Der **APM-Guide** zu:

Uwe Vigenschow

APM – Agiles Projektmanagement

dpunkt.verlag

Der APM-Guide
zu »APM – Agiles Projektmanagement«

Uwe Vigenschow ist Abteilungsleiter bei Werum IT Solutions. Er ist seit über 25 Jahren in der Softwareentwicklung als Entwickler, Berater, Projektleiter und Führungskraft tätig. Mit agilen Konzepten befasst er sich seit Ende der 1990er-Jahre und hat APM, Scrum und XP bei verschiedenen Firmen eingeführt und an besondere Rahmenbedingungen angepasst.

Uwe Vigenschow ist Autor von »APM – Agiles Projektmanagement«, drei Büchern zu Soft Skills in der IT sowie von »Testen von Software und Embedded Systemen«.

Uwe Vigenschow

Der APM-Guide

zu »APM – Agiles Projektmanagement«

dpunkt.verlag

Uwe Vigenschow
uwe@vigenschow.com

Lektorat: Christa Preisendanz
Copy Editing: Ursula Zimpfer, Herrenberg
Satz: Uwe Vigenschow, Hamburg
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de

Der APM-Guide basiert auf dem Buch
»APM – Agiles Projektmanagement«, dpunkt.verlag, 2015, ISBN 978-3-86490-211-6.

1. Auflage 2015

Wieblinger Weg 17
69123 Heidelberg

Vorwort

APM steht für ein **A**giles **P**rojekt**m**anagement von anspruchsvollen Softwareprojekten. Der *APM-Guide* ist ein kostenloser Auszug aus dem ersten Teil des Buchs *APM– Agiles Projektmanagement* [21] als E-Book. Sie erhalten im *APM-Guide* einen Überblick über die zentralen Techniken von *APM* und ihr Zusammenspiel. Er ist sowohl zum Kennenlernen von *APM* als auch später zum schnellen Nachschlagen gedacht.

Im Buch *APM– Agiles Projektmanagement* [21] werden diese Themen vertieft und um weitere Aspekte und Methoden ergänzt. Es bietet Ihnen damit einen praxisorientierten Überblick über *APM* und seinen effizienten und erfolgreichen Einsatz in der Projektarbeit. *APM* kann dabei wesentlich umfassender eingesetzt werden, als es z. B. Scrum in seiner reinen Form ermöglicht.

Sie erfahren im Buch, wie von der Projektvorbereitung bis zu einem agilen Requirements Engineering und einer durchgängigen Softwarearchitektur agil entwickelt werden kann. Dabei skaliert *APM* von Sieben-Personen-Teams bis hin zu Großprojekten mit mehreren 100 Mitarbeitern in verteilten Teams. *APM* wird auch für die Softwareentwicklung in regulierten Umfeldern wie der Luftfahrt- und Automotive-Industrie oder der Medizin- oder Pharmatechnik eingesetzt, bei denen besonders hohe Sicherheitsvorgaben einzuhalten sind.

APM kann auf Ihre spezifischen Anforderungen angepasst werden und ermöglicht Lösungswege, die in der Praxis funktionieren. Hinter dem aktuellen Stand von *APM* stecken mittlerweile über zehn Jahre Erfahrung. Vielleicht ermuntert Sie dieser *APM-Guide* auch zum Lesen des *APM*-Buchs.

Uwe Vigenschow Hamburg, im Januar 2015

Inhaltsverzeichnis

1 Die Architektur von APM

Agile Vorgehensweisen sind im grundsätzlichen Ansatz einfach zu verstehen, jedoch wird die innere Komplexität ihrer Umsetzung sofort deutlich, wenn wir versuchen, detailliert zu erklären, wie ein agil durchgeführtes Projekt konkret abläuft. Bevor wir in die Tiefen von *APM* eintauchen, finden Sie hier einen kurzen Überblick über die zentralen Elemente von *APM*. Danach klären wir Begriffe aus dem agilen Projektmanagement, die wir später noch genauer definieren werden, aber bereits für die ersten Darstellungen der Zusammenhänge in ihrer grundlegenden Bedeutung benötigen.

1.1 Was ist APM?

APM steht für **A**giles **P**rojekt**m**anagement und beschreibt ein Framework, um Softwareprojekte mit der notwendigen Flexibilität umzusetzen und dabei der hohen Dynamik des Projektumfelds angemessen begegnen zu können. Es fußt auf fünf Säulen und nutzt eine Vielzahl einzelner Best Practices aus unterschiedlichen methodischen Vorgehensweisen (Abb. 1-1):

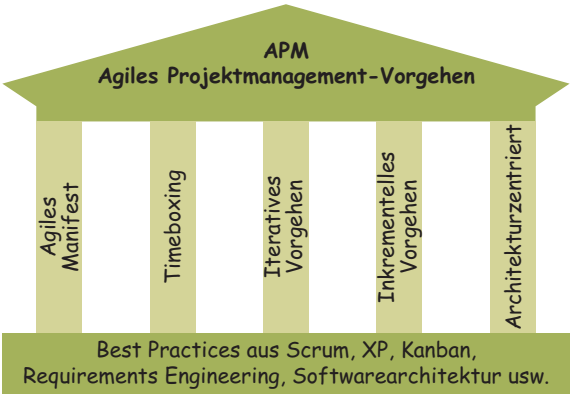

Abbildung 1-1: Das Fundament und die fünf Säulen von *APM*

- *APM* orientiert sich am Agilen Manifest und den darin beschriebenen vier Wertepaaren sowie den dort genannten zwölf Prinzipien als grundlegende Basis für das Vorgehen, die Führung und die Zusammenarbeit innerhalb des Projekts sowie mit den Stakeholdern [2].
- *APM* implementiert ein durchgängiges Timeboxing zur Planung und Steuerung von Projekten. Eine *Timebox* ist ein fester, vorab definierter Zeitabschnitt, in dem wir planen, ein inhaltliches Ergebnis zu erreichen, das wir am Ende überprüfen können. Dieses Konzept durchdringt *APM* von kleinen Abschnitten der täglichen Zusammenarbeit wie in täglichen kurzen Meetings über feste Zeitabschnitte von wenigen Wochen, Iterationen genannt, bis auf die Ebene der Auslieferungen.
- *APM* ist ein iteratives Vorgehen, d. h., der zeitliche Ablauf ist in gleich lange Iterationen von wenigen Wochen unterteilt, was einem zeitlichen Prüfraster mit festem Rhythmus entspricht.
- *APM* beinhaltet ein inkrementelles Vorgehen. In einer Iteration erarbeiten wir ein prüfbares Ergebnis von direktem Nutzen für den Kunden bzw. die Anwender, also ein Stück funktionierender Software. Das Ergebnis eines Projekts wird schrittweise erarbeitet. Ein Zwischenergebnis bezeichnen wir als Inkrement. Die Inkremente bauen aufeinander auf, sodass wir uns mit jedem weiteren Inkrement dem Projektergebnis konkret nähern. Damit ein inkrementell-iteratives Vorgehen funktioniert, strukturieren wir unser angestrebtes Projektergebnis derart, dass wir ausreichend kleine, sinnvolle Teilstücke als Inkremente innerhalb einer Iteration umsetzen können.
- *APM* ist ein architekturzentriertes Vorgehen. Die Arbeit an der Softwarearchitektur findet kontinuierlich statt, wobei regelmäßig im Projektverlauf grundlegende Architekturfragen im Vordergrund stehen.

Diese fünf zentralen Säulen von *APM* sind eingebettet in eine Vielzahl von Best Practices aus unterschiedlichen Frameworks und Methoden. Im Wesentlichen sind das Scrum, eXtreme Programming (XP), Kanban, agiles Requirements Engineering und Softwarearchitektur (Abb. 1-1). Damit ist *APM* besonders dafür geeignet, komplexe Projekte mit einem oder auch mehreren parallel arbeitenden Teams durchzuführen, in denen eine besonders hohe innere und äußere Qualität der Softwareprodukte gefordert ist.

APM stellt ein flexibles Rollenmodell bereit, über das wir weitgehend unabhängig vom Spezialisierungsgrad der einzelnen Teammitglieder crossfunktionale *Featureteams* bilden können, die jeweils gemeinsam verantwortlich einen eigenständig nutzbaren Teil des Softwareprodukts durchgängig und vollständig umsetzen [10]. So entkoppeln wir in *APM* die einzelnen Teams voneinander, sodass gegenseitig induzierte Wartezeiten zwischen den Teams minimiert werden und sich ein kontinuierlicher Projektfortschritt einstellt.

2 Die grundlegenden Konzepte von APM

Sie lernen in diesem Kapitel die Grundlagen von *APM* kennen. Das inkrementell-iterative Vorgehen steht dabei im Mittelpunkt des streng getakteten agilen Vorgehens. Im Rhythmus der Iterationen erfolgt der permanente und transparente Aufbau des Wissens über das Projekt in regelmäßigen Lernschleifen. In diesem Kapitel finden Sie auch die Definition der Grundbegriffe, auf die wir uns im Folgenden beziehen.

2.1 Projekt und Produkt

In Anlehnung an eine weitverbreitete Definition ist ein **Projekt** ein zeitlich begrenztes Vorhaben mit dem Ziel, ein einmaliges Produkt zu schaffen. Aufgrund seines einmaligen Charakters ist ein Projekt stets mit Risiken behaftet und wird meist mit einer eigenen, dafür speziell aufgebauten Organisationsstruktur umgesetzt [17]. Wir setzen im *APM-Guide Projekt* stets mit *Softwareprojekt* gleich. Auch wenn man *APM* in anderen Kontexten anwenden kann, so ist es für Softwareprojekte konzipiert.

Mit **Produkt** bezeichnen wir ein klar definiertes Stück nutzbarer Software sowie alle notwendigen Artefakte und spezielle Hardware, die für den Betrieb dieser Software notwendig sind. Dabei ist es unerheblich, ob das Produkt autark lauffähig sein soll oder als Komponente nur im Zusammenspiel mit anderen Soft- und Hardwarekomponenten einsetzbar ist.

Damit erkennen wir, dass wir, auch wenn nur Softwareprojekte im Fokus der Betrachtung stehen, unter Umständen zumindest die Schnittstellen zur Hardware- und Treiberentwicklung oder zum Betrieb mit in den Projektkontext einzubeziehen haben. Das Produkt definiert also den Projektkontext und seine Schnittstellen.

Dieser Zusammenhang ist nicht neu, aber für agil durchgeführte Projekte von besonderer Bedeutung, da die Projektschnittstellen, z. B. zur Hardwareentwicklung, an die besonderen Anforderungen agilen Vorgehens anzupassen sind. Damit greift agile Softwareentwicklung stets auch in die Vorgehensweisen benachbarter Entwicklungen, bei zuliefernden Elementen und des Betriebs des Produkts ein. Die gesamten Schnittstellen sind in der Projektvorbereitung zu klären.

2.2 Timebox und Meilenstein

Um die Ergebnisse zu erreichen, stukturieren wir die zu erbringenden Inhalte. Zur Steuerung dieses Prozesses kommen Meilensteine und Timeboxen zum Einsatz. Ein **Meilenstein** ist ein weitverbreitetes Konzept. Dazu wird ein fest umrissenes Ergebnis mit einem Termin verbunden. Dies ist nützlich, um entweder anzuzeigen, dass ein definierter Abschluss erreicht ist, oder um einen inhaltlich wichtigen Synchronisationspunkt zwischen verschiedenen Bereichen eines Projekts oder zwischen dem Projekt und seinem Umfeld abzusichern. So kann ein Meilenstein den Abschluss einer Phase definieren oder genutzt werden, um zwei Teams bezüglich ihrer Ergebnisse zu synchronisieren. Meilensteine können auch hierarchisch heruntergebrochen werden. Dann wird oft von Zwischenmeilensteinen gesprochen.

Bei Problemen mit der Erreichung des definierten Umfangs des Meilensteins wird typischerweise der Termin verschoben. Der inhaltliche Umfang bleibt bei einem Meilenstein fest. Dieser Mechanismus gehört zur Definition des Meilensteins dazu. Genau hier kommen wir zum zweiten Konstrukt, das uns dabei unterstützt, die Ergebnisse auch zu erreichen, der Timebox.

Auch bei einer **Timebox** werden inhaltlicher Umfang und geplanter Termin miteinander verbunden. Hier bleibt jedoch der Zeitrahmen unverändert. Eine Timebox ist somit ein fester Zeitraum bzw. -rahmen, dem gewünschte Ergebnisse zugeordnet sind, deren Umsetzung für diesen zeitlichen Rahmen geplant worden ist. Wenn der Zeitrahmen erreicht ist, werden die Ergebnisse bewertet. Eine Timebox bildet also ein zeitliches Prüfraster für den Projektfortschritt und unterstützt uns bei der Fortschrittskontrolle.

Für unfertige Ergebnisse erfolgt eine Restaufwandsbetrachtung, sodass diese unfertigen Teile als neue Aufgaben wieder eingeplant werden können. Falls das Ergebnis bereits vor Ende der Timebox erreicht wird, kann entweder mit neuen anstehenden Aufgaben begonnen werden oder die Timebox wird beendet, um keine Zeit zu verschwenden. Dieses agile Controlling-Instrument ist für eine inkrementell-iterative Entwicklung essenziell. Sinnvolle Timeboxen haben eine Dauer von 5 Minuten z. B. im Rahmen von Besprechungen bis zu einem Monat bei langen Iterationen.

2.3 Iteration und Inkrement

Eine **Iteration** ist eine Timebox, um einen definierten Inhalt zu erarbeiten, der einen messbaren bzw. prüfbaren Kundennutzen ergibt. Dieses neu hinzugekommene, auslieferbare und geprüfte Ergebnis einer Iteration wird als **Inkrement** bezeichnet. Mit dieser inkrementellen Entwicklung wird das Produkt also schritt- bzw. scheibchenweise entwickelt.

Über die einzelnen Schritte, also die Inkremente, erfolgt eine belastbare Fortschrittskontrolle, da jedes Inkrement einen Mehrwert für den Kunden liefert und damit von ihm prüfbar ist. Das Controlling eines agilen Projekts erfolgt ausschließlich über *fertige* Software, also auslieferbare Inkremente.

Damit das Controlling über die Inkremente funktioniert und in ausreichend kurzen Zeiträumen erfolgt, wird eine Reihe von Anforderungen an die Iteration gestellt, also die auf die Inkremente bezogene Timebox. So dauern die Iterationen eines konkreten Projekts stets gleich lang. Für fast jedes Projekt liegt dabei die erfolgversprechende Iterationsdauer zwischen zwei und vier Wochen [23]. Wir erreichen so effiziente Rückkopplungsschleifen zwischen Entwicklung und Kunde und sind gezwungen, den Gesamtumfang des Projekts in viele kleine prüfbare Schritte zu zerlegen, was meist auch die Reaktionsfähigkeit und -geschwindigkeit des Projektteams erhöht.

Wird die Iterationsdauer noch kürzer, und damit die Rückkopplungsschleifen, besteht die Gefahr, dass ein zu wenig prüfbares Ergebnis erstellt wird und damit der Rückkopplungsoverhead im Vergleich zum Nutzen zu groß wird. Wir führen das Projekt dann ineffizient durch. Bei Iterationslängen über einen Monat Dauer kommt hingegen das belastbare Feedback oft zu spät, sodass Arbeiten wieder zurückgenommen werden müssen oder wir zu lange auf Antworten warten müssen. Hier ist der Overhead für die notwendigen Rückkopplungen zwar eher niedrig, jedoch entsteht viel zu viel Mehraufwand in der Entwicklung für unnötige Korrekturen. Auch so entfernen wir uns vom betriebswirtschaftlichen Optimum. Es bleibt dabei: Zwei bis vier Wochen sind die anzustrebende Dauer für eine Iteration. Ausnahmen von dieser Regel sind sehr selten.

2.4 Iterationsebenen

Es gibt in einem *APM*-Projekt mindestens drei aufeinander aufbauende Ebenen, in denen wir iterativ vorgehen. Wir bezeichnen die auf die Erstellung des Inkrements bezogene Timebox als **Iteration** (Abb. 2-1, Mitte). Damit ergibt sich darunter mindestens eine weitere Ebene iterativen Vorgehens für die einzelnen Entwickleraufgaben, die **Tasks**, und eine Ebene darüber für das **Release** bzw. die Auslieferung (Abb. 2-1, oben bzw. unten).

Beginnen wir mit der Entwicklersicht (Abb. 2-1, unten). Um eine Iteration selbst aus Projektmanagementsicht steuern zu können, werden die einzelnen Aufgaben für einen Entwickler, die Tasks, möglichst klein geschnitten. Diese Zerlegung erfolgt durch die Entwickler selbst, da sie am besten wissen, wie sie eine Anforderung umsetzen möchten. Damit wir dort ebenfalls kleine, überschaubare, oft durch Unit Tests überprüfbare Schritte durchführen können, lautet die Rahmenbedingung für diese Zerlegung, dass deren Umsetzung nur ca. einen Tag dauern darf.

