzise beschreiben
- Kann kompliziert sein, ist letztlich aber im Voraus planbar
- Planungsfokus: Vorabplanung

- Leitung: „Command and Control“ oder kooperative Führung
- Beispiele sind das klassische Fließband oder eine klassische Sachbearbeitung wie eine Rentenbewilligung

Empirische Prozesse:

- Umgebung und Anfangsbedingungen sind nicht vollständig definiert
- Anforderungen können sich über die Zeit ändern
- Das Wissen über das beste Vorgehen ist unvollständig
- Planungsfokus: „inspect and adapt“
- Selbstorganisierte Teams funktionieren am besten
- Beispiele sind die Passagierbetreuung während eines Flugs oder Autofahren im dichten Verkehr

2.6 Softwareentwicklung ist empirisch

Seit vielen Jahren hat es eine Menge Bestrebungen gegeben, die „handwerkliche“ und „unkontrollierte“ Softwareentwicklung unter Kontrolle zu bringen. Dabei stand das Paradigma der „Industrialisierung“ im Vordergrund, mit dem in anderen Industrien bisher als unerreichbar erscheinende Produktivitätssteigerungen erreicht worden waren. Die wesentlichen Elemente davon sind:

- Eine Festlegung des Prozesses „von außen“, d.h. die einzelnen Arbeitsschritte werden vor der Ausführung festgelegt – in der Regel von einer anderen Person als den Ausführenden.
- Aufteilung in viele kleine Einzelschritte, die jeder für sich gut messbar und kontrollierbar sind.

Um diese Ziel in der Softwareentwicklung zu erreichen, unternahm man große Anstrengungen, standardisierte Prozesse zu entwerfen und zum Teil mit Toolunterstützung durchzusetzen. Beispiele hierfür sind das V-Modell, Prince2 und andere. Dieser Ansatz hat in der Softwareentwicklung allerdings nie wirklich gut funktioniert – so war in den 1980er Jahren viel die Rede von der „Softwarekrise“.

So lange man in einem relativ stabilen Umfeld überschaubare Batch- und Reporting-Systeme schrieb, konnte man aber einigermaßen über die Runden kommen.

Mit den heutigen Softwaresystemen ist man damit endgültig an eine Grenze gestoßen und man kommt nicht mehr umhin anzuerkennen, dass Softwareentwicklung (fast immer) ein empirischer Prozess ist:

- Heutige Softwaresysteme sind wesentlich umfangreicher.
- Die Umgebung und Voraussetzungen sind nicht vollständig definiert.

- Anforderungen ändern sich über die Zeit.
- Das Wissen über die beste Vorgehensweise ist unvollständig.
- Das System ist komplex.

In der Konsequenz hängt der Erfolg eines Softwaresystems von folgenden Prämissen ab:

- Frühes und häufiges Feedback zum entstehenden System zu bekommen, sowohl zu Anforderungen als auch zur Lösung – z.B. durch frühzeitiges und häufiges Demonstrieren der laufenden Teillösungen („running increments“).
- Alle Beteiligten können problemlos auf Änderungen und neue Erkenntnisse reagieren und die Anforderung und die Lösung anpassen.
- Effektive Kommunikation zwischen allen an der Entwicklung Beteiligten, sodass das geschäftliche und technische Wissen für jeden verfügbar ist, der es benötigt. Oft ist das effektivste Mittel zur Kommunikation das gesprochene Wort.

Agile Prinzipien basieren auf diesen Einsichten. Daher eignet sich agile Entwicklung für den allergrößten Teil der Softwareentwicklung.

2.7 Empirische Prozesse und Selbstorganisation

Es lohnt sich, noch einen zweiten Blick auf die Prinzipien der Selbstorganisation zu werfen. Selbstorganisation der Teams ist eine unabdingbare Voraussetzung für eine agile Organisation.

In der agilen Welt wird „Command and Control“ schon fast als Schimpfwort benutzt – trotzdem sollten wir uns ansehen, wo genau die Konflikte zu agiler Entwicklung liegen.

