1 Einleitung

Dieses Kapitel beginnt mit dem Agilen Manifest als Grundfeste der agilen Methoden und erklärt dann kurz die heute am weitesten verbreitete agile Vorgehensweise »Scrum«. Darauf aufbauend wird ein Überblick über verschiedene Methoden und Techniken der Requirements-Spezifikation, die heute in agilen Projekten Verwendung finden, gegeben. Die Grundprinzipien des Requirements Engineering für die agile Softwareentwicklung werden erläutert und es wird auf andere Aspekte des Requirements Engineering im agilen Umfeld eingegangen.

1.1 Das Agile Manifest

Im Februar 2001 traf sich eine Gruppe von 17 Personen in den Bergen von Utah, um über die Gemeinsamkeiten von verschiedenen Ansätzen der Softwareentwicklung zu diskutieren. Diese Gruppe nannte sich »The Agile Alliance« und das Ergebnis dieses Meetings waren der Begriff »agil« und das »Agile Manifest« [Agile Manifesto].

Das Agile Manifest besteht aus vier Leitsätzen und zwölf Prinzipien, die die Leitsätze erläutern. Es ist das Fundament für agiles Arbeiten und damit für alle agilen Methoden.

Manifest für Agile Softwareentwicklung:

Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen.

Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:

- **Individuen und Interaktionen mehr als Prozesse und Werkzeuge**
- **Funktionierende Software mehr als umfassende Dokumentation**
- **Zusammenarbeit mit Kunden mehr als Vertragsverhandlung**
- **Reagieren auf Veränderung mehr als das Befolgen eines Plans**

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.

→

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Die zwölf Prinzipien hinter dem Agilen Manifest

- Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen.
- Heiße Anforderungsänderungen selbst spät in der Entwicklung willkommen. Agile Prozesse nutzen Veränderungen zum Wettbewerbsvorteil des Kunden.
- Liefere funktionierende Software regelmäßig innerhalb weniger Wochen oder Monate und bevorzuge dabei die kürzere Zeitspanne.
- Fachexperten und Entwickler müssen während des Projekts täglich zusammenarbeiten.
- Errichte Projekte rund um motivierte Individuen. Gib ihnen das Umfeld und die Unterstützung, die sie benötigen, und vertraue darauf, dass sie die Aufgabe erledigen.
- Die effizienteste und effektivste Methode, Informationen an und innerhalb eines Entwicklungsteams zu übermitteln, ist im Gespräch von Angesicht zu Angesicht.
- Funktionierende Software ist das wichtigste Fortschrittsmaß.
- Agile Prozesse fördern nachhaltige Entwicklung. Die Auftraggeber, Entwickler und Benutzer sollten ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können.
- Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität.
- Einfachheit – die Kunst, die Menge nicht getaner Arbeit zu maximieren – ist essenziell.
- Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams.
- In regelmäßigen Abständen reflektiert das Team, wie es effektiver werden kann und passt sein Verhalten entsprechend an.

Das Agile Manifest ist ein Meilenstein in der Geschichte der Softwareentwicklung. Es gab bisher nur wenig Literatur, die so große Auswirkungen auf die Softwareentwicklung hatte wie dieses kurze Dokument. Heute kann man sagen, dass das Agile Manifest und seine Prinzipien praktisch alle Bereiche der Softwareentwicklung erreicht haben. Auf Basis des Agilen Manifests haben sich verschiedene Vorgehensweisen etabliert wie z. B. Scrum, Kanban, Extreme Programming oder Feature Driven Development.

Bei den agilen Vorgehensweisen stehen das Ergebnis und der Wert für den Kunden im Vordergrund. Anstelle einer einmaligen umfassenden Vorausplanung wird eine ständig rollierende Planung in Verbindung mit schnellem Feedback durch kurze Iterationen zur Risikominimierung verwendet. Dies zielt darauf ab, dass der Kunde möglichst rasch ein erstes Ergebnis ansehen und am besten auch gleich verwenden kann. Ein starker Fokus liegt auf Transparenz, Kommunikation und einem Team, das sich selbst steuert.

Neben diesen im Agilen Manifest und in Vorgehensmodellen wie Scrum explizit genannten Werten und Themen gibt es viele weitere Aspekte in der Softwareentwicklung, die in einem Projekt berücksichtigt werden müssen, auch wenn sie nicht durch die agilen Basiskonzepte explizit abgedeckt sind. Bereiche wie Risikomanagement, Testen und eben auch Requirements Engineering sind nach wie vor kritisch für den Erfolg von Softwareprojekten.