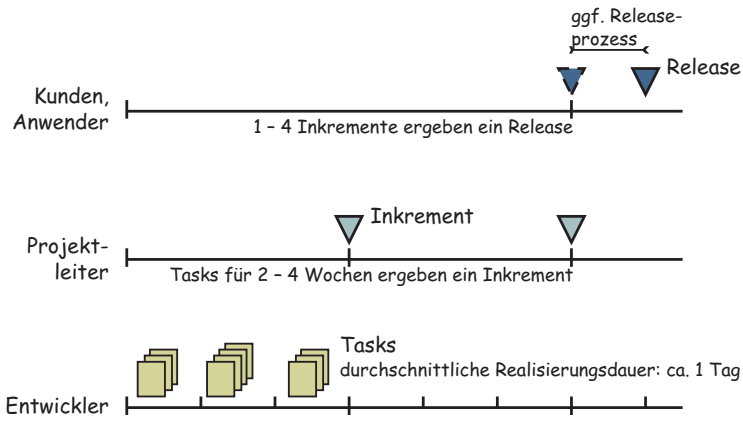

Abbildung 2-1: Die drei wesentlichen Iterationsebenen im Zusammenspiel

Damit sehen alle Beteiligten tagtäglich den konkreten Fortschritt. Der überschaubare zeitliche Horizont ist dabei häufig eine Arbeitswoche. Auf dieser Detaillierungsebene kann die Iterationsdauer für eine zielführende inhaltliche Aufgabenzerlegung einfach zu lang sein, um als Entwickler den Fokus halten zu können. Daher erfolgt dafür die Zerlegung in kleine Tasks, die eine tägliche Fortschrittsbetrachtung erlauben.

Diese Art des Arbeitens mit tagtäglichen oder besser kontinuierlichen Rückkopplungsschleifen für jeden einzelnen Entwickler stellt Bedingungen an die technische Infrastruktur. Vom in Arbeit befindlichen Gesamtsystem werden mindestens einmal täglich, besser kontinuierlich neue Arbeitsstände benötigt. Diese Arbeitsstände werden als **Builds** bezeichnet.

Damit ein Build als Element einer Rückkopplungsschleife für die Entwickler sinnvoll genutzt werden kann, benötigen wir eine integrierte Build-Umgebung. Diese erlaubt es – am besten kontinuierlich (Continuous Integration) –, neu in das Versionskontrollsystem eingecheckte Versionen zu kompilieren, automatisch mit Unit Tests zu prüfen, zusammenzubauen und das Ergebnis, den Build, ebenfalls einer kurzen automatischen Prüfung zu unterziehen. Der maximal gerade noch sinnvolle Rhythmus für einen Build-Zyklus beträgt einen Tag, sodass wir mindestens einmal in der Nacht unseren aktuellen Stand erzeugen (**Nightly Builds**).

Kommen wir nun zur Kundensicht (Abb. 2-1, oben). In der Regel bilden die Inkremente aus ein bis vier Iterationen eine Auslieferung (Release). Warum kann es sinnvoll sein, ein Release aus mehreren Iterationen zusammenstellen zu lassen? Anders als ein Inkrement, das eng an den Timebox-Charakter der Iteration gebunden ist, hat ein Release eher Meilenstein-Charakter. Da Kunden, also Auftraggeber, Fachexperten und Anwender, sowohl den Projektfortschritt als auch die Qualität des Ergebnisses im We-

sentlichen über den Einsatz der aktuellen Releases bewerten können, ist es sinnvoll, sowohl den Umfang als auch die Qualitäten der Auslieferung über den Rückkopplungseffekt von ein bis drei Iterationen abzusichern. Durch die Rückmeldung zum Inkrement können wir die Inhalte der folgenden Iteration(en) so steuern, dass die Kundenzufriedenheit maximiert und der angestrebte Releaseumfang erreicht wird.

Neben dieser Risikosteuerung kommen noch zwei weitere Gründe für die Aufteilung in Inkrement- und Releaseebene hinzu. Zum einen möchten wir die Iterationsdauer (Abb. 2-1, Mitte) möglichst kurz halten, um eine optimale Rückkopplung durch das Kundenfeedback in unser Projekt zu erreichen. Ein Inkrement bietet damit aber oft zu wenig inhaltlichen Umfang, um ein Release zu rechtfertigen. Wir benötigen zwei bis vier Iterationen dafür. Dazu kommt oft, dass für ein Release ein zusätzlicher betrieblicher Aufwand notwendig wird, um eine ausgelieferte Software oder ein Softwareteilprodukt in Betrieb zu nehmen (Abb. 2-1, oben rechts). Auch dieser Aufwand kann durch die zusätzliche Ebene wieder in ein ausgewogenes Kosten-Nutzen-Verhältnis gebracht werden.

2.5 Interne und externe Releases

Die im letzten Absatz dargelegten Argumente können dazu führen, dass z. B. in einem regulierten Umfeld ein Release erst nach ein bis zwei Jahren erfolgt. Die entsprechenden Abnahmen durch externe Gremien, wie z. B. die FDA (Food and Drug Administration[1]) im medizinischen Kontext, sind besonders aufwendig und dauern alleine oft mehrere Wochen. Das würde die aussagekräftigste Rückkopplung in unserem iterativen Ansatz verhindern.

Zur Lösung dieses Problems führen wir eine zusätzliche Ebene zwischen Inkrement und Release ein, die **interne Releaseebene** (Abb. 2-2). Dazu werden die Abnahmekriterien für ein Release so aufgeteilt, dass möglichst viele aussagekräftige, aber zeitlich gut machbare Aspekte dem internen Release zugeordnet werden und nur wenige, extrem aufwendige Aspekte noch zusätzlich für das eigentliche, externe Release hinzukommen.

Für die internen Releases gilt nun alles, was im Standardfall aus Abbildung 2-1 auf das Release zutrifft. Insbesondere liegt der Rhythmus der internen Releases bei ein bis vier Iterationen. Erst wenn es zur tatsächlichen Auslieferung kommt, wird der komplette Freigabeprozess angestoßen. Durch die vielen Durchläufe von weiten Teilen dieses Freigabeprozesses im Rahmen der internen Releases wird so die Gefahr unangenehmer Überraschungen und damit verbundener zeitlicher Verzögerungen minimiert. Dieses Vorgehen mit internen und externen Releases ist generell sinnvoll anzu-

[1] Behörde für Lebensmittelüberwachung und Arzneimittelzulassung der USA.

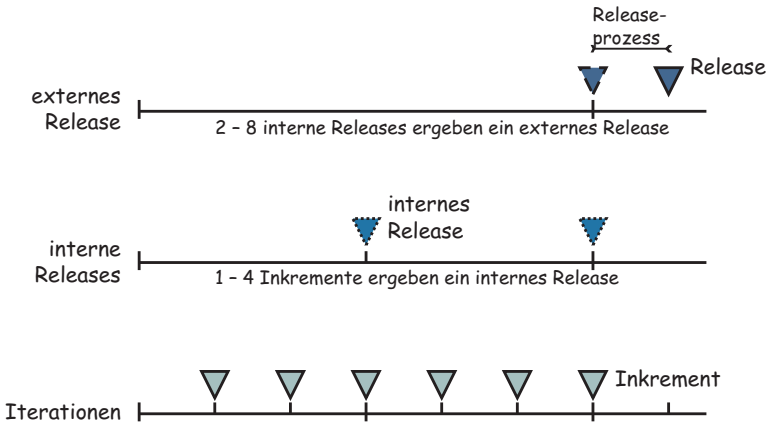

Abbildung 2-2: Interne Releases zur Entkoppelung des inkrementell-iterativen Entwicklungsprozesses von externen Vorgaben zu Releases im regulierten Umfeld

wenden, wenn durch äußere Vorgaben eine inkrementelle Inbetriebnahme über regelmäßige Releases in kurzen Abständen nicht möglich ist. Die internen Releases sind dann unsere beste Annäherung an das Ideal.

Warum können wir nicht einfach auf die internen Releases verzichten und uns nur auf die echten, externen Releases konzentrieren. Interne Releases klingen doch nach einer enormen Verschwendung. Die externen Releases sind für unsere Kunden das Wichtigste am ganzen Projekt. Jetzt bekommen sie den Gegenwert für ihre Investition. Dadurch lastet ein großer Druck auf allen Beteiligten. Stress, Nervereien und Überstunden stehen an, wo es doch eigentlich die Früchte unserer Arbeit zu feiern gilt. Wie so oft, wenn uns etwas schwerfällt und wir schnell unter Druck geraten, hilft es, die Abläufe zu üben. Nicht nur ein- oder zweimal, sondern so lange, bis jeder Handgriff automatisch sitzt. Was für Feuerwehrleute oder Leistungssportler angemessen ist, hilft uns auch: üben, üben, üben.

Die internen Releases sind wie Vorbereitungswettkämpfe vor einem wichtigen Großereignis. Wenn es dann so weit ist, sind alle Beteiligten ruhig, fokussiert und eingespielt. Das iterative Vorgehen gibt uns einen großartigen Rahmen dafür, regelmäßig alle wichtigen Abläufe intensiv zu üben. Bei den Releases wird dieser Zusammenhang besonders deutlich. Er gilt aber auch ebenso für die Reviews, Retrospektiven oder interne Abnahmeprozesse sowie für die Softwareentwicklung selbst. Letzteres wird im Software-Craftsmanship-Gedanken und der Clean-Code-Bewegung z. B. durch Code-Katas aufgegriffen, in denen regelmäßig Codingtechniken und die Anwendung von Mustern und Prinzipien geübt werden [13, 19].

3 Anforderungen strukturieren

»In agilen Projekten arbeitet man mit User Stories!« Diesen Satz haben Sie vielleicht auch schon gehört. Doch was sind User Stories und wofür sind sie gut geeignet? Wie spielen die User Stories mit der Anforderungsanalyse z. B. mit Use Cases bzw. der Unified Modeling Language (UML) zusammen? Den Antworten auf diese Fragen widmen wir uns in diesem Kapitel.

3.1 User Stories

User Stories liefern einerseits analytische Aspekte, andererseits dienen sie uns als zentrale Elemente der Projektsteuerung. Das liegt daran, dass in einer User Story Analyse- und Planungsaspekte vermischt sind. *APM* zeichnet einen Weg auf, ihre Vorteile zu nutzen, ohne von den Nachteilen behindert zu werden. Was ist also eine User Story?

Eine **User Story** beschreibt aus der Perspektive eines Nutzers des von uns entwickelten Produkts, was das System machen soll. Formal besteht sie aus einem Titel und der Beschreibung einer Nutzer-System-Interaktion. Mit einer User Story beschreiben wir zusammengehörige Anforderungen, die in funktionierender Software umsetzbar und ausführbar sind, die wir eigenständig testen können und die einen Wert (Business Value) bzw. Produktcharakter für den Anwender bzw. Kunden haben. Das allgemeine Muster und ein Beispiel können Sie Abbildung 3-1 entnehmen.

Zwei Regeln sind für User Stories noch relevant: CCC und INVEST. CCC steht für **C**ard, **C**onversation, **C**onfirmation, also Karteikarte, Gespräch und Bestätigung durch Abnahmekriterien [9]. An eine User Story stellen wir also nicht den Anspruch, eine vollständige Beschreibung zu liefern. Sie ist vielmehr ein Merker dafür, dass Entwickler, Projektleiter und Fachexperten gemeinsam darüber reden, was genau gewünscht ist. Neben einer kurzen inhaltlichen Beschreibung in Form einer Demonstration finden wir auf der Karte noch die Akzeptanzkriterien, um das Ziel besser verstehen zu können. Damit *repräsentiert* eine User Story eher eine Anforderung, als dass sie diese dokumentiert [3].

Mit dem Akronym INVEST werden bestimmte Anforderungen an eine *gute* User Story bezeichnet. Im Einzelnen sind dies [24]:

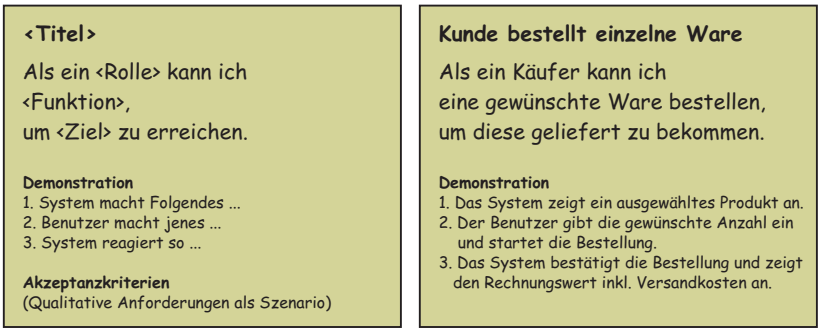

Abbildung 3-1: Muster und Beispiel einer User Story

I ndependent – unabhängig: Können die repräsentierten Anforderungen in mehreren User Stories unabhängig voneinander umgesetzt, getestet und ausgeliefert werden?

N egotiable – verhandelbar: Eine User Story stellt einen Platzhalter für eine Anforderung dar. Die eigentlichen Inhalte und Details werden in direkten Gesprächen zwischen Entwickler, Projektleiter und Fachexperte bestimmt und ausgehandelt.

V aluable to users or customer – von geschäftlichem Wert für Anwender oder Kunde: Lässt sich mit der repräsentierten Anforderung ein geschäftlicher Wert erreichen? Worin besteht der Wert und kann er quantitativ bestimmt werden?

E stimatable – abschätzbar: Kann das Entwicklerteam den Aufwand für die Umsetzung relativ zu anderen User Stories abschätzen?

S mall – klein: Ist die repräsentierte Anforderung innerhalb einer Iteration realisierbar?

T estable – testbar: Wie lässt sich überprüfen, ob die Anforderung auch wie gewünscht umgesetzt wurde? Welche Akzeptanzszenarien gibt es?

Jetzt wissen wir vorerst genug, um User Stories bewerten zu können. Doch warum werden auch die bewährten Analysemethoden wie Use Cases bzw. UML noch eingesetzt? User Stories haben auch ihre Nachteile, insbesondere, wenn die analytische Sichtweise gefordert ist.

User Stories sind so klein, dass es schwierig ist, nur mit ihnen den Überblick über die übergeordneten Abläufe zu bekommen bzw. zu behalten. Sie fördern nicht den Blick auf das Ganze. User Stories geben auch kein Abstraktionsniveau vor, was das gemeinsame Verständnis erschwert, wenn verschiedene Personen User Stories schreiben. Sie unterstützen uns auch nicht dabei, komplexe Abläufe zu durchdringen. Dabei helfen uns dann Use Cases.

3.2 Use Cases

Ein Anwendungsfall oder englisch Use Case ist in der UML etwas *schwammig* definiert, weil die UML keine Methodik vorgibt und daher das Konstrukt des Use Case offen hält. Wenn wir eine weitverbreitete Methodik hinzunehmen, können wir einen **Use Case** pragmatisch in folgender Weise definieren [14]:

> Ein Anwendungsfall beschreibt eine *zeitlich zusammenhängende* und zielgerichtete Interaktion eines Akteurs mit einem System, an deren Anfang *ein fachlicher Auslöser* steht und an dessen Ende ein definiertes *Ergebnis von fachlichem Wert* entstanden ist.

Im Unterschied zu Prozessen, die in der Regel zeitlich unterbrechbar definiert sind, befindet sich ein Use Case auf dem Abstraktionsniveau eines Prozessschrittes (Abb. 3-2, links). Da sein Anfang und Ende über den fachlichen Auslöser und geschäftlichen Wert definiert sind, beschreiben wir in einem Use Case meist längere Interaktionen weitgehend voll umfassend, in denen mehrere mögliche Varianten und Ausnahmen betrachtet werden.

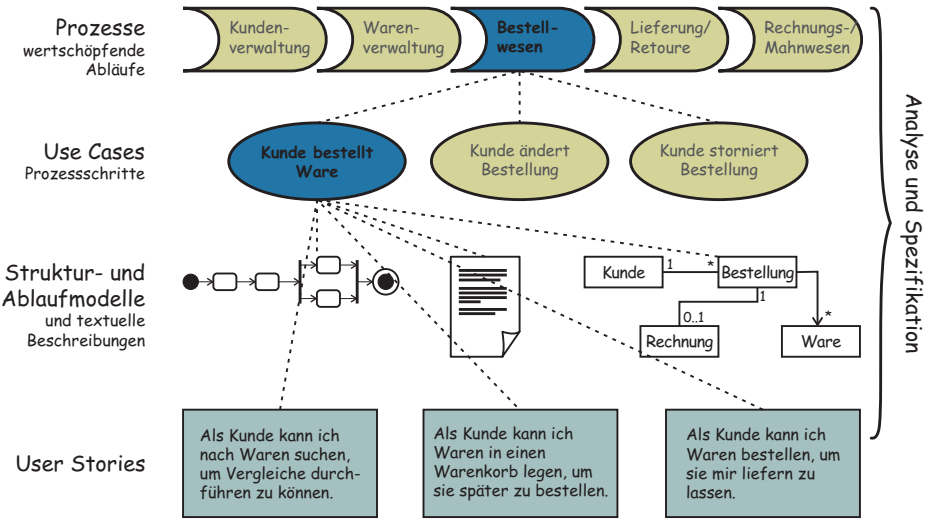

Abbildung 3-2: Use Case: Prozessschritt und Klammer über User Stories

Use Cases sind also meist umfangreicher als User Stories. Sie haben einen klaren Bezug zu übergeordneten Prozessen und lassen sich relativ eindeutig und damit einheitlich beschreiben. Mit Use Cases in der detaillierten Projektplanung und -steuerung zu arbeiten, ist möglich, aber oft zu grob,

da ihre Umsetzung meist nicht ganz in unsere möglichst kurzen Iterationen passt und damit *unhandlich* ist.