In einer Umgebung, in der man schnell auf Änderungen von Anforderungen oder von technischen Rahmenbedingungen reagieren muss, braucht man:

- Schnelle Reaktionen auf neue Einflüsse. Neue Situationen können jederzeit entstehen – vom Auftreten eines Fehlers bis zu einer dringenden neuen Anforderung, die so im Plan nicht vorgesehen war. Wenn man eine Organisationsstruktur hat, in der zuerst der Plan angepasst oder der Vorgesetzte konsultiert werden muss, führt das zu signifikant längeren Reaktionszeiten. Ebenso kommt es fast zwangsläufig zu Engpässen und langen Wartezeiten, wenn man keine Flexibilität in der Verteilung der Aufgaben hat. Alle agilen Methoden sehen daher vor, dass Teams sich ihre Arbeit selbst organisieren – es ist ein Kernpunkt des Begriffs agil.
- Einen geschützten Bereich im Sinn von Zeitautonomie. Manche Aufgaben erfordern einfach Zeit und Konzentration. Wenn man dauernd unterbrochen wird, kann man komplexere Aufgaben nicht erfolgreich bewältigen.
- Freiwilliges Mitmachen und Initiative. Komplexe Aufgaben kann man nur dann gut bewältigen, wenn man auch Spaß daran hat (wenn die Primärmotivation stimmt). Das

kann nicht durch kurzfristige finanzielle Anreize oder sogar durch Druck ersetzt werden. Im Gegenteil – empirische Untersuchungen zeigen, dass die Leistung bei kreativen Aufgaben durch Belohnungen und Druck schlechter wird.

- Kreativität aller Beteiligten. Die Struktur der Probleme kann sich schnell ändern. Wenn die Beteiligten dann darauf bestehen, weiter in eingefahrenen Gleisen zu bleiben, ist der Misserfolg schon fast vorprogrammiert. Kreativität braucht ebenfalls einen geschützten Zeitbereich. Zusätzlich muss auch ein Klima von Vertrauen und freundlicher/wohlwollender Zusammenarbeit herrschen, in der man auch riskante und unerprobte Dinge ausprobieren kann.
- Erfolgreiches Lernen erfolgt im Team. Das hat zum einen den Grund, dass man dann ermutigt wird, Dinge aus dem eigenen Kopf zu verbalisieren, d.h. oft auch erst, sie explizit zu machen. Und es bedeutet, dass man sich gegenseitig weiterhelfen kann und neue Aspekte zeigt.

Selbstorganisation ist hervorragend für die Bedingungen in der Softwareentwicklung geeignet. Deshalb sind selbstorganisierte Teams auch eine der tragenden Säulen agiler Softwareentwicklung.

2.8 Feedback und Demings PDCA-Zyklus

Feedback ist eines der zentralen Konzepte beim Steuern empirischer Prozesse. Wenn man nicht die vollständigen Informationen zu Beginn einer Entwicklung hat, muss man eben nachsteuern – auf der Basis von mehr Informationen, die man im Lauf der Zeit bekommt.

Scrum als Projektmanagement-Framework kennt verschiedene Ebenen von Feedback: auf Tagesbasis den Daily Scrum, auf Sprintbasis das Review-Meeting und die Retrospektiven. In den agilen Softwaretechniken sind noch spezifischere Feedback-Mechanismen verborgen, z.B. Unit-Testing und kontinuierliche Integration.

Allerdings fehlt hier noch etwas Wesentliches. Für Feedback wird immer wieder der Regelkreis eines Thermostats als Beispiel angeführt: wenn es zu kalt wird, wird mehr warmes Wasser zugeführt, wenn es zu warm wird entsprechend weniger. Das Thermostat „weiß“, was die richtige Temperatur ist. Wenn wir diese Analogie auf ein Scrum-Team anwenden, müssen wir uns fragen, was denn in diesem Zusammenhang die „richtige“ Stellgröße ist – und das ist im Einzelfall gar nicht so leicht zu beantworten.

Double Loop Learning

Wenn die richtigen Ziele vorgegeben sind, kann man die Thermostat-Metapher anwenden. Interessant wird es aber dann, wenn diese Ziele hinterfragt werden: dann entsteht faktisch ein zweiter Feedback-Zyklus, das „System“ optimiert sich oder – im realen Leben: die Organisation lernt als Ganzes dazu und verbessert ihre Arbeitsweise. Chris Ar-

gyris [Arg1976] nennt diesen Prozess „double loop learning“[5]. Er setzt voraus, dass sich die Beteiligten für die Ziele interessieren und über ihren Tellerrand hinaussehen, und das erzeugt ein ungeheures Potenzial für Verbesserungen.