Für die erfolgreiche Einführung und den Einsatz agiler Vorgehensweisen ist es daher wichtig, dass einerseits eine Ausrichtung des Teams bzw. der Organisation an den agilen Werten und einer disziplinierten Anwendung der Regeln der von der Organisation gewählten agilen Vorgehensweise erfolgt, und andererseits auch alle anderen für den Erfolg wesentlichen Themen berücksichtigt werden, sodass diese in einer sinnvollen Art und Weise zu einem effektiven Framework für die Softwareentwicklung integriert werden.

1.2 Requirements Engineering im Kontext des Agilen Manifests

Im Agilen Manifest selbst steht fast nichts über Requirements Engineering. Trotzdem haben seine Leitsätze großen Einfluss darauf, wie ein agiles Team mit Anforderungen umgeht:

- **Individuen und Interaktionen mehr als Prozesse und Werkzeuge**
  Aus Sicht des Requirements Engineering kann man diese Aussage auf die Interaktion und Kommunikation zur Ermittlung und Abstimmung von Anforderungen sowie auf den Prozess und die Tools zu deren Verwaltung und Dokumentation beziehen.

  Dass die **Interaktion und Kommunikation** der Beteiligten und Betroffenen der wichtigste Faktor im Requirements Engineering ist, ist seit Langem bekannt. Wichtig ist aber auch, dass die Aussagen der Individuen und die Ergebnisse im Rahmen der Interaktion auf angemessene Art und Weise dokumentiert werden.

  Eine große Herausforderung in der Praxis ist die **Definition des Requirements-Engineering-Prozesses**. Dieser wird auch in agilen Teams oft zu formal und sequenziell beschrieben. Um die Agilität zu gewährleisten, ist es daher bei der Prozessdefinition zu empfehlen, nicht den Prozess insgesamt, sondern nur die einzelnen Teilprozesse und Techniken, wie z. B. Erhebungstechniken, Darstellungstechniken, einzelne Artefakte, Analysemethoden und Management-

techniken, klar zu beschreiben und zu einem modularen flexibel einsetzbaren Methodenbaukasten zusammenzustellen. In Kapitel 4 ist ein Teil eines solchen Baukastens zu finden.

Flexible **Requirements-Werkzeuge** helfen, das Requirements Engineering agil umzusetzen. Vor allem auch die Integration mit anderen Werkzeugen und das Vermeiden von Brüchen im Entwicklungsprozess sind essenziell für das effiziente Funktionieren des Requirements-Prozesses. Microsoft Word und Excel als die am meisten verwendeten Requirements-Werkzeuge sind nicht gerade dazu geeignet, den Requirements-Prozess nachhaltig und effizient zu gestalten. Aber auch »echte« Requirements-Management-Werkzeuge haben oft zu starre Prozesse implementiert oder ergehen sich in einer Attributierungsorgie. Die Integration mit anderen Werkzeugen im Entwicklungsprozess ist ebenfalls oft mangelhaft. Insofern ist die Werkzeugfrage auch im agilen Umfeld sehr wichtig.

- **Funktionierende Software mehr als umfassende Dokumentation**

Natürlich ist die gebaute Software wichtiger als die Dokumentation. Aber was ist eigentlich unter »Dokumentation« zu verstehen? Es kann jegliche Art von Anforderungsspezifikation, Architekturdokumenten, Designvorgaben, Prozessdefinitionen, Benutzerdokumentation, Entwicklerkommentaren im Sourcecode etc. gemeint sein.

Wenn man die Kosten der Erstellung einer funktionierenden Software inklusive Sourcecode im Vergleich zu den Dokumentationskosten inklusive Sourcecode-Kommentare in Abhängigkeit vom Dokumentationsgrad betrachtet, kommt man zu folgendem Ergebnis: Eine umfassende Dokumentation vor der Erstellung der ersten Codezeile macht aus Zeit- und Kostengründen keinen Sinn. Auf die Dokumentation komplett zu verzichten, ist aber ebenfalls keine gute Lösung. Die Kommunikationskosten und die Kosten für zusätzliche unnötige Iterationen werden sonst wegen vergessener Informationen, Spätfolgen durch fehlendes Verständnis für Umsetzungsentscheidungen und fehlende Nachvollziehbarkeit ebenfalls explodieren (siehe Abb. 1–1).