Was allerdings gut zusammenpasst, ist die Kombination aus Use Cases und User Stories. So können wir methodisch sauber und effizient auf Basis von Use Cases und der UML unsere Anforderungen beschreiben, um dann die Use Cases in User Stories zu zerlegen. Dies erfolgt, bevor sie zu ihrer Umsetzung in der nächsten Iteration anstehen. Use Cases und User Stories sind also kein Widerspruch, sondern eine perfekte Ergänzung (Abb. 3-2). Damit bilden die Use Cases das Äquivalent zum Begriff **Epos** (engl.: epic) in Scrum. Use Cases können in der UML in fachlich zusammenhängenden Paketen gruppiert werden. Der dazu äquivalente Begriff in Scrum lautet **Thema** (engl.: theme) (Abb. 3-5).

3.3 Story Map

Für User Stories und meist auch eine Abstraktionsebene höher für Anwendungsfälle gilt, dass wir damit eine einzelne Anforderung beschreiben. Um zu erkennen, wie diese in den Gesamtzusammenhang der Menge an Anforderungen passen, stellen wir sie in einen gemeinsamen Kontext, d. h. zueinander in Bezug. In einem eher traditionellen Projektrahmen erfolgt das z. B. durch eine Geschäftsprozessmodellierung. Dabei stoßen wir bei einem agilen Projekt auf das Problem, dass eine typische Geschäftsprozessmodellierung meist bereits zu detailliert ist und daher auch zu viel Zeit für die Erstellung benötigt, als dass wir sie in unsere agile Vorgehensweise integrieren könnten.

In einem agilen Projekt wird der Kontext über das Backlog für das Produkt geschaffen. Das reicht aber meist nicht aus. Wir können jedoch das Backlog anders strukturieren, um den Kontext für die einzelnen Anforderungen herzustellen, und transformieren das Backlog in eine Story Map.

Eine **Story Map** ist die visuelle Darstellung der geordneten Anforderungen (Backlog) entlang von zwei Achsen (Abb. 3-3). Horizontal läuft eine grob zeitlich geordnete Sicht auf unsere Anforderungen und vertikal die Wichtigkeit einer Aufgabe oder einer Detailaufgabe für das Produkt [16]. Die Aufgaben aus unserem Backlog sind also ihrer Ordnung nach von links nach rechts und dann von oben nach unten angeordnet.

Die Beschreibung einer Story Map verwendet die drei Abstraktionsstufen Aktivität, Aufgabe und Detail. Das Prinzip lässt sich auf jede hierarchische Form von Aufgaben anwenden. Werden auf der Ebene der Aufgaben User Stories verteilt, so entsprechen die Details zusätzlichen Anforderungsdetails zu den Stories. Die Aktivität als Bündelung von zugeordneten User Stories kann als Epos bezeichnet werden. Zur besseren Übersicht werden die Details unter die zugehörige Aufgabenkarte geschoben [25].

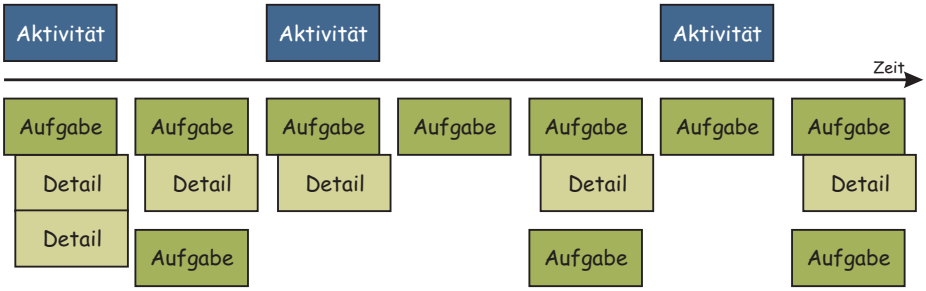

Abbildung 3-3: Das Prinzip einer Story Map [16]

Im Kontext von *APM* werden wir auf der obersten Ebene der Aktivitäten statt Epen meist Themen, also Prozesse bzw. Use-Case-Pakete, finden. Der Aufgabenebene ordnen wir Use Cases bzw. Epen zu und setzen User Stories auf der Detailebene ein. Für kleinere Projekte kann auch der Use Case auf der oberen Aktivitätsebene liegen und die User Stories auf der Ebene der Aufgaben. Diese Form der Darstellung ist vielseitig einsetzbar und erlaubt es, jederzeit den Überblick zu haben.

Der Bezug zwischen Story Map und Backlog wird offensichtlich, wenn wir der Story Map zusätzlich zur horizontalen Gliederung noch eine vertikale Achse hinzufügen (Abb. 3-4). So erhalten wir die Möglichkeit, auch die Wichtigkeit bzw. Priorität der einzelnen Stories auszudrücken. Die Priorität entspricht eher Prioritätskategorien, denen die Stories zeilenweise zugeordnet sind. Innerhalb einer Zeile läuft die innere Ordnung der Stories von links nach rechts. Sammeln wir die Stories also von links nach rechts zeilenweise ein, erhalten wir die Reihenfolge für unser Backlog.

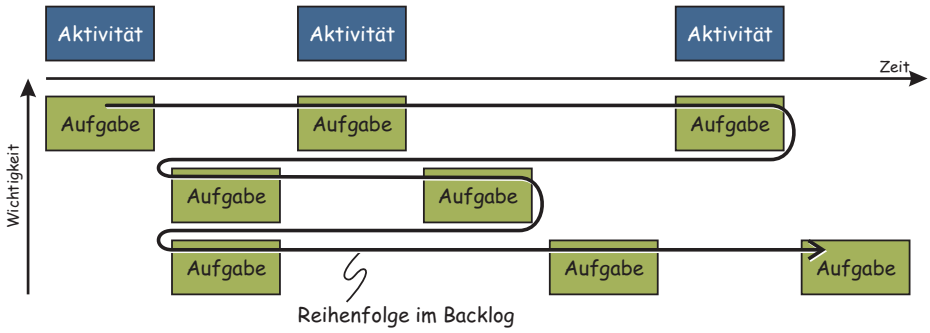

Abbildung 3-4: Die zeilenweise Anordnung der User Stories in einer Story Map ergibt die Reihenfolge im Backlog unseres Produkts [16].

3.4 Features, Themen und Epen

Da *APM* zwar gut zu einer Analyse mithilfe von Use Cases und User Stories passt, aber nicht darauf angewiesen ist, verwenden wir oft den Feature-Begriff, um verschiedene Abstraktionsebenen in der Analyse den Planungsebenen zuzuordnen. Im Einzelnen sind dies [15]:

Produktfeature oder *nur* Feature bezeichnet ganz allgemein eine Anforderung an eine als eigenständig zu betrachtende Funktionalität eines Produkts.

Releasefeature bezeichnet ein Feature, das einem konkreten Release zugeordnet ist. Dabei sind zwei Randbedingungen zu beachten:

1. Vom Realisierungsaufwand her ist ein Releasefeature innerhalb der Iterationen für dieses Release vollständig umsetzbar.
2. Ein Releasefeature liefert einen wertvollen Nutzen für den Kunden und ist von der Reihenfolge der Releasefeatures in den zugeordneten Releases sowohl für den Kunden als auch aus Projektsicht sinnvoll.

Iterationsfeature ist ein bereits vom Realisierungsumfang her heruntergebrochenes Teil eines Releasefeatures, das

- vom Umfang her in eine Iteration passt,
- für sich alleine von Kundenseite prüfbar ist und
- den Kunden bzw. Anwendern einen Mehrwert bietet.

Diese Sicht auf Features passt damit gut zu den Ebenen aus Abbildung 2-1, wobei die Kundensicht mit der Ebene der Releasefeatures korrespondiert und die Projektleitersicht eher auf der Iterationssicht mit den Iterationsfeatures liegt.

Doch wie beschreiben wir Features im Zusammenhang? Wie passen Features und die Use-Cases-Analyse aus Abschnitt 3.2 zusammen? Und wie stellen wir sicher, dass die Analyseergebnisse mit der späteren Software in Design und Architektur zusammenpassen? Doch der Reihe nach ...

Features lassen sich meist gut im Zusammenspiel beschreiben. Eine solche textuelle Beschreibung nennt man je nach Abstraktionsebene ein Epos [12] oder ein Thema. Epen liegen etwa auf dem Abstraktionsniveau von Use Cases und damit von Release- oder Iterationsfeatures. Themen korrespondieren dagegen eher mit einem Produktfeature und damit mit Use-Case-Paketen bzw. Prozessen (Abb. 3-5).

Aus diesem Bezug ergibt sich ein interessanter Aspekt. Es besteht aus der Use-Case-Analyse auch ein grober Bezug zu Architektur und Design der Software bzw. zum Entity-Control-Boundary-Architekturmuster [14]. Alle drei Sichten sind in Abbildung 3-5 im Zusammenhang dargestellt.

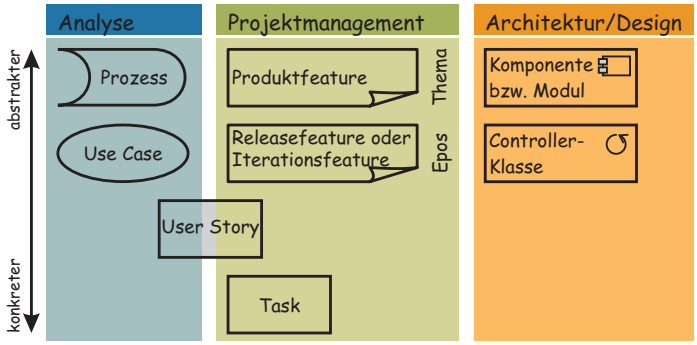

Abbildung 3-5: Analyse-, Projektmanagement- und Architekturartefakte im Zusammenspiel

Noch eine abschließende Anmerkung zum Feature-Begriff in *APM*: In der weitverbreiteten Analysemethodik von Dean Leffingwell wird nicht zwischen Produkt-, Release- und Iterationsfeature unterschieden. Leffingwell verwendet den Begriff *Feature* nur im Sinne eines Releasefeatures. Das Produktfeature entspricht bei ihm meist einem Epos und wird daher dort oft synonym gebraucht, was sicher oft als Struktur ausreichend ist [12]. In *APM* lassen wir mit dem Iterationsfeature eine weitere Differenzierung zu. Sie kann bei Projekten mit mehreren Featureteams hilfreich sein, um eine möglichst unabhängige Verteilung der Arbeit auf die einzelnen Teams zu unterstützen. Dadurch verschieben sich jedoch die Abstraktionsebenen im Vergleich zu Leffingwells Methode, die ansonsten gut zu *APM* passt.

4 Artefakte

Wir behandeln in diesem Kapitel die Artefakte zur Planung und Steuerung eines Projekts. Sie lernen die verschiedenen Planungshorizonte und -ebenen kennen, und wir betrachten mit der *Definition of Done* eines des zentralen Dokumente in APM.

4.1 Backlog, Anforderungen und Tasks

Mit dem Begriff **Backlog** wird eine geordnete Liste von Anforderungen bezeichnet, deren Umsetzung für das Produkt relevant ist. Im Backlog finden wir also alle notwendigen Aufgaben für die Produkterstellung. Dabei ist das Backlog ein Planungsartefakt und entspricht nicht der Anforderungsspezifikation, sondern wird aus ihr gefüllt (Abb. 4-1, unten). Dabei werden nicht alle Anforderungsdetails in das Backlog übertragen, sondern nur eine entsprechende Angabe oder ein Verweis auf einem Backlog-Eintrag vermerkt, der den Bezug zur Spezifikation herstellt. Die Ordnung des Backlogs erfolgt in der Regel derart, dass die anstehenden Aufgaben oben stehen und nachfolgende darunter.[1]

Da wir in *APM* wie in anderen agilen Vorgehensweisen davon ausgehen, dass die Anforderungsspezifikation fortlaufend vertieft, aktualisiert und erweitert wird, folgt das Backlog dieser Dynamik. Wir erkennen dies daran, dass weiter unten stehende Einträge weniger detailliert sind und größere Anforderungsblöcke zusammenfassen (Abb. 4-1, links). Die Arbeitsschritte Analyse, Implementierung und Test erfolgen eng verschachtelt und möglichst zeitnah und weitgehend innerhalb einer Iteration, um eine optimale Rückkopplung zu ermöglichen (Abb. 4-1, rechts oben, und 4-2).

Das Backlog in *APM* spielt dabei optimal mit Anforderungsartefakten der UML zusammen. Use-Case-Pakete bzw. Prozessbeschreibungen finden sich weiter unten im Backlog wieder und werden in der Analyse erst tiefer betrachtet, wenn sie weiter nach oben kommen. Dann werden sie in Use Ca-

[1] In Abbildung 4-1 wird auch eine sogenannte Tech-Story dargestellt. Diese Stories sind ausschließlich als Merker für umfangreichere Refactoring-Aufgaben zu sehen.

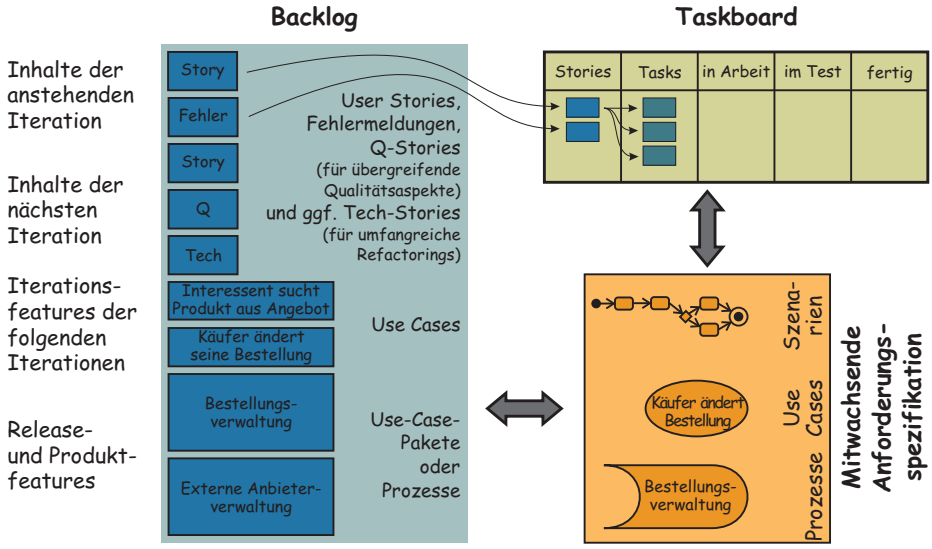

Abbildung 4-1: Das Backlog (links) als zentrale geordnete Liste aller relevanten Anforderungen an das Produkt

ses zerlegt. Diese werden dann weiter verfeinert, um daraus User Stories abzuleiten. Auf diesem Weg zerfallen vormals einzelne grobe in mehrere detaillierte Anforderungen. Dabei »wandern« die Anforderungen weiter im Backlog nach oben, bis sie ausreichend für die Umsetzung zerlegt und erfasst sind (Abb. 4-1, links).

Eine Grundvoraussetzung dafür, dass ein neuer Eintrag in das Backlog kommt, ist, dass er vom Entwicklerteam schätzbar ist. Hierfür reicht allerdings eine Schätzung auf Basis von Komplexitätsvergleichen relativ zu den anderen Einträgen aus. Alle Einträge im Backlog sind zu jeder Zeit relativ zueinander geschätzt. Diese Schätzwerte in Form von Schätzpunkten (Story Points) oder idealen Tagen bleiben in der Regel unverändert und werden nicht regelmäßig neu geschätzt.

Beim Zerlegen einer groben Anforderung wird diese aus dem Backlog entfernt und die neuen, zerlegten Aufgaben werden entsprechend ihrer Priorität in das Backlog einsortiert. Beim Zerlegen erfolgt für diese neuen Aufgaben eine neue relative Schätzung, sodass beim Zerlegen größerer Aufgaben stets auf der dann aktuellen Informations- und Erfahrungsbasis geschätzt wird. Der gesamte Schätzumfang des Backlogs, also die Summe der Komplexitätspunkte oder der idealen Tage, variiert daher stets etwas.

Werden oben stehende Aufgaben zur Umsetzung ausgewählt, so werden sie aus dem Backlog entfernt und auf die Taskboards der Entwicklerteams verteilt (Abb. 4-1, oben). Dort werden die Backlog-Einträge, also die Aufga-

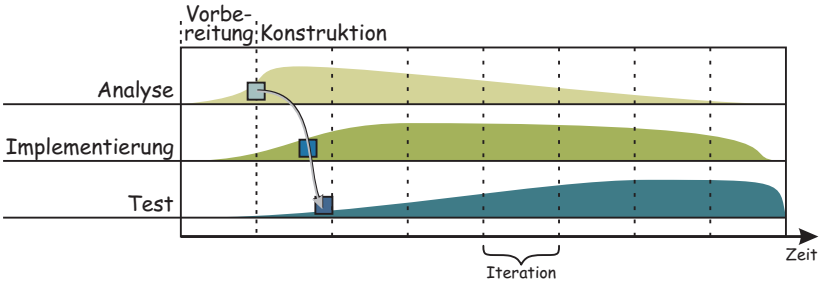

Abbildung 4-2: Das Zusammenspiel analytischer, umsetzender und testender Aktivitäten geschieht verschachtelt und möglichst zeitnah innerhalb einer Iteration (Weg vom hellen Quadrat oben links zum dunklen Quadrat unten links). Die Zerlegung großer Backlog-Einträge in ausreichend kleine User Stories erfolgt dabei mit dem kleinstmöglichen Vorlauf. Die farbigen Flächen symbolisieren die Verteilung und zeitliche Verschiebung der Arbeitsschwerpunkte über die Iterationen hinweg.

ben, weiter in Entwickleraufgaben, die Tasks, zerlegt. Dabei kommen auch alle relevanten technischen Aspekte zum Tragen. Nur nicht umgesetzte Anforderungen, die nicht in der Folgeiteration eingeplant werden, wandern, meist neu formuliert und relativ geschätzt, wieder zurück in das Backlog und werden dort einsortiert.