Deming-Zyklus

William Edwards Deming[6] machte die Systematik der kontinuierlichen Verbesserung eines Prozesses bekannt. Ihn hat interessiert, wie die Innovationen ihren Weg aus einem Team in die Organisation insgesamt finden und er hat dazu den Feedback-Zyklus erweitert. Dieser PDCA- (oder Deming-)Zyklus besteht aus vier Elementen:

- *Plan:* Eine Prozessverbesserung wird geplant – Verbesserungspotenziale werden identifiziert, der Ist-Zustand wird bestimmt und der Soll-Zustand wird entwickelt, eine Maßnahme wird identifiziert.
- *Do:* Die Verbesserung wird ausprobiert und praktisch optimiert.
- *Check:* Der getestete neue Prozess wird sorgfältig geprüft und gegebenenfalls freigegeben.
- *Act:* Der freigegebene Prozess wird eingeführt und ausgerollt und gegebenenfalls festgeschrieben.

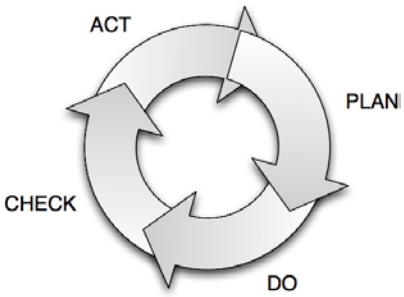

Abbildung 2.5: PDCA-Zyklus

Gerade der vierte Schritt macht den Unterschied: er sorgt dafür, dass die wichtigen Innovationen in der ganzen Firma Verbreitung finden, dass diese einzelnen Verbesserungen zu einem kohärenten kontinuierlichen Verbesserungsprozess werden können.

Wir wenden diesen kontinuierlichen Verbesserungsprozess sowohl im Team als auch für die Weitergabe neuer Ideen zwischen den Teams und Organisationseinheiten an. Bei Deming ist hauptsächlich das mittlere Management für diese Weitergabe zuständig. Wir

5 Argyris, C. (1976): Single-loop and double-loop models in research on decision making. In: Administrative Science Quarterly, Vol. 21, S. 363–375. Das ist schon ziemlich schwerer Stoff. Eine schöne Zusammenfassung findet man unter: *http://www.infed.org/thinkers/argyris.htm*

6 W. E. Deming: Out of the Crisis. MIT Press 1982

denken, das sollte nicht darauf beschränkt bleiben: wir schlagen die Nutzung von „communities of practice“ vor, wie sie in Kapitel 22 beschrieben werden.

Die Umgebung für empirische Prozesssteuerung

Wie wir sehen, erfordert die Lösung komplexer Probleme und die weitere Optimierung von Organisationen konstante Anpassungen und benötigt

- Kreativität
- Initiative
- Individuen und Teams, die lernen

Damit das gedeiht, ist eine Kultur gefordert, die die richtigen Werte schätzt:

- Vertrauen statt Schuldzuweisungen, bei Fehlern das Wahrnehmen einer Gelegenheit zum Lernen.
- Respekt: Menschen sind keine Ressourcen.

Dies ist zugegebenermaßen ein weiter Bogen, er ist aber notwendig, um deutlich zu machen, das Agil und Scrum nicht nur eine Sammlung von Techniken sind und nicht wirklich funktionieren werden, wenn man sie darauf reduziert. Die Vorteile kommen erst dann voll zum Tragen, wenn auch die entsprechenden Werte gelebt werden.

2.9 Und die Methoden?

Die bekannten agilen Methoden haben verschiedene Schwerpunkte: zum Teil konzentrieren sie sich auf technische Prinzipien und zum Teil auf Prinzipien des Projektmanagements. Wir denken, dass man als Teammitglied einen soliden Einblick in diese Methoden haben muss, um zu beeinflussen, wie man sich in dem Umfeld bewegt und wie man sie nutzt. Wir beschreiben die Methoden, die wir nutzen, im Detail.

2.9.1 Scrum

Scrum ist Mitte der 1990er von Ken Schwaber und Jeff Sutherland entwickelt worden und hat sich ursprünglich nur um Aspekte des Projektmanagements, genauer: des Prozesses auf der Ebene eines Teams, gekümmert. Es ist daher prinzipiell breiter einsetzbar als nur in der Softwareentwicklung, hat aber dort seine größte Verbreitung.

Scrum-Teams in der Softwareentwicklung haben sich in aller Regel die technischen Prinzipien von Extreme Programming zu Eigen gemacht. Scrum-Teams, die darauf verzichtet haben, hatten in aller Regel ein wesentlich härteres Leben, wenn sie zuverlässig und effektiv Produktinkremente abliefern wollten.

In letzter Zeit werden die wichtigsten technischen Praktiken direkt in Scrum integriert betrachtet. Und es gibt parallel dazu einen erweiterten Diskurs in der Scrum-Community über die Organisation von agiler Entwicklung im Großen.

Scrum ist die am meisten eingesetzte der agilen Methoden und wir widmen ihm daher einen relativ breiten Raum.

2.9.2 Extreme Programming – XP

Extreme Programming wurde von Kent Beck, Ward Cunningham und Ron Jeffries während ihrer Arbeit an einem Projekt bei Chrysler entwickelt. Es entstand zwischen 1995 und 2000 und hat um 2000 eine atemberaubende Adoptionskurve, einen richtigen Hype.

XP nennt 12 Praktiken, meist Programmierpraktiken und kennt ähnliche Rollen wie Scrum. Die Praktiken wurden inzwischen breiter übernommen und die meisten bilden inzwischen ein Art Grundkonsens für agile Softwareentwicklung. Wir referenzieren sie einfach als Agile-Engineering-Techniken, um dieser Verbreitung Rechnung zu tragen.

2.9.3 Lean Software Development

Lean Software Development wurde von Mary und Tom Poppendieck direkt aus der Übernahme der Prinzipien des Lean Thinking in die Softwareentwicklung übernommen.

Diese Methode bietet wertvolle Einblicke in Lean als eine Wurzel der agilen Denkweise, und man entdeckt hier viele der allgemeinen Aspekte von „agil“ wieder und kann sie direkt ihren Grundlagen zuordnen.

Lean Software Development ist für uns auch eine Brücke, mit der man genauer die Einordnung der Kanban-Methodik vornehmen und sie besser verstehen kann.

2.9.4 Kanban

Kanban wurde von David Anderson aus der Theory of Constraints entwickelt und verwendet kanban, ein Werkzeug aus der Lean Production als sein Markenzeichen. Kanban ist das japanische Wort für eine Signalkarte, mit der ein Arbeitsfortschritt verfolgt wird.

Kanban hat in letzter Zeit ein hohes Maß an Aufmerksamkeit bekommen und die Nutzer schwören darauf, wie einfach und mit wie wenig Hindernissen Kanban eingeführt werden kann. Kanban beschreibt eine Möglichkeit, wie ein Team seine tägliche Arbeit organisieren kann und ist damit eine wertvolle Ergänzung zu den anderen agilen Methoden. Diese Arbeitsorganisation

- kann im Kontext eines Sprints erfolgen
- kann Aufgaben beschreiben, die wegen externer Abhängigkeiten schwer im Rahmen eines Sprints abzubilden sind
- kann die Bearbeitung wiederkehrender Aufgaben, wie z.B. Wartungstätigkeiten und Bugfixes, transparenter machen.

Manche Teams mit vorwiegend wiederkehrenden Aufgabenstellungen verzichten ganz auf die Planung von Iterationen und setzen ganz auf Kanban.

2.9.5 Weitere Methoden

Crystal

Die Methodenfamilie Crystal wurde von Alistair Cockburn entwickelt und wird von ihm weiterentwickelt. Bei Crystal gibt es verschiedene Ausprägungen für kleine und größere Projekte, für Projekte in stark regulierten Umgebungen etc.

Alistair Cockburn hat zwar große Verdienste in der agilen Community und Diskussion, seiner Crystal-Methode ist aber eine größere Wirkung versagt geblieben – und er ist mit der Methodenfamilie nie wirklich fertig geworden

DSDM

DSDM „Dynamic Systems Development Method“ hat mehr Strukturierungen als die meisten anderen Methoden.

DSDM ist vor allem in Großbritannien verbreitet.