Da alle fachlich für unser Produkt notwendigen Aufgaben im Backlog zu finden sein sollen, werden dort auch noch nicht behobene Fehlermeldungen oder spezielle Qualitätsanforderungen in Form von sogenannten Q-Stories eingeordnet und gelistet. In einer Q-Story beschreiben wir eine qualitative Anforderung, die für mehrere User Stories gilt, aber nicht als allgemeine Anforderung in die *Definition of Done*[2] aufgenommen werden soll. Typische Beispiele für Inhalte einer Q-Story sind qualitative Anforderungen an die Benutzerschnittstelle oder an die zu verarbeitenden Mengengerüste und die notwendigen Durchsätze oder das Antwortverhalten. Über das Backlog haben wir also stets die vollständige Übersicht, was noch im Projekt ansteht und ab der nächsten Iteration umgesetzt werden soll (Abb. 4-1, links).

Eine Besonderheit stellen große technische Aufgaben wie z. B. ein notwendiges, sehr umfangreiches Refactoring dar, die unter bestimmten Bedingungen als sogenannte Tech-Stories in das Backlog aufgenommen werden können. Damit soll erreicht werden, dass wichtige technische Aufgaben, die zu umfangreich sind, um sie im Rahmen der normalen Abarbeitung von Tasks zu erledigen, auch wirklich eingeplant werden.

[2]Die Definition of Done ist die generelle Beschreibung, wann bestimmte Artefakte als fertiggestellt gelten. In Abschnitt 4.4 gehen wir genauer darauf ein.

4.2 Planungshorizonte

In *APM* gibt es drei Planungshorizonte, auf denen wir unterschiedlich weit in die Zukunft blicken. Gleichzeitig ist der Detaillierungsgrad der Planung auf jedem Horizont unterschiedlich. Je weiter wir in die Zukunft blicken, desto geringer wird der Detaillierungsgrad. Die drei Planungshorizonte ergeben sich aus den iterativen Ebenen von Task, Inkrement und Release (Abb. 2-1). Schematisch sind die Horizonte in Abbildung 4-3 dargestellt.

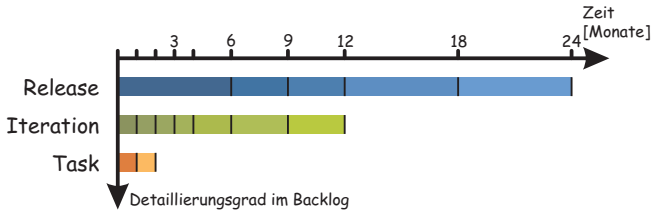

Abbildung 4-3: Planungshorizonte – das Zusammenspiel von Planungsdetaillierung und Betrachtungszeitraum. Je dunkler die Balken dargestellt sind, desto höher ist ihr relativer Detaillierungsgrad.

Wenn wir ein Projekt komplett und detailliert vorab planen würden, wäre dies eine enorme Verschwendung von Arbeitszeit für die Erstellung von Artefakten, die später anders oder gar nicht benötigt werden. Aufgrund der Komplexität von Softwareprojekten können wir die Dynamik des Projekts nicht konkret vorhersehen. Daher betrachten wir, wenn wir auf der Releaseebene weit in die Zukunft schauen wollen, auch nur mehr oder weniger große und abstrakte Blöcke von Aufgaben, die sich an den zu unterstützenden Prozessen und Abläufen orientieren. Erst wenn wir die nächste oder maximal die nächsten beiden Iteration betrachten, brauchen wir mehr Anforderungsdetails. Konkret benötigen wir alle Informationen erst, wenn eine Aufgabe zur Umsetzung direkt ansteht.

Auf diese Art und Weise halten wir den Anteil später in dieser Form nicht benötigter Detaillierungen gering, die später wieder überarbeitet und angepasst werden müssen oder im schlimmsten Fall gar nicht mehr notwendig sind. Auch minimieren wir so die Rückkopplungsschleifen und nehmen die nächste Detaillierung auf der Basis des aktuellen Wissens über die laufende Software vor.

Der aus unserer Erfahrung maximale Zeithorizont für die Grobplanung agiler Softwareprojekte liegt bei 18 bis 24 Monaten. Umfangreichere Projekte können auf noch abstrakterer Ebene strukturiert und dann in ein bis zwei Jahresprojekte zerlegt werden. Dabei ist es wichtig, dass jedes dieser Teilprojekte einen einsetzbaren und nutzbaren Wert für den Kunden liefert.

Auf diese Art wird das Folgeprojekt durch konkrete Erfahrung aus dem Vorgängerprojekt abgesichert.

Wenn wir die Releaseplanung mit einer Vorlaufzeit von 12 bis 24 Monaten iterativ entwickeln, dann hat die Iterationsebene, auf der wir die Inkremente definieren, einen Zeithorizont von 6 bis maximal 12 Monaten. Die Tasks innerhalb einer Iteration werden nur für die anstehende Iteration aus den Aufgaben aus dem Backlog heruntergebrochen bzw. abgeleitet. Bei Projekten mit mehreren Teams und einer Lookahead-Planung kann diese Vorlaufzeit maximal acht Wochen betragen. Auf das *Lookahead* gehen wir in Abschnitt 5.1 genauer ein. Meist werden bei Iterationen von drei oder vier Wochen Dauer noch nicht einmal alle Tasks für die Iteration erstellt und nach etwa der Hälfte der Iteration ergänzt und nachgearbeitet, was als Taskboard-Pflege und Verfeinerung bezeichnet wird (Abschnitt 5.2).

4.3 Planungsebenen

Aus den mindestens drei iterativen Ebenen von Task, Inkrement und Release aus Abbildung 2-1 und den Planungshorizonten aus Abbildung 4-3 ergeben sich mindestens drei damit zusammenspielende Planungsebenen (Abb. 4-4). Entlang dieser Ebenen erhalten wir stets den jeweils notwendigen Überblick oder können schrittweise die Anforderungen konkretisieren und detaillieren.

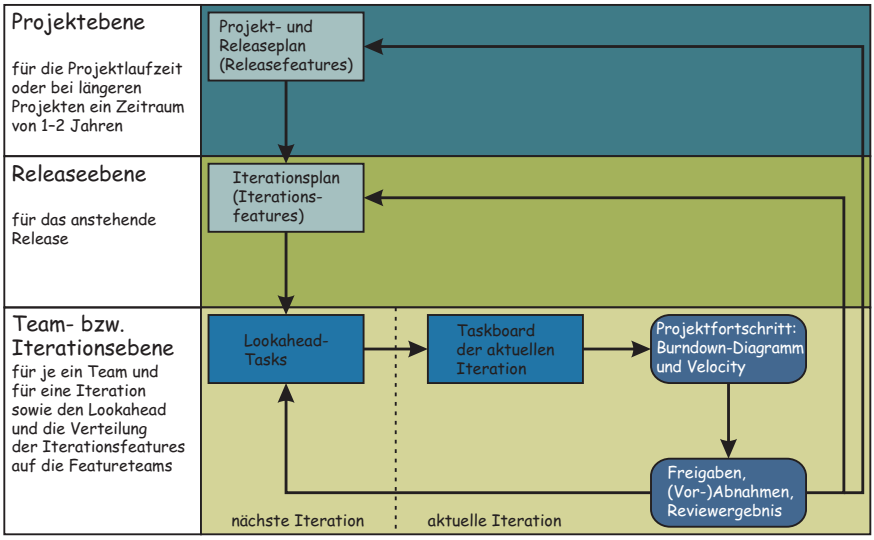

Abbildung 4-4: Rückkopplungsschleifen innerhalb der Iteration und über Release- und Produktebene hinweg [15]

In Abbildung 4-4 erkennen wir auch die verschiedenen Rückkopplungsschleifen, die uns unter anderem dazu dienen, Auswirkungen auf unsere bisherige Planung früh zu erkennen, um diese an die aktuellen Gegebenheiten anzupassen. So erfolgt die Steuerung des Projektfortschritts hin zum tatsächlich benötigten Produkt.

Dazu kommt noch eine übergeordnete Sichtweise auf das ganze Produkt, also die Summe aller Releases. Wir beginnen unsere Betrachtungen zu den Planungsebenen und den sich daraus ergebenden Artefakten auf der übergeordneten, abstraktesten Ebene, der Produktvision (Abb. 4-5). Daraus leiten wir einen ersten Releaseplan in Form einer Story Map ab. Die Story Map wird in das Backlog überführt, aus dem heraus die Iterationen geplant werden.

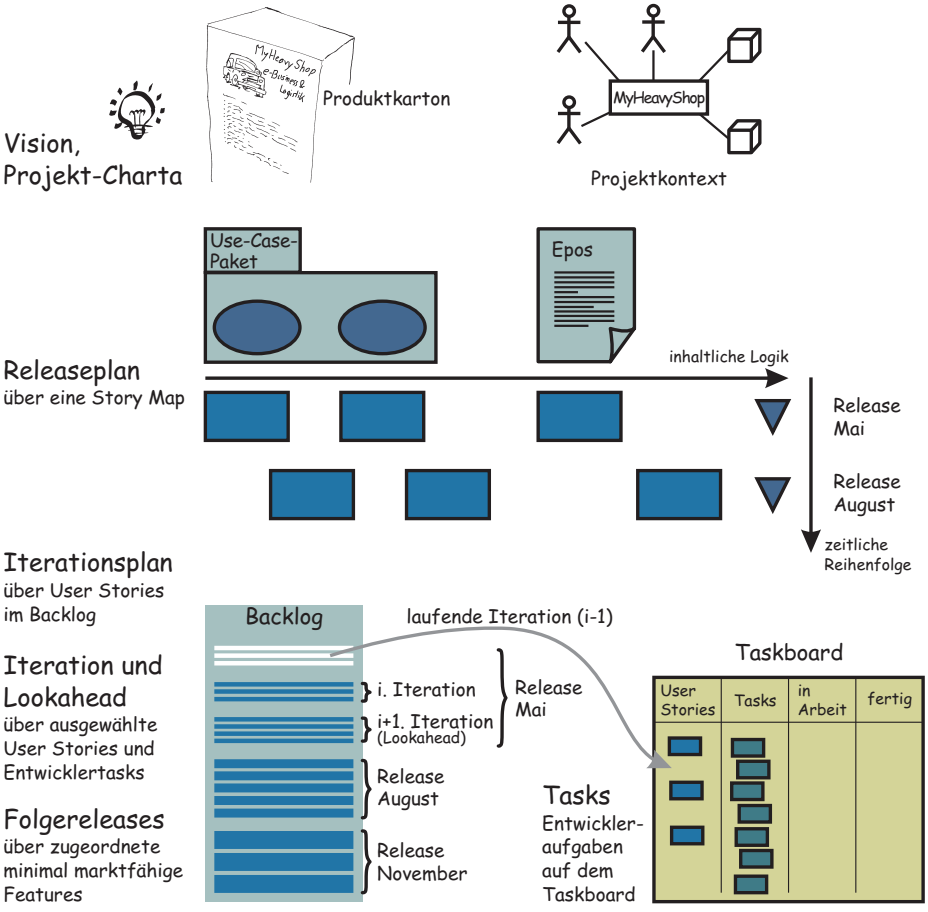

Abbildung 4-5: Planungsebenen – das Zusammenspiel von Vision, Release und Iteration

4.3.1 Produktvision, Projekt-Charta und Projektkontext

Eine Projekt-Charta bildet die oberste Leitlinie eines Projekts. Das klingt jetzt sehr gewichtig und ihre Erstellung kostet auch etwas Mühe. Sie sollte jedoch auch bei großen Projekten übersichtlich bleiben, mit geringem Aufwand überarbeitet werden können und daher auf ein Blatt Papier passen. Eine **Projekt-Charta** besteht aus drei Teilen [8]:

Produktvision: die Antwort auf die Frage nach dem Grund für das Projekt
Mission: die Antwort auf die Frage, was getan wird, um die Ziele des Projekts zu erreichen
Erfolgskriterien: die außerhalb des Projektkontextes sichtbaren Aspekte und Faktoren, an denen das Management den Erfolg des Projekts überprüfen kann

In der Produktvision (Abb. 4-5, oben) können wir uns der Frage auch spielerisch nähern und versuchen, den Produktkarton, also die Verpackung, für das erwartete Ergebnis des Projekts inhaltlich zu gestalten. Auch wenn unser Produkt nie im Regal eines Händlers stehen wird, erlaubt dieser Betrachtungswechsel eine schnelle und motivierende Beschreibung der zentralen Inhalte einer Projekt-Charta. Wir finden in der Beschreibung des Produktkartons die zentralen Features, die wertvollsten Aspekte für den Kunden und eine Reihe von Rahmenbedingungen sowie möglichen Schnittstellen zu anderen Softwareprodukten. Es lohnt sich, mit der Metapher des Produktkartons in der frühen Projektphase zu arbeiten, da sie uns schnell auf die essenziellen Bestandteile des Projekts führt und sich die Projekt-Charta daraus ableiten lässt. Zusätzlich ist es sinnvoll, ein Diagramm des Systemkontextes zu erstellen. Darin steht das Projektergebnis als Blackbox im Zentrum der Darstellung und alle Schnittstellen nach außen werden vereinfacht modelliert. Dabei werden sowohl technische Schnittstellen zu Umfeldsystemen[3] betrachtet als auch die direkten Zugriffe durch menschliche Akteure (Abb. 4-5, oben rechts).

4.3.2 Releaseplan und minimal marktfähiges Release

Aus den zu Projektbeginn noch überwiegend groben Anforderungen kann ein erster Releaseplan abgeleitet werden, indem die Anforderungen einerseits in eine fachliche Reihenfolge nach ihrer inhaltlichen Logik gebracht werden und andererseits nach einer fachlichen Priorität eine Gruppierung in einzelne fachliche Releases erfolgt.

Da die Genauigkeit auch bei einer solchen groben Planung abnimmt, je weiter in die Zukunft geplant wird, werden meist nur die ersten Relea-

[3] Andere Softwaresysteme, mit denen unser Projektergebnis interagiert.

ses genau betrachtet und spätere Releases nur grob oder gar nicht weiter ausdifferenziert. Für die Zusammenstellung eines ersten oder zweiten Releases gilt die Vorgabe, die vom Umfang her kleinste sinnvoll mögliche und für den Kunden eigenständigen Nutzen liefernde Menge an Anforderungen zu wählen. Es wird daher als minimal marktfähiges Release bezeichnet. Auf diesem Weg kommen wir zu möglichst kurzen Lieferzyklen.

Ein minimal marktfähiges Release beinhaltet also die kleinste Menge an Funktionalität, die umgesetzt sein muss, damit ein Kunde einen zusätzlichen Wert erhält. Es ist *minimal*, weil es nicht marktfähig wäre, wenn es kleiner wäre, und es ist *marktfähig*, weil es Anwender nutzen und Kunden kaufen würden, wenn es als Teil eines Produkts ausgeliefert werden würde. Methodisch kann uns bei dieser Arbeit unsere Story Map unterstützen. Neben der Zeitachse kommt senkrecht dazu die Wichtigkeit als zweite Achse. Jetzt können wir einzelne Aufgaben herausziehen und nach unten verschieben. So ergeben sich mehrere Zeilen, die jeweils für ein minimal marktfähiges Release stehen (Abb. 4-6). Die obere Zeile der Aktivitäten bildet das *Rückgrat* unserer inhaltlichen Entwicklung und das erste minimal marktfähige Release. Die erste Zeile der Aufgaben beschreibt das erste lauffähige *Skelett* unseres Produkts [16].

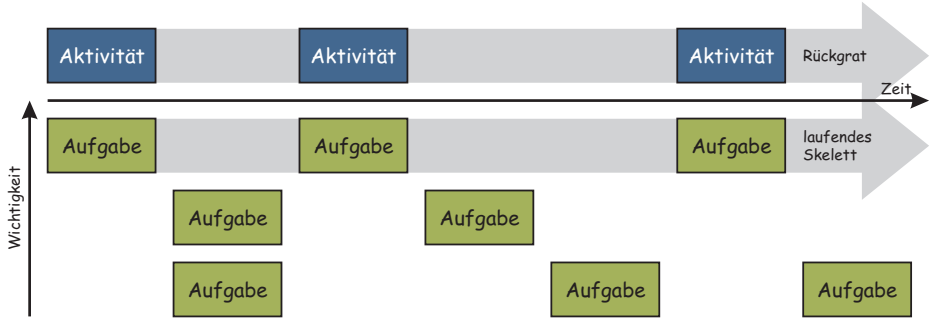

Abbildung 4-6: Über die Story Map minimal marktfähige Releases zusammenstellen [16]

Da es in der Regel nicht möglich ist, alle Anforderungen vorab detailliert zu erfassen, was ja einen der zentralen Ausgangspunkte agilen Vorgehens darstellt, spiegelt sich das iterative Klären und Ordnen von Anforderungen auch im Releaseplan wider. Den Effekt, dass nur die ersten Releases vergleichsweise klar sind, haben wir bereits beschrieben. Dies schlägt jedoch auch auf die Definition der minimal marktfähigen Releases durch. Zu Beginn des Projekts ist die konkrete Bedeutung von *minimal* für Teile des Produkts noch schwammig. Im Laufe der Analyse und der daraus resultierenden Zerlegung der oberen Anforderungen im Backlog wird auch das

minimal marktfähige Release immer klarer. Die Releaseplanung unterliegt also den gleichen Rhythmen und Effekten wie die Iterationsplanung und sie geht Hand in Hand mit dem Erstellen des initialen Backlogs.