Übrigens: die Moscow-Methode[7], die manche Scrum-Teams für die Darstellung ihres Team Commitments benutzen, kommt ursprünglich aus DSDM.

Agile RUP

Agile RUP ist als agile Methode aus RUP hervorgegangen. RUP (Rational Unified Process) hat zwar iterative Elemente eingeführt, verharrt aber in der Denkweise eines Wasserfallprozesses (auch wenn diese Behauptung bei einigen der Befürworter strittig ist). Agile RUP macht hier weitere Anpassungen in Richtung Agil, behält aber trotzdem für unseren Geschmack zu viel Ballast aus seiner Herkunft.

7 *http://de.wikipedia.org/wiki/MoSCoW-Priorisierung*

Literatur

- Manifesto for Agile Software Development. *http://agilemanifesto.org/*
- [Arg1976] Argyris, C. (1976): Single-loop and double-loop models in research on decision making. In: Administrative Science Quarterly, Vol. 21, S. 363–375.
- [Dem1982] W. Edwards Deming: Out of the Crisis. MIT Press 1982
- [Jac1999] Jacobson, Booch, Rumbaugh: The unified software development process, Addison-Wesley, 1999

  Inzwischen hat sich auch der RUP weiterentwickelt: eine stärkere Betonung der iterativen Entwicklung und verschiedene Ableger, die Gedanken aus den agilen Methoden übernehmen.
- [Sno2007] David Snowden and Mary Boone: A Leader's Framework for Decision Making, Harvard Business Review 2007
- [VMo2010] Entwicklungsstandard für IT-Systeme des Bundes (V-Modell).

  In verschiedenen Versionen veröffentlicht. Die letzte Version „XT" beinhaltet auch eine (u. E. recht halbherzige und verzerrte) Option zur „agilen delivery", d.h. im Team zu arbeiten: *http://download.4soft.de/v-modell-xt-bund/releases/1.0/V-Modell-XT-Bund-Gesamt.pdf*

3 Scrum-Konzepte

Wir stellen in diesem Kapitel den Methodenkern von Scrum vor. Zusammen mit der Beschreibung von Lean und Kanban in Kapitel 4 legen wir damit eine gemeinsame Grundlage für die Gedankenwelt und das Vokabular, mit denen wir unsere Arbeit im Team organisieren.

Im hinteren Teil des Buchs ab Kapitel 14 werden wir detaillierter auf agile Praktiken und typische Situationen im Projekt eingehen.

3.1 Warum Scrum?

Scrum[1] ist die am weitesten verbreitete agile Methode der Softwareentwicklung und umfasst einen minimalen Satz von Praktiken, um Software effizient zu entwickeln. Es ist die bewusst unvollständige, knappe Beschreibung einer Managementmethode, in der komplexe Aufgaben kreativ gelöst werden. Im Mittelpunkt von Scrum steht der Entwicklungsprozess, nicht etwa Programmiertechniken. Diese werden in Scrum aus Extreme Programming (XP) entlehnt. Die meisten erfolgreichen Software-Implementierungen verwenden eine Kombination von Scrum und XP und in letzter Zeit wandern agile Entwicklungstechniken in den Kern des Scrum-Diskurses. Scrum kann als Projektmanagement-Methodik ohne Engineering-Techniken auch in anderen Kontexten als der Softwareentwicklung verwendet werden, auch wenn wir uns hier auf Softwareprojekte beschränken.

Agile Methoden im Allgemeinen und Scrum im Besonderen brechen mit dem lange Zeit dominierenden Vorgehen, dass der Kunde die Anforderungen vor dem eigentlichen Projektstart detailliert beschreibt, sie dann als komplette Spezifikation an die Programmierer übergibt und daraus dann möglichst eins zu eins die Software entsteht, während Änderungen grundsätzlich als unerwünschte Störungen gelten. Diese Art der Projektabwicklung bezeichnen wir als Wasserfallprozess: eine Phase muss zuerst abgeschlossen werden, bevor man unwiderruflich in die nächste Phase eintritt.

Der Siegeszug von Agilen Methoden wie Scrum ist in erster Linie dadurch begründet, dass sie die Entwicklung komplexer Systeme mit hoher Qualität und Effektivität in einem Umfeld erlauben, in dem nicht alle Voraussetzungen bekannt und stabil sind. In Scrum akzeptieren wir, dass sich die Komplexität von Entwicklungsprozessen nicht im Voraus

1 Die Bezeichnung Scrum taucht zum ersten Mal in [Tak1986] auf. Scrum wurde zum ersten Mal ca. 1993 von Jeff Sutherland als Methode in der Software angewendet. Sie wurde dann in einem Vortrag auf der Konferenz „OOPSLA-95“ als Methode beschrieben [Sch1997], das erste Buch über Scrum erschien 2001 ([Sch2001]). Scrum ist keine Abkürzung, sondern basiert auf einem Spielzug im Rugby.

exakt in Phasen und Arbeitsschritte mit der Genauigkeit von Tagen oder sogar Stunden planen lässt. Statt dessen setzt Scrum auf einen iterativen Prozess, der die Forderungen des Agilen Manifests erfüllt. Erfahrungen aus zahllosen Projekten haben gezeigt, dass Scrum die Arbeit der Entwickler aufwertet und dadurch ihre Effektivität erhöht und auch dem Auftraggeber völlig neue Transparenz und Steuerungsmöglichkeiten bringt.

Kompakt zusammengefasst bedeutet Scrum:

- Der Auftraggeber erhält in kurzen Abständen ausführbare Software, die er testen kann.
- Die laufende Priorisierung der Anforderungen führt dazu, dass das Dringendste zuerst erledigt wird und der Auftraggeber auf jeden Fall die wichtigsten Features als funktionsfähige Software erhält.
- Die ständige Rückkopplung zwischen Auftraggeber und Team stellt eine hoch effiziente Form ständiger Qualitätsverbesserung dar und fördert die kontinuierliche Weiterentwicklung aller Beteiligten.
- Missverständnisse und Fehlentwicklungen werden durch den intensiven Kontakt zwischen Auftraggeber und Team während des Prozesses minimiert.
- Die ständige Integration und Präsentation der Softwareinkremente verhindern, dass Inkompatibilitäten und andere Schnittstellenprobleme in einer späten Projektphase zu Rückschlägen führen.

Durch die Reduktion auf die wesentlichen Grundtechniken ist Scrum, verglichen mit komplexen Projektmanagement-Methoden, schnell erlernbar. Bei Stress im Projekt fällt es erfahrungsgemäß relativ leicht, die Scrum-Prinzipien konsequent durchzuhalten.

Als Konsequenz daraus ist Scrum ein Framework, das bewusst Lücken lässt, die durch eigene Entscheidungen gefüllt werden dürfen und müssen. Für solche Situationen hat sich in zahlreichen bisherigen Projekten eine ganze Reihe von „Good Practices“ herauskristallisiert, und selbstverständlich lässt sich Scrum bei Bedarf durch andere etablierte Techniken erweitern. Die Praxis hat zudem gezeigt, dass der Satz von Werten und Prinzipien, den Scrum beschreibt, in vielen Fällen hilft, schnell eine eigene Lösung zu finden.

Im Folgenden werden wir die zentralen Konzepte – den Kern von Scrum – vorstellen. Dazu kommen Erfahrungen und Tipps, wie Softwareprojekte im konkreten Projekt mit Scrum umgehen und darauf aufbauend ihre eigene Vorgehensweise entwickeln.

3.2 Grenzen von Scrum

Scrum basiert auf einer Balance zwischen

- der Möglichkeit, nach jedem Sprint die Prioritäten anzupassen und
- dem geschützten Raum, den der Sprint für das Team bildet und in dem das Team auch Fehler machen, ausbügeln und Experimente machen kann.

Manche Rahmenbedingungen bieten keine Möglichkeit zur iterativen Planung bzw. machen ein iteratives Vorgehen obsolet:

- Wenn die Aufgaben sehr unvorhersehbar sind, zum Beispiel in einem reinen Wartungsteam.
- Wenn die Aufgaben ziemlich standardisiert sind und wenig Raum für kreative Bearbeitung lassen.

Wenn eine dieser Rahmenbedingungen vorliegt, sollte man sich überlegen, ob man statt Scrum lieber eine Methode wie Kanban[2] verwendet, die eher auf eine Optimierung der Durchgangszeit und eine effektive Bearbeitung von Routineaufgaben optimiert ist. Kanban und verschiedene Möglichkeiten, Scrum und Kanban zu kombinieren, beschreiben wir im folgenden Kapitel.