4.3.3 Iterationsplan

Der Iterationsplan beinhaltet die Anforderungen, die den nächsten Iterationen zugeordnet sind. Auch hier sind nur die nächsten beiden Iterationen bereits mit auf User-Story-Niveau zerlegten Anforderungen inhaltlich geplant. Nachfolgende Iterationen haben nur einen gröber zugeordneten Inhalt (Abb. 4-5, unten links).

Die Stories der anstehenden Iteration werden zu Beginn aus dem Backlog entnommen und auf die Taskboards der einzelnen Teams verteilt. Die Ordnung bzw. Reihenfolge der Stories bleibt dabei erhalten (Abb. 4-5, unten rechts). Auf den Taskboards erfolgt dann die Zerlegung in einzelne Tasks, die im Laufe der Iteration abgearbeitet werden. Der Projektplan besteht also aus dem geordneten Backlog, dessen Einträge anhand der Schätzungen den nächsten Iterationen bzw. Releases zugeordnet sind, und den aktuellen Taskboards der Teams. Letztere sind auf dem tiefsten Detaillierungsniveau, betrachten aber nur den Zeithorizont einer Iteration. Im Backlog finden wir die Übersicht über das Projekt oder bei großen Projekten über ca. ein Jahr der Projektlaufzeit, ohne jedoch eine zu feine Detaillierung zu haben.

4.4 Definition of Done

Da es unter Umständen sehr unterschiedliche Sichtweisen unter den Projektteammitgliedern gibt, wann eine Aufgabe fertig ist, kann es in der Kommunikation darüber leicht zu Missverständnissen kommen, die oft die Qualität des Produkts betreffen. Damit sind die allgemeinen Abnahmekriterien für eine Aufgabe auf einer Iterationsebene essenziell für ihren Umfang und beeinflussen so die Schätzungen.

Die Definition of Done ist ein grundlegendes Dokument, das deshalb im Konsens vom ganzen Projektteam erstellt wird. Es gibt für alle Entwicklungsschritte die Qualitätsvorgaben für die entsprechenden Arbeitsergebnisse vor und steuert damit die grundsätzliche Produktqualität, die angestrebt wird. Damit beeinflusst die Definition of Done auch wesentlich den Entwicklungsprozess und damit den Gesamtaufwand.

Der Konsens im Projektteam über die notwendigen, allgemeingültigen Abnahmekriterien einer Aufgabe auf einer der Iterationsebenen ist daher wertvoll und wird in der *Definition of Done* (DoD) schriftlich festgehalten. Es gibt also für jede Iterationsebene einen eigenen, diesbezüglichen Teil in

der Definition of Done. Damit haben wir bei der Definition of Done mindestens die folgenden drei Ebenen zu betrachten (Abb. 4-7):

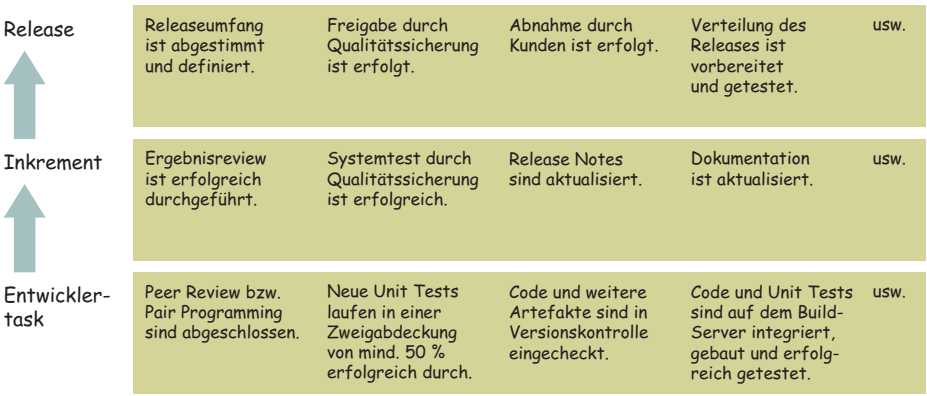

Abbildung 4-7: Beispiel für die drei Ebenen einer Definition of Done

- Entwicklertaskebene: Tasks abarbeiten, Komponenten integrieren und Unit Tests durchführen
- Inkrement- bzw. Iterationsebene: System integrieren, Integrations- und Systemtests durchführen und Review des Inkrements
- Releaseebene: Release definieren und erstellen sowie Freigabe- und Abnahmetests durchführen

Grundsätzlich gilt eine Definition of Done speziell für ein Projekt und ist dort allerdings allgemeingültig. Sie wird in der Projektvorbereitung und Architekturphase vom Projektteam erarbeitet und verbindlich beschlossen. Bei Bedarf erfolgen Erweiterungen oder Änderungen, was jedoch im laufenden Projekt nur eher selten der Fall ist.

5 Meetings

Im Rahmen des iterativen Vorgehens nach *APM* gibt es eine Reihe regelmäßig in einer Iteration vorkommender Besprechungen und Zusammenkünfte (Abb. 5-1). Betrachten wir sie der Reihe nach.

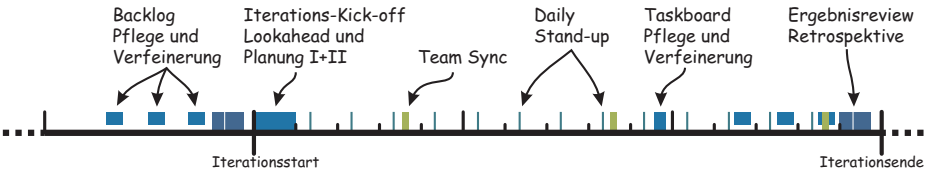

Abbildung 5-1: Verteilung der regelmäßigen Meetings in einer Iteration

5.1 Mehrstufige Iterationsplanung und Lookahead

Die Iteration beginnt mit einem gemeinsamen Kick-off von wenigen Minuten und der konkreten Planung der Iteration für den Rest des Tages. Diese Planung erfolgt in drei Schritten, dem Lookahead, der Planung I und der Planung II, die auf dem aktuellen Stand des Backlogs aufsetzen (Abb. 5-2).

5.1.1 Kick-off: Jede Iteration ist ein kleines Projekt

Man kann die Zerlegung eines Projekts in Releases und Iterationen auch als Zerlegung eines großen Projekts in viele aufeinander aufbauende, kleine Projekte mit einem eigenen prüfbaren Ergebnis, dem Inkrement, verstehen. Infolgedessen gibt es zum Beginn einer Iteration einen kurzen Kick-off mit allen Beteiligten. Das bedeutet, bei mehreren parallel arbeitenden Featureteams ein gemeinsames Meeting zu gestalten.

Ziel dieser Veranstaltung ist, für alle Projektbeteiligten die übergeordneten Zusammenhänge und Abhängigkeiten darzustellen und die Wünsche an das Ergebnis zu äußern. Gerade bei größeren Projekten mit mehreren Teams ist der Kick-off für die gemeinsame Identität des gesamten Projektteams besonders wichtig. Die Dauer des Kick-off beträgt nur maximal 15

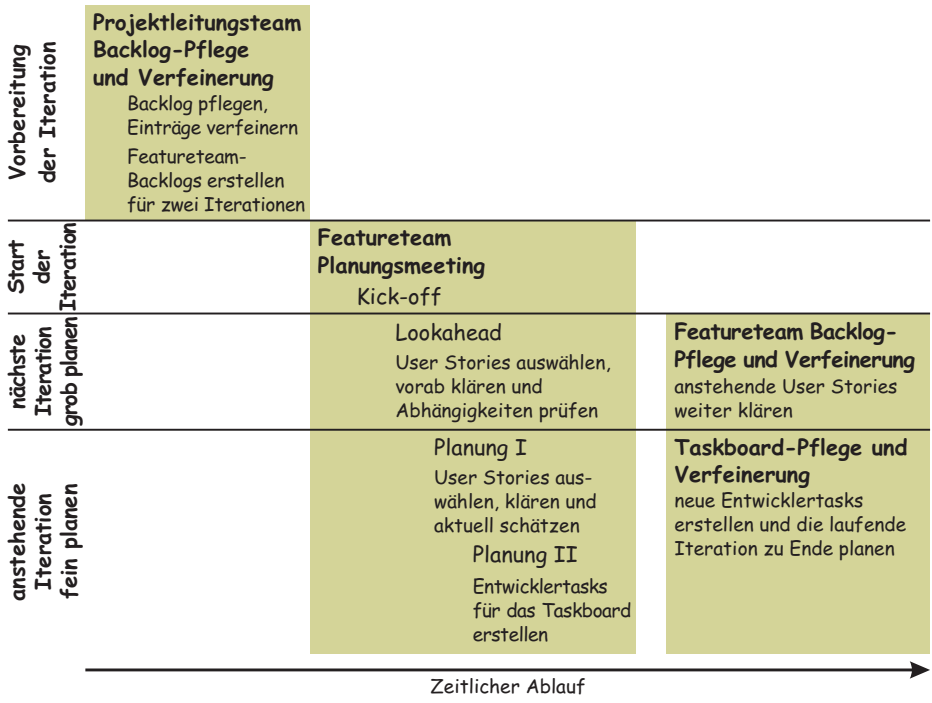

Abbildung 5-2: Prinzip der mehrstufigen Iterationsplanung und ihrer Vorbereitung in der Backlog-Pflege und Verfeinerung

Minuten, gerne weniger. Es ist jedoch als gefühlter Startpunkt für die Iteration die zentrale Informationsveranstaltung und für den Zusammenhalt im gesamten Team wertvoll.

Die folgenden Planungsbesprechungen schließen direkt an den Kick-off an und finden in den jeweiligen Featureteams statt (Abb. 5-2). Dazu muss als Voraussetzung das Backlog auf einem aktuellen und ausreichend detaillierten Stand sein. Zusätzlich hat bei mehreren Featureteams die Verteilung von Features auf die Teams bereits stattgefunden.

5.1.2 Rolling Lookahead: der Blick über den Tellerrand

Die Planungen beginnen mit dem Lookahead, das auch als *Rolling Lookahead*, *Rolling Wave* bzw. *Progessive Elaboration* oder *Zwei-Bugwellen-Planung* bekannt ist [4, 8, 15, 17]. Dabei werfen wir zuerst einen Blick hinter die anstehende, d. h. auf die nachfolgende Iteration (Abb. 5-2).

Dabei werden im ganzen Featureteam fachliche Fragen geklärt und es folgt eine erste grobe Zerlegung der zugeordneten Backlog-Einträge in Entwickleraufgaben, die *Tasks*. Damit erreichen wir bei größeren Projekten mit

mehreren parallel arbeitenden Featureteams eine bessere Möglichkeit, die groben Aufgaben mit geringst möglichen Abhängigkeiten zwischen den einzelnen Teams zuzuordnen. Auch bei sehr risikoreichen Projekten kann sich diese Planungstechnik bereits bei nur einem Featureteam lohnen. Die riskanten Aspekte können besser herausgearbeitet und dann zum Teil bereits in der anstehenden Iteration, z. B. in einem angemessenen Design durch die Auswahl eines speziellen Entwurfsmusters, angegangen werden.

Damit das Lookahead nicht zum unnützen Overhead verkommt und somit Verschwendung wäre, ist es wichtig, das Meeting fokussiert zu halten. Dabei hilft eine feste Timebox. Das Lookahead kann je nach Teamgröße und Iterationsdauer ein bis zwei Stunden dauern. Da es hierbei primär um Informationsgewinn für die anstehende Iteration geht, müssen in der Regel nicht alle Themen abschließend behandelt werden. Einen Teil dieses Aufwands bekommen wir wieder herein, weil wir in Planung I der *nächsten* Iteration von dieser Vorarbeit profitieren sollten. Wichtig ist allerdings dafür, dass das Backlog entsprechend vorbereitet ist. Das gilt jedoch in noch viel stärkerem Maße für das direkt nachfolgende *Planung-I*-Meeting.

5.1.3 Planung I: funktionaler Umfang und Iterationsziel

An Planung I nehmen das ganze Team mit dem Projektleiter und bei Bedarf fachliche Ansprechpartner und Experten teil. Es geht darum, die Backlog-Einträge für die anstehende Iteration auszuwählen. Dafür braucht das Entwicklerteam in der Regel weitere Informationen und Hintergrundwissen über die einzelnen Einträge im Backlog. Das Ziel dieses Meetings ist es, dass die Entwickler genau verstehen, was von ihnen erwartet wird.

Erst dann kann ggf. eine neue Schätzung durch die Entwickler erfolgen. Auf deren Basis wählen diese dann die Aufgaben unter Berücksichtigung der Reihenfolge aus dem Backlog aus, von denen die Entwickler meinen, sie in der Iteration umsetzen zu können. Hierbei kann es auch vorkommen, dass neue Aufgaben erkannt oder bestehende erneut weiter zerlegt werden. Der Projektleiter ist dafür verantwortlich, dass diese an die passende Stelle ins Backlog kommen.

Wichtig bei der Auswahl des funktionalen Umfangs für die anstehende Iteration ist die Betrachtung des Nutzens für den Kunden. Das geplante Iterationsergebnis muss einen Wert für den Kunden liefern und von Kundenseite prüfbar sein. Nur dann funktionieren die Rückkopplungsschleifen!

Bei dieser Betrachtung ist es nützlich, gleichzeitig das Iterationsziel zu schärfen. Das Iterationsziel ist nicht einfach nur die Menge aller Backlog-Einträge, die das Team für die Iteration ausgewählt hat. Damit wäre kaum ein Iterationsziel je wirklich erreicht. Irgendetwas kommt in Projekten immer dazwischen. Das Iterationsziel ist vielmehr die Beschreibung des Nutzens für den Kunden. Diese Sichtweise hilft beim Fokussieren der Entwick-

ler und unterstützt auch beim Schneiden der Backlog-Einträge für die einzelnen Featureteams.

Ein Iterationsziel kann also sein, dass ein Kunde mit unserer Software Bestellungen tätigen und verwalten kann. Diese Sicht hilft bei der Auswahl der passenden User Stories aus dem Backlog, und das Ziel kann auch mit einer gewissen Unabhängigkeit von der konkreten Umsetzung einzelner User Stories erreicht werden. Wir haben also zwei Vorteile: Jeder Entwickler kann in der konkreten Ausgestaltung seiner Tasks auf das Ziel hin arbeiten und wir gewinnen etwas Spielraum in der Zielerreichung bezogen auf die Umsetzung einzelner Stories. Wir können also das Iterationsziel erreichen, obwohl nicht alle für die Iteration ausgewählten User Stories erfolgreich umgesetzt wurden.

Die Planung I dauert je nach Teamgröße und Iterationsdauer zwei bis vier Stunden. Auch hier hilft eine Timebox beim Fokussieren. Jedoch bedarf es einer gewissen inhaltlichen Durchdringung der Anforderungen durch das Entwicklerteam, um mit der Planung II weitermachen zu können.

5.1.4 Planung II: die Arbeit strukturieren

Die Planung II findet nur noch im Entwicklerteam statt ohne die Fachexperten und meist auch ohne den Projektleiter. Dieses Meeting erfolgt am Taskboard. Die Entwickler zerlegen jetzt gemeinsam die der Iteration zugeordneten Anforderungen und technischen Aufgaben in einzelne Tasks. Neue Tasks werden sofort auf das Taskboard aufgenommen. Die Dauer dieses Planungsteils liegt zwischen drei und vier Stunden.

Der zeitliche Umfang eines Tasks sollte ca. einen Arbeitstag nicht überschreiten. Diese feine Zerlegung hilft also beim Strukturieren und Fokussieren und ermöglicht gleichzeitig ein tagesaktuelles Controlling der Tasks, ohne dass genauere Schätzungen erforderlich sind. Wir brauchen in der laufenden Iteration nur täglich die Anzahl der fertigen Tasks zu betrachten.

Leider findet in Planung II oft eine implizite Zuordnung der Tasks zu einzelnen Entwicklern statt. Dies ist jedoch nur im Einzelfall erwünscht. In der Regel sollten die Tasks von mehreren Entwicklern umsetzbar sein. Da auf dem Taskboard das Pull-Prinzip gilt, um einen gleichmäßigen Fluss der Tasks über das Taskboard zu erreichen, ist die Flexibilität der Entwickler der Performanzfaktor für den Projektfortschritt.

5.1.5 Commitment oder Forecast?

Wie ist die Auswahl der Backlog-Einträge für die anstehende Iteration zu bewerten? Auch hier hat in den letzten Jahren ein Umdenken stattgefunden. Aufgrund des Projektcharakters unserer Arbeit kann kein echtes Commitment, also eine Verpflichtung der Entwicklerteams, erfolgen. Dafür

kann selbst bei Iterationen unter einem Monat Dauer zu viel Unvorhergesehenes passieren. Die Folge war häufig eine eher defensive Planung durch die Entwickler, damit ihnen das Commitment nicht regelmäßig *um die Ohren fliegt*. Eine solche defensive Planung führt allerdings zu einer unnötig niedrigen Geschwindigkeit des Teams und widerspricht einem mutigen Vorgehen, wie es einer agilen Vorgehensweise entspricht.