Auf der anderen Seite gibt es Aufgaben, die fast vollständig festgelegt sind, bei denen die technischen Rahmenbedingungen und Anforderungen vollständig klar sind und viele Erfahrungen im Team verfügbar sind. Das ist zwar, wenn man genau hinsieht, nicht sehr häufig der Fall, aber immerhin möglich. In diesem Fall funktionieren auch die traditionellen Planungsmethoden, vielleicht brauchen wir hier kein Scrum.

3.3 Wesentliche Scrum-Begriffe

In Scrum gibt es einige Begriffe, die im normalen Sprachgebrauch nicht verwendet werden oder mit etwas anderen Bedeutungen verbunden sind:

Der Product Owner

hat die Rolle des Auftraggebers bzw. Kunden und trägt die Hauptverantwortung für den geschäftlichen Erfolg des zu entwickelnden Produkts. Im Scrum-Prozess ist er oder sie während der gesamten Projektdauer präsent und u.a. für die Erstellung und Pflege einer priorisierten Liste offener Features, des Product Backlog zuständig.

Der Scrum Master

ist der Hauptverantwortliche für die Implementierung des Scrum-Prozesses, d.h. der Organisation der Zusammenarbeit mit dem Mitteln von Scrum und die Einhaltung der dazugehörigen Prinzipien. Er oder sie ist Moderator und Coach, beseitigt Hindernisse und stellt sicher, dass das Team effektiv arbeiten kann.

Das Team oder auch Umsetzungsteam

ist nach Scrum das Team der Programmierer, Systemarchitekten, Tester und allen anderen beteiligten Softwarespezialisten und trägt die Hauptverantwortung für die technische

2 Siehe Kapitel 4

Umsetzung der Anforderungen. Dazu gehören die selbstorganisierte Entwicklungsarbeit während des Sprints und die Beratung des Product Owner.

Das Scrum-Team

Product Owner, Scrum Master und Team haben ein gemeinsames Interesse an einem Projekterfolg – in ihren jeweiligen Rollen. Sie bilden damit auch so etwas wie ein Team. Um die beiden Teambegriffe abzugrenzen, nennt man diese alle zusammen das Scrum-Team.

Der Sprint

ist eine meist zwei- bis vierwöchige Entwicklungsphase, in der das Team selbstorganisiert an der Umsetzung der für diesen Zeitraum festgelegten Anforderungen arbeitet und dabei potenziell auslieferbare Produktinkremente erstellt.

Das Product Backlog

ist eine priorisierte, dynamische Anforderungsliste für ein Projekt. Alle Anforderungen an das Umsetzungsteam stehen im Product Backlog. Die obersten Einträge, die in nächster Zeit umgesetzt werden, sind detailliert, die späteren können noch allgemeiner formuliert sein und werden kontinuierlich konkretisiert, wie sie nach oben wandern. Es ist Eigentum des Product Owner.

Das Sprint Backlog

ist eine priorisierte Liste aller Aufgaben zur Umsetzung der Features, die für den aktuellen Sprint ausgewählt wurden – es enthält die Aktivitäten, die für den aktuellen Sprint geplant sind. Es ist Eigentum des Teams.

Das Sprint Burn-Down Chart

ist die täglich aktualisierte visuelle Darstellung der Aufgaben, die das Team während des aktuellen Sprints noch zu erledigen hat, und ist ebenfalls Eigentum des Teams.

Das Impediment Backlog

ist eine Liste mit Hindernissen, die das Entwicklungsteam bei der Arbeit stören. Es wird vom Scrum Master geführt, der dafür Sorge trägt, diese Hindernisse möglichst schnell zu beseitigen und dies mit dem Impediment Backlog transparent macht.

Das Daily Scrum Meeting

ist ein (maximal 15-minütiges) Zusammentreffen, in dem sich die Teammitglieder während eines Sprints täglich synchronisieren.

Das Sprintplanungs-Meeting

vor jedem Sprint ist ein Zusammentreffen, in dem das Team entscheidet, wie viele der am höchsten priorisierten, vom Product Owner definierten Features es im nächsten Sprint fertig-