Von daher wird z. B. im Scrum Guide [18] nur noch von *Forecast*, also Vorhersage bzw. Prognose, gesprochen. Das erscheint auf den ersten Blick schwächer, ermöglicht aber ein mutiges Vorgehen mit einer hohen Geschwindigkeit im Team. Daher verwenden wir auch in *APM* den Forecast.

5.2 Grooming oder Refinement?

Grooming und Refinement sind Begriffe aus Scrum, wobei Refinement dort Grooming abgelöst hat. Grooming bedeutet *Putzen* bzw. *Aufräumen*. Im Grooming organisieren wir unsere Arbeit im Projekt und bereiten die noch nicht ausreichend vorbereiteten, aber demnächst anstehenden Aufgaben vor, indem wir sie in verständliche Arbeitspakete zerlegen (Refinement: Verfeinerung, Zerlegung).

In *APM* sehen wir beides als sinnvoll und notwendig an und verwenden daher die beide Begriffe *Pflege* und *Verfeinerung*, um damit auszudrücken, dass es sich bei dieser Arbeit sowohl um ein Aufräumen als auch um eine Verfeinerung der Inhalte handelt. Beides erfolgt auf zwei Ebenen, im Backlog für das Produkt und auf dem Iterations-Taskboard.

5.2.1 Backlog-Pflege und Verfeinerung

Mit der Backlog-Pflege und Verfeinerung erfolgt die wesentliche Vorarbeit, damit Lookahead und Planung I effizient und in der geplanten Zeit durchgeführt werden können und die erwartete Ergebnisqualität geliefert werden kann. Daher wird es parallel zum Ende der vorhergehenden Iteration, aber vor Ergebnisreview und Retrospektive durchgeführt (Abb. 5-1).

In der Backlog-Pflege und Verfeinerung steht die aktuelle Qualität des Backlogs für das Produkt im Zentrum. Die für uns oberste relevante Ebene ist das aktuelle Release. Dann betrachten wir die zwei anstehenden Iterationen (Lookahead) und verteilen danach ggf. die einzelnen Aufgaben auf die Featureteams. Wir klären also, welche Fachlichkeit im Release genau umgesetzt werden soll, und schnüren Aufgabenpakete daraus. Diese verfeinern wir bis auf User-Story-Niveau, sodass die Teams auf diesem Level mit ihren Planungen aufsetzen können. Dabei werden automatisch notwendige Informationen gegeben oder zumindest als zu klären identifiziert sowie die Reihenfolge an die Prioritäten und Rahmenbedingungen angepasst.

An der Backlog-Pflege und Verfeinerung nehmen neben dem Projektleiter ggf. alle Teamprojektleiter und das Entwicklerteam bzw. Vertreter der Featureteams sowie ausgewählte Fachexperten teil. Eine einzelne Pflegesession dauert meist nur 60 bis 90 Minuten. Meist braucht es aber mehr als ein Meeting, um das Backlog auf den gewünschten Stand zu bringen. Zu einem gewissen Teil ist die Pflege des Backlogs eine Daueraufgabe des Projektleiters bzw. des Projektleitungsteams. Zur Vorbereitung der nächsten Iteration bedarf es expliziter Meetings mit Entwicklern und fachlichen Ansprechpartnern, um diese Aufgabe rechtzeitig und angemessen zu erfüllen. Gerade diese Meetings, die am Ende der vorhergehenden Iteration erfolgen, um die nachfolgende Iterationsplanung vorzubereiten, werden rechtzeitig vorher geplant und die Entwickler dazu eingeladen. Damit können alle relevanten Personen teilnehmen.

Wie viele dieser einzelnen Pflegesessions notwendig sind, hängt vom Zustand des Backlogs und der Anzahl paralleler Featureteams ab. Ist eine sehr aufwendige Backlog-Pflege und Verfeinerung abzusehen, kann auch bereits zwei Wochen vor Iterationsende mit den ersten Pflegesessions begonnen werden. Wichtig ist es dabei, so spät wie möglich die Backlog-Pflege durchzuführen, um den Anteil der Vorabplanung minimal zu halten. Ansonsten entsteht eine unnötige Verschwendung wertvoller Arbeitszeit.

5.2.2 Taskboard-Pflege und Verfeinerung

Das Aufräumen des Taskboards kann bei längeren Iterationen von drei bis vier Wochen ab etwa $2/3$ der Iteration wertvoll sein. Bei diesen Iterationslängen kann es schwerfallen und ist es kaum sinnvoll, in Planung II bereits alle Tasks für die Aufgaben aus dem Backlog herunterzubrechen. Insbesondere bei erkannten Abhängigkeiten zwischen einzelnen Aufgaben wäre dies eine Verschwendung der Arbeitszeit in Planung II.

Diese Defizite müssen allerdings zum Ende hin beseitigt werden, damit die Iteration mit gleichmäßig hoher Geschwindigkeit zu Ende gebracht werden kann. Und genau dazu dient die Taskboard-Pflege und Verfeinerung. Sie findet nur bei Bedarf statt und dauert meist etwa eine Stunde in Abhängigkeit von der verbleibenden Iterationsdauer. Diese Arbeiten werden von allen Entwicklern eines Teams durchgeführt. Parallel arbeitende Featureteams führen die Taskboard-Pflege und Verfeinerung voneinander unabhängig durch.

5.3 Daily Standup – Synchronisation im Team

Beim Daily Standup treffen sich alle Mitglieder eines Entwicklerteams untereinander, um den gegenseitigen Status abzugleichen und Probleme zu

identifizieren bzw. bekannt zu machen. Die Entwickler synchronisieren dabei ihre Aktivitäten und gleichen ihre individuellen Pläne bis zum nächsten Daily Standup ab. Dabei orientieren sich alle Teilnehmer bei ihren kurzen Statements an dem aus Scrum bekannten Fragenkatalog:

1. Was habe ich seit dem letzten Daily Standup getan?
2. Was plane ich, bis zum nächsten Daily Standup zu tun?
3. Welche konkreten Probleme sind dabei aufgetreten oder habe ich?

Das Daily Standup findet täglich stets zur gleichen Zeit statt und gibt damit dem Arbeitstag einen Rhythmus. Es dauert nur 10 bis maximal 15 Minuten. Mehrere parallele Featureteams sollten ihre eigentlich voneinander unabhängigen Daily Standups zeitlich nah beieinander durchführen, damit sie bei nachfolgenden Treffen zur Abstimmung zwischen den Teams auf dem aktuellen Stand sind.

Eine Falle bei den Daily Standups ist, dass dabei gerne begonnen wird, Probleme zu diskutieren. Dafür ist dieses Treffen allerdings nicht gedacht. Der zeitliche Rahmen wäre auch zu kurz und die Anzahl der Teilnehmer zu groß. Ein solches Vorgehen wäre also wiederum Verschwendung von wertvoller Arbeitszeit. Der agile Coach als Moderator hat darauf zu achten, dass das nicht passiert, und ggf. dafür zu sorgen, dass nachfolgende Treffen zur Problemlösung mit den beteiligten Personen anberaumt werden.

5.4 Synchronisation von mehreren Featureteams

Die *Team-Sync-Meetings* dienen der Koordination von mehreren Teams in einem Projekt. Wir dehnen damit das Konzept des Daily Standup auf teamübergreifende Themen aus.[1] Sie sind also nur notwendig, wenn mehrere parallele Featureteams im Projekt arbeiten. Für jeden Themenbereich, der einer Synchronisation zwischen den Featureteams bedarf, gibt es ein eigenes Team-Sync-Meeting. Diese Meetings sind voneinander unabhängig und haben in der Regel auch eigene Rhythmen und zeitliche Dauer.

Typische Themen für Team-Sync-Meetings sind Architektur und Test, um eine zueinander passende und weitgehend aufeinander abgestimmte Architektur im ganzen Projekt zu erzielen sowie aufeinander abgestimmte Systemtests und daraus abgeleitete Tests für die einzelnen Featureteams. Auch die agilen Coaches halten regelmäßig Team-Sync-Meetings ab, um das Lernen zwischen den Teams und untereinander zu fördern und so die Entwicklungsprozesse kontinuierlich weiterentwickeln zu können.

Der Rhythmus für die jeweiligen Team-Sync-Meetings kann zwischen täglich, wöchentlich und einmal pro Iteration schwanken, bleibt jedoch für

[1] Das Konzept der Team-Sync-Meetings basiert auf der Idee des Scrum of Scrums.

jedes Team-Sync-Meeting gleich. Er kann ggf. an veränderte Gegebenheiten angepasst werden. Es nehmen jeweils ein, maximal zwei Vertreter aus jedem Featureteam daran teil. Diese Rollen können je nach der Qualifikationsverteilung in den Teams wechseln oder eher fest einzelnen Personen zugeordnet werden. So ist z. B. die Tester-Rolle meist festgelegt, wohingegen die Architekten-Rolle häufiger wechselt und von möglichst vielen Teammitgliedern wahrgenommen werden sollte.

5.5 Iterationsende: Review und Retrospektive

Am Ende einer Iteration erfolgen noch zwei Meetings, die essenziell für den Projekterfolg sind: das Ergebnisreview und die Retrospektive.

5.5.1 Ergebnisreview

Das Ergebnisreview ist eine weitgehend offene Veranstaltung, bei der das Inkrement Vertretern der Kundenseite und des Managements vorgestellt wird. Ziel des Reviews ist es, Rückmeldung zum aktuellen Stand zu bekommen, um die Entwicklung optimal an die aktuellen Anforderungen und Bedürfnisse anzupassen.

Das Review hat damit eher Demonstrationscharakter und darf auf keinen Fall mit der Freigabe oder Abnahme verwechselt werden. Freigabe- und Abnahmeprozesse sind in der Regel vor dem Review für das Inkrement abgeschlossen. Während des Reviews werden nur qualitätsgesicherte Funktionen gezeigt. Die Features werden vom Projektleiter und den Entwicklern, die in der Iteration für bestimmte neue Features verantwortlich waren, vorgestellt. So können auch aufkommende Fragen sofort kompetent beantwortet werden.

Weil das Review eher eine Demonstration des Projektfortschritts als eine Prüfung darstellt und möglichst viele verschiedene Stakeholder anwesend sind, um ihre Rückmeldungen zu geben, ist es wichtig, dass es entsprechend sorgfältig vorbereitet wird:

- Der oder die Demonstrationsrechner sind vorbereitet, konfiguriert und eingeschaltet.
- Die Software ist in der korrekten Version installiert.
- Der Demonstrationsrechner ist an den Beamer angeschlossen.
- Die Listen über die neue Funktionalität sind im Zugriff und optisch ansprechend aufbereitet, sodass die Demonstration systematisch durchgeführt werden kann.
- Ein Flipchart zum Festhalten von wertvollen Rückmeldungen ist gut sichtbar aufgestellt und Stifte liegen bereit.

Meist wird die Vorbereitung reihum durch einen Entwickler geleistet und überschreitet kaum eine halbe Stunde Dauer. Das Review selbst dauert meist zwischen einer und drei Stunden, je nach Umfang des Projekts und Engagement der Stakeholder.

Falls es im Review auf einigen Testrechnern auch zum Ausprobieren der aktuellen Entwicklungsversion inklusive des neuen Inkrements kommen sollte, so sind die entsprechenden Rechner ebenfalls einheitlich vorbereitet und einsatzfähig. Wichtig ist, dass das Ausprobieren der Software, z. B. durch spätere Anwender, *nach* der Demonstration der neuen Features durch den Projektleiter und die Entwickler erfolgt. Es steht zusätzlich jeweils ein Entwickler pro Testrechner als Ansprechpartner bereit, um Fragen zu klären [22]. Das *begleitete Ausprobieren* kann gut in einer Timebox von 30 Minuten ablaufen.

5.5.2 Retrospektive

Im Gegensatz zum Review ist die Retrospektive eine rein projektteaminterne Veranstaltung. Hier geht es um eine sehr ehrliche und offene Erörterung der aktuellen Abläufe und Probleme im Projekt. Die Retrospektive ist das Schlüsselelement zur permanenten Verbesserung der Entwicklungsprozesse im laufenden Projekt.

Wir haben bereits die Analogie benutzt, dass eine Iteration als kleines Projekt betrachtet werden kann. Die Retrospektive entspricht dann den traditionellen *Lessons Learned* zum Projektabschluss. Der wesentliche Vorteil der Retrospektiven am Ende jeder Iteration liegt darin, dass die darin gewonnenen Erkenntnisse bereits ab der nächsten Iteration dem laufenden Projekt zugute kommen.

Jedes Featureteam führt seine eigene Retrospektive durch. Bei Bedarf, aber spätestens nach einem externen Release erfolgt eine teamübergreifende Gesamtprojektretrospektive. Jede Retrospektive dauert typischerweise ca. drei Stunden. Die Grundstruktur einer Retrospektive orientiert sich an vier Fragenblöcken [6]:

- Was ist in der Iteration bzw. im Betrachtungszeitraum passiert?
- Was lief besonders gut? Was wollen wir feiern? Was war schwierig?
- An welchen Stellen können wir uns Verbesserungen vorstellen?
- Was wollen wir konkret ab sofort anders machen?

Wichtig ist es dabei, alle Mitglieder des Featureteams einzubinden. Das Ziel einer Retrospektive ist, Ursachen zu ergründen. Es werden also nicht nur Symptome bearbeitet. Das braucht seine Zeit!

Der agile Coach moderiert die Retrospektive und wählt dabei immer wieder unterschiedliche Moderationsmethoden, um jede Retrospektive ab-

wechslungsreich und anregend zu gestalten. Der Ablauf einer Retrospektive orientiert sich an den folgenden fünf Schritten [6]:

1. Ankommen und einen Rahmen für die Retrospektive schaffen.
2. Information über Ereignisse in der Iteration sammeln.
3. Erkenntnisse über Hintergründe, Ursachen und Wirkungen produzieren.
4. Im Team entscheiden, was gemacht wird.
5. Die Retrospektive abschließen.

Die Retrospektive ist wohl die anspruchsvollste Veranstaltung, die der agile Coach moderiert.

6 Das Rollenmodell

Als Nächstes betrachten wir die Rollen in *APM*. Aufgrund der Skalierbarkeit und Anpassungsmöglichkeit an konkrete Rahmenbedingungen in Organisationen gibt es ein Grundmodell als Ausgangspunkt und eine exemplarische Ausbaustufe zu Ihrer Orientierung für Ihr eigenes Rollenmodell.

6.1 Ein agiles Rollenmodell

Es ist ohne Weiteres möglich, *APM* nur mit den Scrum-Rollen durchzuführen. *APM* entspricht dann in etwa einem durch zusätzliche Techniken erweiterten Scrum. Falls der Bedarf nach Ansprechpartnern für bestimmte Aspekte entsteht, kann das Entwicklungsteam dafür in Anlehnung an das nachfolgende Rollenmodell selbstorganisiert aus seinen Reihen Personen dafür benennen.

Der besondere Wert von *APM* entsteht dann, wenn es darum geht, ein agiles Vorgehen bestimmten Rahmenbedingungen anzupassen. Wenn z. B. der Anteil an Universalisten in den Featureteams gering ist oder Regeln bzw. Vorschriften den Einsatz einer spezialisierten Qualitätssicherung und von Testern notwendig machen, kann es weiterhin sinnvoll sein, Spezialisten im Team zu haben. Das wird dann meist am Rollenmodell sichtbar.

Allerdings besteht bei jedem Rollenmodell, das über die drei Scrum-Rollen hinausgeht, die Gefahr, sich damit von der Umsetzung der agilen Werte und Prinzipien zu entfernen. Ein agiles Rollenmodell unterliegt daher einer regelmäßigen Veränderung und Anpassung an die aktuellen Gegebenheiten. Ein Rollenmodell, wie das hier vorgestellte, kann daher nur als Ausgangspunkt für die Teams gelten. Die weitere konkrete Ausprägung entwickelt sich im Rahmen der Sebstorganisation der Teams weiter.

Diese Dynamik wird umso größer werden, je flexibler die Teammitglieder einsetzbar sind. Das Rollenmodell stellt sicher, dass alle relevanten Aufgaben und Verantwortungen dem Team bekannt sind und wahrgenommen werden. Im Projektverlauf werden sich in wirklich agilen Teams immer wieder Veränderungen ergeben, die Rollen überflüssig machen oder neu hinzukommen lassen. Auch kann eine konkrete Person eine Rolle über längere Zeit wahrnehmen, während andere Rollen regelmäßig getauscht werden.

Zusammenfassend erfüllt das *APM*-Rollenmodell daher zwei Aufgaben. Zum einen stellen wir darüber sicher, dass alle wichtigen Aufgaben und Verantwortungen in den Teams wahrgenommen werden. Zum anderen stellt es einen Ausgangspunkt dar, um mit einem konkreten Rollenmodell ein Projekt aufsetzen und starten zu können. Danach unterliegt es der Selbstorganisation der Teams, die das Projekt umsetzen.

6.2 Die Grundlagen des APM-Rollenmodells

Ein minimales *APM*-Rollenmodell besteht aus den drei Rollen Projektleiter, Entwickler und agiler Coach. In dem Minimalmodell in Bezug zu den Scrum-Rollen entspricht der Projektleiter dem Product Owner, die Entwickler dem Entwicklungsteam und der agile Coach dem Scrum Master (Abb. 6-1).

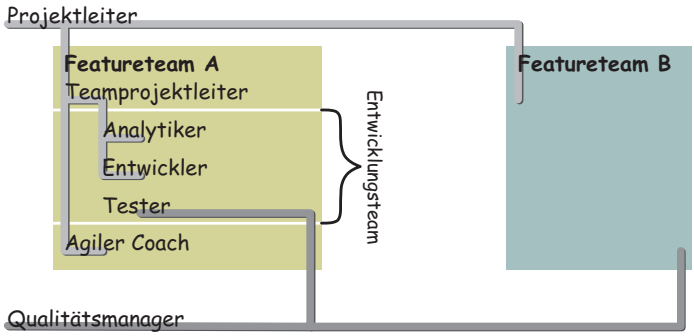

Abbildung 6-1: Ein einfaches *APM*-Rollenmodell für kleine Projekte

Warum benutzen wir in *APM* nicht einfach die Scrum-Rollen? Weil in *APM* entgegen Scrum sehr wohl spezialisierte Rollen in der Entwicklung zum Einsatz kommen können. Eine typische Rolle ist z. B. der Tester oder der Analytiker für komplexere Rahmenbedingungen (Abb. 6-1). Der Tester ist in der Regel einem Qualitätsmanager unterstellt und der Analytiker kennt sich fachlich und in der Systemanalyse bestens aus, wird aber selten selbst Software entwickeln.

Wir erkennen an diesem ersten Beispiel in Abbildung 6-1 auch die Aufteilung in mehrere Featureteams mit jeweils einem eigenen Teamprojektleiter und einem agilen Coach. Über allem steht dann der Projektleiter auf meist gleicher Hierarchieebene wie der Qualitätsmanager. Es hat sich bereits eine differenzierte Struktur gebildet, die oft weiterentwickelt wird. Häufig kommen noch zwei weitere Rollen innerhalb des Entwicklungsteams

ins Spiel: der Architekt und der Testautomatisierer (Abb. 6-2). Mit dieser Rollenmodellvariante können bereits viele Großprojekte aufgesetzt und Featureteams gebildet werden, ohne die bisherigen Arbeitsplatzbeschreibungen auszuhebeln. Gehen wir die Rollen im Detail durch. Worin liegt ihr Wert für das Projekt?

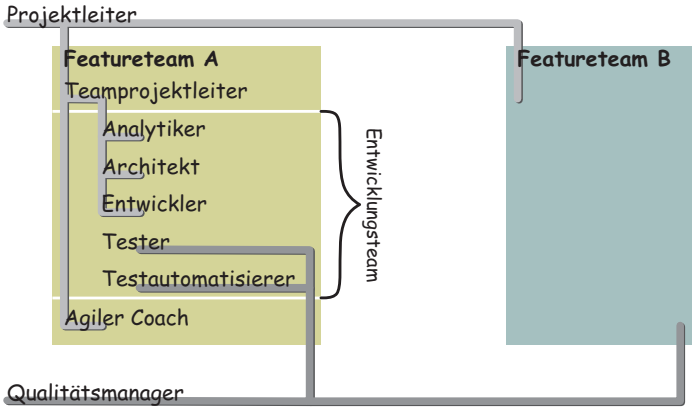

Abbildung 6-2: Das *APM*-Rollenmodell skaliert mit dem Projekt und den Rahmenbedingungen.

Projektleiter: Hier liegt die Gesamtverantwortung für das Projekt bzw. das Projektergebnis. Die Steuerung erfolgt primär über das Backlog für das Produkt. Der Projektleiter ist auch die oberste Entscheidungsstufe bei Konflikten zwischen den Featureteams. Der Projektleiter und seine Teamprojektleiter für die Featureteams bilden das Projektleitungsteam, in dem alle für das Backlog relevanten Entscheidungen getroffen werden. Dazu führt das Projektleitungsteam regelmäßig entsprechende Team-Sync-Meetings durch, in denen als eine der wichtigsten Aufgaben die Verteilung der Backlog-Einträge auf die Featureteams erfolgt (Abschnitt 5.4).

In der Zusammenarbeit mit dem Qualitätsmanager, dessen Rolle weiter unten erläutert wird, werden für das Projekt relevante Aspekte aus dem Qualitätsmanagement erarbeitet und integriert. Organisatorisch gibt es zwei Varianten. Entweder sind beide in der Hierarchie gleichgestellt oder der Projektleiter ist dem Qualitätsmanager überstellt. Bei Konflikten zwischen Projektleiter und Qualitätsmanager muss im ersten Fall ggf. ein übergeordneter Programmmanager oder eine andere Führungskraft hinzugezogen werden. Im zweiten Fall gilt es, in der Projektvorbereitung gemeinsam eine Regelung für solche Fälle zu erarbeiten und zu implementieren.

Teamprojektleiter: Hier liegt die Teilproduktverantwortung für die Umsetzung der Features des jeweiligen Featureteams. Gemeinsam mit den anderen Teamprojektleitern und dem Projektleiter werden die entsprechenden Abstimmungen für das Backlog getroffen. Der Teamprojektleiter ist dabei für das Featureteam-Backlog verantwortlich, also für die Aufgaben aus dem Backlog, die seinem Team für die nächsten beiden Iterationen zugeteilt sind. An dieser Zuteilung ist jeder Teamprojektleiter beteiligt. Danach kann er die Reihenfolge, Pflege und notwendige zusätzliche Verfeinerungen der Aufgaben in seinem zugeordneten Teil, dem Featureteam-Backlog, eigenständig verantworten und in Abstimmung mit dem Entwicklungsteam durchführen.

Analytiker: In komplexen Projekten mit mehreren Featureteams wird es für die Teamprojektleiter schwierig, allen analytischen Aufgaben nachzukommen, um die Fragen der Entwickler zeitnah beantworten zu können. Auch haben sie oft nicht das ausreichende analytische Können und fachliche Prozesswissen, um tief genug in die Details gehen zu können. Der Analytiker übernimmt daher diese Aufgaben, arbeitet einerseits bei der Feature-Backlog-Pflege dem Projektleiter zu und übernimmt andererseits die Detaildiskussionen und fachlichen Vorabnahmen mit den Entwicklern. Dazu braucht er fachliches und methodisches Know-how, aber nur wenig bis gar kein technisches Implementierungswissen.

Architekt oder Architecture Owner: In der Softwareentwicklung lässt sich zwischen übergeordneten Architekturaspekten und eher kleinteiligeren Design- und Implementierungsthemen unterscheiden. Die Softwarearchitektur zeichnet sich dabei dadurch aus, dass sie die später nur schwer änderbaren und damit teuren Aspekte betrifft. Von daher kann es in großen Projekten mit mehreren Teams sinnvoll sein, für die Softwarearchitektur eine eigene dafür verantwortliche Rolle zu besetzen.

Die Architekten aus den Featureteams stimmen sich in übergreifenden Team-Sync-Meetings zu gemeinsamen, teamübergreifenden Softwareaspekten ab und bringen die Ergebnisse in ihre Teams. Die Architekten tragen die Verantwortung dafür, dass eine durchgängige Architektur von den Featureteams implementiert wird.

Die Architekten implementieren in der Regel mit den Entwicklern gemeinsam das Produkt. Ein Architekt ist also stets auch Entwickler. Gleichzeitig stehen sie teamintern für die anderen Entwickler als technische Coaches und Ansprechpartner bereit. Der Architekt wird auch dem Teamprojektleiter als technischer Ansprechpartner dienen, z. B. um Refactoring-Zyklen im Team festzulegen.

Entwickler: Bei den Entwicklern liegt die Verantwortung für die Aufwandsschätzung und Implementierung der Anforderungen. Letztendlich sind sie auch verantwortlich für die Produktqualität, obwohl sie häufig nur bis zur Komponentenebene (Abb. 6-3) umfassend und automatisch tes-

ten können. Die Entwickler arbeiten testgetrieben und kleinteilig. Eine konkrete Aufgabe wird so lange in kleinere Tasks zerlegt, bis jeder Task zumindest an einem Tag umsetzbar ist.

Tester: Über die Rolle des Testers wird die Verantwortung für die höheren Teststufen, d. h. Integrations- und Systemtest, sowie die Begleitung von Kundenvertretern bei Abnahmetests sichergestellt (Abb. 6-3). Damit ein Tester seinen Aufgaben nachkommen kann, braucht er fachliche Informationen von der Kundenseite und vom Projektleiter und, falls die Rolle besetzt ist, vom Analytiker zeitgleich zu den Entwicklern. Seine Aufgabe ist die Erstellung und Pflege übergreifender, testmethodisch aussagekräftiger Abflauftests für das Produkt bzw. Inkrement. Damit bringt der Tester eine Sicht auf das Produkt ein, die häufig von den Entwicklern nicht ausreichend wahrgenommen werden kann. Gleichzeitig gibt er mit der frühzeitigen Bereitstellung von Testszenarien den Entwicklern wertvolle Informationen für ihre Entwicklung [5].

Durch das iterative Vorgehen entstehen mit jeder Iteration neue Regressionstests, über die sich mit fortschreitendem Projektverlauf ein hoher Druck auch zur Testautomatisierung auf der Integrations- und Systemtestebene aufbaut. Wichtig ist, dass solche automatisierten Ablauftests in die Build-Umgebung integriert werden, um automatisch mitzulaufen und jederzeit den Entwicklern wertvolle Informationen über die Qualität des aktuellen Builds rückzukoppeln.

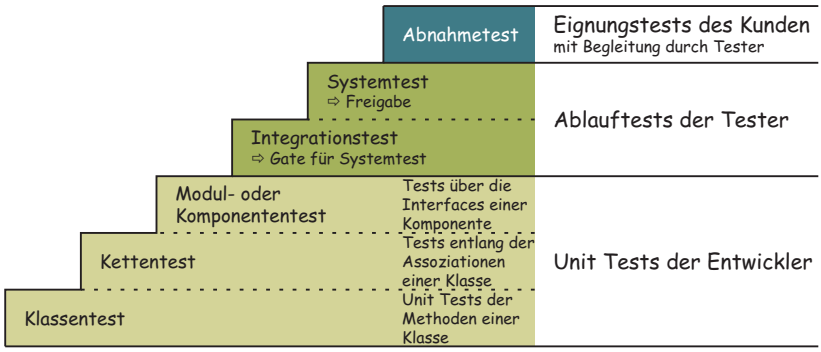

Abbildung 6-3: Überblick über die gängigsten Teststufen [20]

Testautomatisierer: Da die Tester selten in der Lage sind, ihre stabilen Regressiontests selbst zu automatisieren, übernehmen entweder Entwickler oder dedizierte Testautomatisierer diese Aufgabe. Testautomatisierer sind also auch Entwickler, die jedoch mit einer eigenen Entwicklungsumgebung Systemtests meist über simulierte User-Interface-Interaktionen programmieren und pflegen. Je nach Umfang dieser Ar-

beit kann ein Testautomatisierer einem oder zwei Featureteams zugeordnet sein. Die Testautomatisierer pflegen also ihr eigenes Testautomatisierungsprojekt im Rahmen des Gesamtprojekts.

Agiler Coach: Ähnlich einem Scrum Master begleitet der agile Coach sein Featureteam durch den agilen Entwicklungsprozess nach *APM*. Er moderiert die Meetings, achtet darauf, dass die agilen Werte in der täglichen Arbeit ausreichend berücksichtigt werden, und sorgt für die Einhaltung der *APM*-Regeln und Methodik. Wie ein Scrum Master kümmert er sich auch um die Beseitigung von Hindernissen und ist wie eine Art agiler Change Agent für die kontinuierliche Weiterentwicklung der Vorgehensweise und der Teamentwicklung verantwortlich [1].

Meist ist es sinnvoll, den agilen Coach auf gleicher hierarchischer Ebene wie die Teamprojektleiter anzusiedeln. So kann der agile Coach seiner Prozessverantwortung auch kraftvoll nachkommen, wobei er sich durch eine *konstruktive* Konfliktfähigkeit auszeichnet. Gegebenenfalls von einem Featureteam nicht selbst lösbare Konflikte zwischen Teamprojektleiter und agilem Coach sollten vom Projektleiter auf der nächsthöheren Hierarchieebene geklärt werden.

Qualitätsmanager: Der Qualitätsmanager ist eine Rolle, die meist nur in einem Umfeld mit besonderen Anforderungen an die Qualität der Software und der Erstellungsprozesse notwendig ist. Er verantwortet die Richtungsentscheidungen der Qualitätssicherung für das gesamte Unternehmen oder eine Produktfamilie derart, dass er den Testern einen organisatorischen Rahmen gibt, innerhalb dessen sie selbstorganisiert arbeiten können. Er plant und steuert gemeinsam mit den Testern die Teststrategie und entwickelt dafür aussagekräftige Qualitätsmetriken. Seine Aufgabe ist es dabei im Wesentlichen, den Überblick über die Testaktivitäten in den einzelnen Teams und im zugeordneten Projekt zu halten und koordinierend tätig zu werden, wenn Bedarf erkannt wird. Die Tester sind in alle Entscheidungsfindungen eingebunden und setzen die Vorgaben innerhalb ihrer Featureteams um.

Der Qualitätsmanager sorgt auch dafür, dass regelmäßige Team-Sync-Meetings der Tester stattfinden, damit aus den einzelnen Testfällen der Tester ein sinnvoller und angemessener Gesamtsystemtest entsteht. Gleichzeitig ist er auch Ansprechpartner und methodischer Coach für die Tester in den Featureteams. Die konkrete Umsetzung dieser Rolle kann z. B. als Leiter der *Community of Practice* für das Testen erfolgen, zu der alle Tester aktiv beitragen.

Der Qualitätsmanager ist *Sparringspartner* für den Projektleiter und achtet auf eine ausreichende Berücksichtigung qualitativer Aspekte im Projekt. Dabei ist es meist sinnvoll, die Rolle Qualitätsmanager in der Hierarchie der Rolle des Projektleiters gleichzusetzen, damit der Qualitätsmanager seine Aufgaben eigenverantwortlich erledigen kann.

Zusätzlich zu diesen Rollen kann es in größeren Firmen analog zum *Qualitätsmanager* auch einen *Architekturmanager* geben, der die Community of Practice für die Architekturthemen organisatorisch leitet, zu der alle Teams über ihre Architekten bzw. Architecture Owner aktiv beitragen.

Die Community of Practice (CoP) als Mittel zum gemeinsamen Lernen und Weiterentwickeln von übergreifenden Ideen und Konzepten geht auf Jean Lave und Étienne Wenger zurück [11]. Eine CoP bezeichnet eine praxisbezogene Gemeinschaft von Personen, die sich mit ähnlichen Themen befassen und in dieser Gruppe Erfahrungen austauschen und voneinander lernen wollen.

7 Das Phasenmodell

Obwohl die Iterationen von ihrer Struktur her stark an Scrum angelehnt sind, sieht *APM* ein zusätzliches Phasenmodell vor, in das die Iterationen eingepasst werden. Genau genommen sind es sogar zwei Phasenmodelle, die auf die Produkterstellung bis zum Betrieb (Abb. 7-1) und die Produktpflege und -erweiterung im laufenden Betrieb (Abb. 7-2) ausgelegt sind.

Die wesentliche Erweiterung im *APM*-Phasenmodell ist die explizite Berücksichtigung einer kurzen Architekturphase bzw. der regelmäßige Einbau von Refactoring-Zeiten für umfangreichere Redesigns der Software. *APM* ist damit wesentlich stärker als Scrum oder XP eine architekturzentrierte Vorgehensweise.

7.1 Produkterstellung bis zum vollen Betrieb

In diesem Abschnitt befassen wir uns mit Projekten, die neu aufgesetzt werden. Ihre Dauer liegt zwischen 9 und 18 Monaten, und es ist eine Reihe von Risiken zu adressieren. Größere Projekte werden in Teilprojekte zerlegt, die wieder in das zeitliche Raster fallen und sequenziell abgearbeitet werden. Betrachten wir die einzelnen Phasen nacheinander (Abb. 7-1):

Vorbereitung: Das anstehende Projekt wird in dieser Phase vorbereitet. Sie dauert typischerweise eine halbe bis eine Iterationslänge, übersteigt aber nie die Dauer von zwei Iterationen. Das Ziel der Vorbereitung ist es, die grundlegenden Entscheidungen über das Projekt zu treffen, also ob und in welchem Rahmen es durchgeführt werden soll. Hierbei spielen sowohl betriebswirtschaftliche wie technische und auch organisatorische Aspekte eine Rolle. Von daher sind die beteiligten Personen in dieser Phase bereits sehr heterogen aufgestellt.

Im Einzelnen gilt es, die Produktvision so weit zu entwickeln, dass ein erstes Backlog für das Produkt inklusive qualitativer Anforderungen initial erstellt werden kann. Des Weiteren kommen eine initiale Stakeholder-Analyse, erste Risikobetrachtungen, eine erste Makroschätzung mit anschließender Wirtschaftlichkeitsbetrachtung sowie

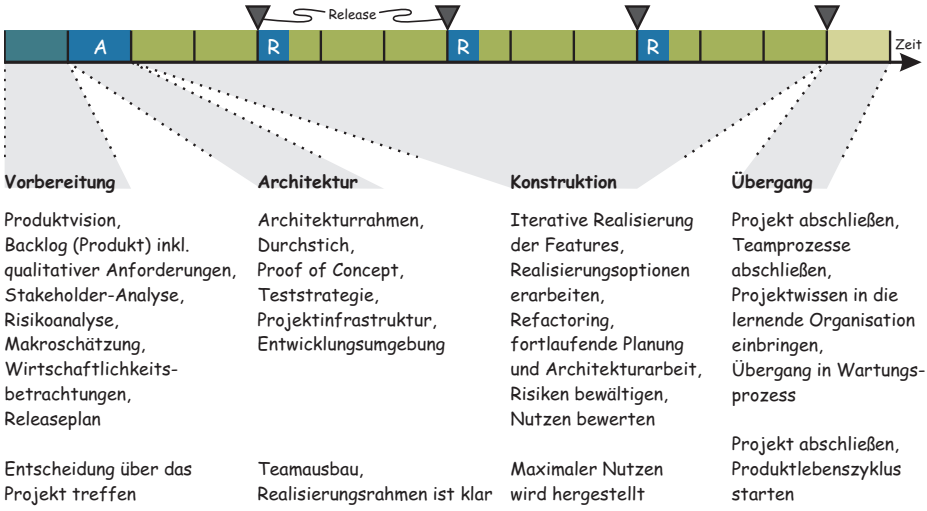

Abbildung 7-1: Das *APM*-Phasenmodell für die Projektdurchführung bzw. die Produktentwicklung (A: Architekturphase, R: umfassendes Refactoring)

ein erster Releaseplan hinzu. Auf dieser Basis können jetzt die Entscheidungen über das Projekt getroffen werden.

Aufgrund der vielfältigen Anforderungen hat der Projektleiter hier bereits die Zügel für das Projekt fest in der Hand und ist entsprechend von den Verantwortlichen in der Organisationsstruktur eingesetzt worden. Der Projektleiter stellt dann sein Kernteam zusammen, mit dem gemeinsam die technischen und organisatorischen Fragen geklärt werden. Zusätzlich arbeiten von Kundenseite Fachexperten und Entscheider bereits hier intensiv zusammen, damit sowohl der Anforderungsüberblick und die erste Anforderungszerlegung als auch die Wirtschaftlichkeitsbetrachtungen erfolgen können.

Es ist hilfreich, wenn im Kernteam bereits alle später im Projekt erforderlichen Rollen vertreten sind. So werden unnötige Verzögerungen vermieden und das spätere Kernteam kann schon zu einem frühen Zeitpunkt die konkrete Zusammenarbeit üben. Mit der Vorbereitung ist dann auch die Kernteambildung abgeschlossen.

Architektur: In dieser Phase geht es darum, das Kernteam zur vollen Projektteamstärke auszubauen und den Realisierungsrahmen zu klären. Dazu werden gezielt technische Durchstiche durch die geplanten Architekturschichten vorgenommen, ggf. ein Proof of Concept erstellt und so der Architekturrahmen gebildet. Des Weiteren wird die Projektinfrastruktur mit der Entwicklungsumgebung aufgesetzt. Parallel zu diesen

Tätigkeiten wird die Teststrategie entwickelt und notwendige Automatisierungswerkzeuge werden in die Projektumgebung integriert.

Alle diese Tätigkeiten sind nicht abschließend, sondern initial gemeint. Es wäre ein großes Missverständnis, wenn es das Ziel wäre, am Ende der Architekturphase die Architektur und Infrastuktur bereits komplett fertig erstellt zu haben. In einem agilen, iterativen Vorgehen gilt es, nur insoweit vorzuarbeiten, dass die nachfolgende Konstruktionsphase ohne Probleme beginnen kann. Noch fehlende Architektur und zusätzlich notwendige Anpassungen an der Projektinfrastruktur erfolgen bei Bedarf in der Konstruktionsphase.

Die Dauer der Architekturphase ist auf eine Iterationslänge begrenzt, um wirklich nur auf die notwendigen Aspekte zu fokussieren. In sehr großen, verteilten Projekten mit weit über drei Featureteams und besonders aufwendigen Abstimmungsverfahren kann diese Phase auf den maximal sinnvollen Rahmen von zwei Iterationslängen ausgedehnt werden.

Konstruktion: In dieser längsten Phase im Projekt gilt es, den maximalen Nutzen für den Kunden herzustellen. Nach jeweils ein bis vier Iterationen erfolgt ein evtl. auch nur simuliertes, internes oder besser bereits externes Release. Die Realisierung der Features erfolgt nach der Ordnung im Backlog in inkrementell-iterativer Weise. Die Risiken werden dabei fortlaufend untersucht und eingeschränkt und ggf. werden dazu Realisierungsoptionen erarbeitet.

Die Planung und Architektur erfolgen fortlaufend in den Iterationen. Dazu wird regelmäßig der Nutzen der Features neu bewertet und die Detaillierung der anstehenden Features rechtzeitig, aber so kurz wie möglich vor der Iteration, in der sie umgesetzt werden sollen, vorangetrieben. Die Iterationsdauer liegt zwischen zwei und vier Wochen und überschreitet auch bei sehr großen, verteilten Projekten mit starkem Abstimmungsaufwand nicht eine Maximallänge von sechs Wochen.

Um auch größere Architekturthemen angehen und bewältigen zu können, wird direkt nach jedem internen oder externen Release ein umfassendes Refactoring eingeplant. Es dauert die Hälfte einer Iteration. Dafür werden im Backlog entsprechende Refactoring-Aufgaben gesammelt und zum Refactoring nach oben priorisiert. Liegt weniger oder kein Bedarf für ein Refactoring seitens der Entwickler und der Architekten vor, kann das Refactoring entsprechend verkürzt werden oder ganz entfallen. Es schließt sich dann eine normale Iteration an das Release an.

Übergang: Im Übergang wird das Projekt abgeschlossen und der Produktlebenszyklus gestartet. Damit findet auch der Übergang in den Softwarebetrieb und die Wartung mit seinem modifizierten Phasenmodell statt.

Neben dem Projekt ist auch der Teamprozess abzuschließen. Das gesammelte Projektwissen geht spätestens jetzt in die lernende Organi-

sation über. Eine kleine Projektfeier mit allen Projektteammitgliedern und den ggf. neuen Mitarbeitern im Softwarebetrieb und der Wartung machen diesen Übergang auch für jedermann deutlich sichtbar.

7.2 Softwarebetrieb und Wartung

Der Begriff *Softwarebetrieb* bezeichnet die zusätzlich zur Wartung für den weiteren erfolgreichen Betrieb notwendigen Anpassungen und Erweiterungen mit Projektcharakter, aber nicht die administrativen oder infrastrukturellen Maßnahmen zum Betreiben der Software. Unter Wartung verstehen wir die Modifikation eines Softwareprodukts nach seiner Auslieferung, um Fehlerzustände zu korrigieren, Merkmale wie z. B. die Performanz zu verbessern oder das Produkt für eine andere Umgebung zu adaptieren [7].[1]

Typischerweise setzt sich das Softwarebetriebs- und Wartungsteam aus einigen erfahrenen Mitarbeitern aus dem ursprünglichen Projekt und Entwicklern, die neu ins Team hinzukommen, zusammen. Grundsätzlich ist der Anspruch an die Fähigkeiten dieser neu hinzukommenden Entwickler sehr hoch, da sie sich schnell in eine Menge Code anderer Entwickler einzuarbeiten haben. Außerdem können die zwar von der Dauer her eher kurzen Softwarebetriebsprojekte inhaltlich besonders anspruchsvoll sein. So geht es z. B. darum, komplett neue Anforderungsbereiche in die Software zu integrieren und an vielen qualitativen Aspekten wie Durchsatz oder Laufzeiten deutliche Optimierungen vorzunehmen. Betrachten wir nun die Phasen im Einzelnen (Abb. 7-2).

Jahresplanung: In einem festen Rhythmus von ca. 12 Monaten erfolgt die Jahresplanung im Sinne einer Projektvorbereitung. Diese Jahresplanung läuft parallel zur letzten Iteration des letzten Planungszyklus. Hier werden die größeren anstehenden Themen strukturiert und aufbereitet, sodass sie auch zeitnah angegangen werden können. Es können hier alle grundsätzlichen Entscheidungen über die Weiterentwicklung des Produkts getroffen werden.

Dazu wird die Produktvision weiterentwickelt, das Backlog für das Produkt grundsätzlich überarbeitet und die anderen Artefakte aus der Stakeholder-Analyse und dem Risikomanagement grundsätzlich gepflegt und aktualisiert. Für die größeren Projektthemen erfolgt eine erste Makroschätzung, die entsprechenden Wirtschaftlichkeitsbetrachtungen und eine erste Zuordnung auf die Releases am Ende jeder Iteration, sodass ein aktueller Releaseplan entsteht.

[1] Diese Definition ist konform zum IEEE-Standard 1219.

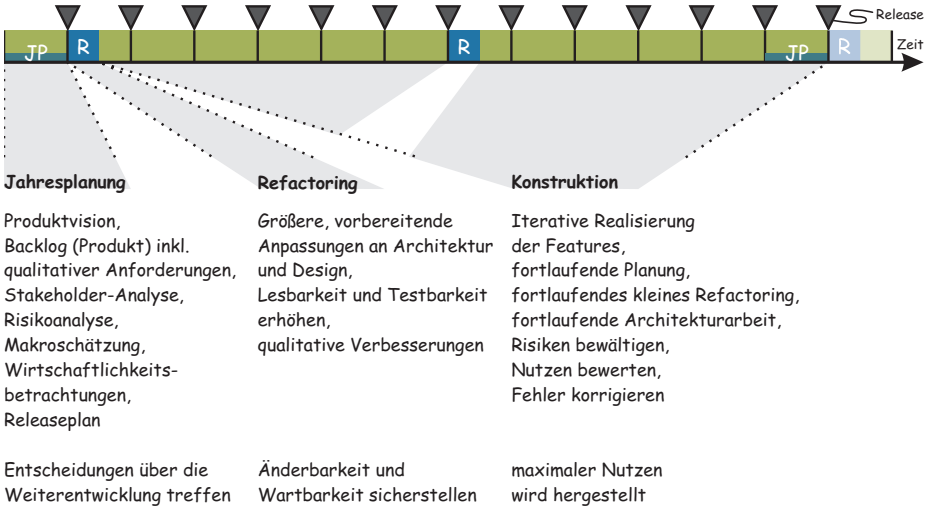

Abbildung 7-2: Das *APM*-Phasenmodell für den Softwarebetrieb und die Produktwartung (JP: Jahresplanung, R: umfassendes Refactoring)

Konstruktion mit fortlaufenden Releases: Die aktuellen Wartungsaufgaben und die anstehenden Aspekte aus den Softwarebetriebsprojekten werden in einzelnen Iterationen erarbeitet und am Ende jeder Iteration mit einem Release ausgeliefert. Falls kein *echtes* Release erfolgen kann, werden dennoch die Releases so weit wie technisch und operativ möglich simuliert und dann ein *internes* Release durchgeführt.

Damit stellen wir den maximalen Nutzen für den Kunden her und sichern somit seine Investition. Ansonsten verhalten sich die Konstruktionsiterationen analog zu denen in der Produkterstellung.

Refactoring: Aufgrund der Kleinteiligkeit der meisten Aufgaben finden viele Refactorings in der laufenden iterativen Entwicklung statt. Von daher besteht meist nur noch wenig Bedarf nach größeren Zeiträumen für ein Refactoring. Je nach Änderungsumfang werden ein bis vier Refactoring-Blöcke von jeweils einer halben Iterationslänge ungefähr gleichverteilt eingeplant. Ansonsten verhält es sich mit diesen Refactoring-Blöcken analog zum Refactoring in der Produkterstellung.

APM legt einen besonderen Fokus auf lang laufende, wartungsintensive Projekte und/oder Projekte mit einem hohen Legacy-Anteil. Es geht in den expliziten Architekturphasen bzw. Architektur-Refactoring-Phasen nicht darum, eine Architektur *vorab* zu erstellen, sondern darum, für die nächsten Iterationen architekturelle Risiken zu entschärfen oder durch Refactorings die Basis für eine effektive Weiterentwicklung zu legen.

Referenzen und weiterführende Literatur

[1]Lyssa Adkins. *Coaching Agile Teams – A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition*. Addison-Wesley, 2010.

[2]Kent Beck et al. *Manifesto for Agile Software Development*. www.agilemanifesto.org, 2001.

[3]Mike Cohn. *User Stories Applied*. Addison-Wesley, 2004.

[4]Mike Cohn. *Agile Estimating and Planning*. Prentice Hall, 2006.

[5]Lisa Crispin und Janet Gregory. *Agile Testing – A Practical Guide for Testers and Agile Teams*. Addison-Wesley, 2009.

[6]Esther Derby und Diana Larsen. *Agile Retrospectives – Making Good Teams Great*. The Pragmatic Bookshelf, 2006.

[7]GTB Working Party Glossary. *ISTQB / GTB Standardglossar der Testbegriffe*. http://www.german-testing-board.info/downloads/pdf/CT_Glossar_DE_EN_V21.pdf, September 2010. Version 2.1.

[8]Mike Griffiths. *PMI-ACP Exam Prep*. RMC Pubns Inc, 2012.

[9]Ron Jeffries. *Essential XP: Card, Conversation, Confirmation*. http://xprogramming.com/articles/expcardconversationconfirmation/, August 2001.

[10]Craig Larman und Bas Vodde. *Scaling Lean & Agile Development – Thinking and Organizational Tools for Large-Scale-Scrum*. Addison-Wesley, 2009.

[11]Jean Lave und Étienne Wenger. *Situated Learning – Legitimate peripherical participation*. Cambridge University Press, 1991.

[12]Dean Leffingwell. *Agile Software Requirements – Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley, 2011.

[13]Robert C. Martin. *Clean Code – A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[14]Bernd Oestereich und Axel Scheithauer. *Analyse und Design mit der UML 2.5 – Objektorientierte Softwareentwicklung*. Oldenbourg, 2013. Unter Mitarbeit von Stefan Bremer.

[15]Bernd Oestereich und Christian Weiss. *APM – Agiles Projektmanagement – Erfolgreiches Timeboxing für IT-Projekte*.

dpunkt.verlag, 2008. Unter Mitarbeit von Oliver F. Lehmann und Uwe Vigenschow.

[16]Jeff Patton. *The new user story backlog is a map*. http://www.agileproductdesign.com/blog/the_new_backlog.html, Oktober 2008.

[17]Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Project Management Institute, 4. Auflage, 2008.

[18]Ken Schwaber und Jeff Sutherland. *The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game*. www.scrum.org/scrumguides, Juli 2013.

[19]Dave Thomas. *CodeKata – How to Become a Better Developer*. http://codekata.pragprog.com/, 2007.

[20]Uwe Vigenschow. *Testen von Software und Embedded Systems – Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. dpunkt.verlag, 2. Auflage, 2010.

[21]Uwe Vigenschow. *APM – Agiles Projektmanagement – Anspruchsvolle Softwareprojekte erfolgreich steuern*. dpunkt.verlag, 2015.

[22]Uwe Vigenschow, Björn Schneider und Ines Meyrose. *Soft Skills für Softwareentwickler – Fragetechniken, Konfliktmanagement, Kommunikationstypen und -modelle*. dpunkt.verlag, 3. Auflage, 2014.

[23]Uwe Vigenschow, Stefan Toth und Markus Wittwer. *Agil auf breiter Front – Techniken für eine erfolgreiche Projektsteuerung*. *dotnetpro*, (10):136 – 139, 2010.

[24]William C. Wake. *INVEST in Good Stories and SMART Tasks*. http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/, August 2003.

[25]Marcus Winteroll. *Geschichtenerzähler – Übersichtliche Anforderungen mit Story Maps*. *iX*, (5):104 – 107, 2011.

Uwe Vigenschow

Der APM-Guide zu: APM – Agiles Projektmanagement

Der *APM*-Guide stellt die zentralen Prinzipien, Methoden und Techniken dar, die den Kern des *APM*-Frameworks bilden. *APM* ist eine Methodik für die konsequente und praxisnahe Umsetzung agilen Projektmanagements im Kontext anspruchsvoller Softwareprojekte. *APM* beantwortet u.a. die Fragen, wie wir ein agiles Projekt vorbereiten und aufsetzen können, wie die Softwarearchitektur und agiles Projektmanagement zusammenspielen, wie eine Skalierung und Verteilung von agilen Projekten aussehen kann und wie wir innerhalb der Iterationen zu hochwertigen Releases kommen, unser Projekt agil steuern sowie Kanban und Lean Software Development in unsere Vorgehensweise integrieren können.

Der *APM*-Guide beschreibt in sieben Kapiteln:

- die Architektur von *APM*
- die grundlegenden Konzepte von *APM*
- wie Anforderungen strukturiert werden können
- die zentralen Artefakte
- die Meetings, über die die Informationsflüsse gesteuert werden
- das skalierbare *APM*-Rollenmodell
- das *APM*-Phasenmodell für die initiale Entwicklung sowie die spätere Wartung und Weiterentwicklung von Software

Der *APM*-Guide gibt Ihnen einen kostenlosen Überblick über das *APM*-Framework. Er dient als Einführung, Leitfaden und Referenz. Wenn Sie tiefer in *APM* einsteigen möchten, empfehlen wir »APM – Agiles Projektmanagement« (ISBN 978-3-86490-211-6), erschienen im dpunkt.verlag, als Buch oder E-Book.