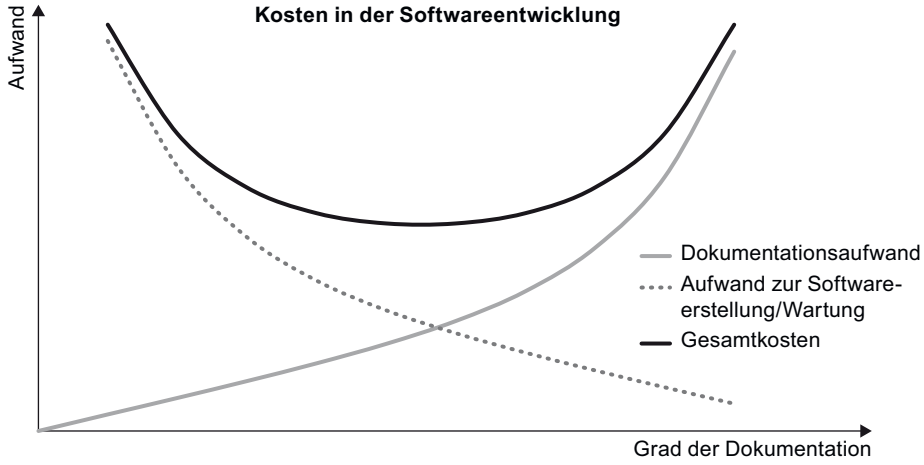

Abb. 1–1 *Kostenkurve in der Softwareentwicklung abhängig vom Dokumentationsgrad*

Jede Dokumentation unterstützt bis zu einem gewissen Grad die Flexibilität und die Effizienz und bremst sie ab einem gewissen (übertriebenen) Grad.

Es ist daher im Sinne der Flexibilität und Effizienz bei jeder Dokumentation zu überlegen, was der für das jeweilige Projekt optimale Grad ist. Nachfolgend sind beispielhaft **einige zentrale Dokumentationsarten** angeführt, für die dies gut überlegt werden sollte:

- Requirements
- Risikoanalysen
- Spikes[1] und Machbarkeitsanalysen
- Architekturspezifikation
- Codekommentare
- Testspezifikation und Testdokumentation
- Benutzerdokumentation
- Prozessdefinition
- Methodenbeschreibungen, Guidelines und Checklisten

Dokumentation, speziell Anforderungen, komplett wegzulassen, verursacht zusätzliche Kosten im Projekt! Den richtigen Grad an Dokumentation zu treffen, ist die Kunst guter Projektabwicklung.

1. Eine Definition des Begriffs Spike ist in Abschnitt 4.6.1 zu finden.

- **Zusammenarbeit mit Kunden mehr als Vertragsverhandlung**
  Hier geht es um *externe* Kunden-Lieferanten-Verhältnisse, die typischerweise auf einem Vertrag – welcher Art auch immer – basieren. Die Interpretation des Agilen Manifests ist in diesem Kontext sehr stark von der jeweiligen Projektsituation abhängig. In Projekten, in denen der Kunde selbst nicht genau weiß, was er will, muss die Zusammenarbeit mit dem Kunden sehr intensiv sein. In Projekten, in denen schon ziemlich klar ist, was der Kunde benötigt und dies auch schon vorab gemeinsam definiert wurde, ist eine weniger intensive Zusammenarbeit erforderlich.

  Die Intensität der Zusammenarbeit wird auch von der Grundhaltung zwischen Kunden und Lieferant stark beeinflusst. Bei Kunden, die partnerschaftlich agieren und zu denen man als Lieferant aufgrund langjähriger Beziehungen großes Vertrauen hat, werden auch kaum intensive Vertragsverhandlungen nötig sein. Umgekehrt ist es genauso. Partnerschaftliche Lieferanten werden im Sinne des Kunden denken und handeln. Ist dies jedoch nicht der Fall, so sind auch in agilen Projekten gute Verträge im Sinne der Risikominimierung und wechselseitigen Klarheit sehr wichtig. Wichtig ist in allen Fällen, dass der Vertrag und die Spezifikation aufeinander abgestimmt sind (siehe auch Kap. 7).
- **Reagieren auf Veränderung mehr als das Befolgen eines Plans**
  Änderungen sind die Normalität in Projekten. Wenn der Kundennutzen durch eine Änderung größer wird, ist eine Änderung auch zu befürworten. Es gibt jedoch Änderungen, die zwar vom Kunden gewünscht werden, die aber den Gesamtkundennutzen des Systems reduzieren. In solchen Fällen ist es besser, dem Kunden die bisher gewählte Umsetzung und die Konsequenzen der Änderung zu erklären und den ursprünglichen Plan weiterzuverfolgen.

Beispiel:

Ein Kundenvertreter wünscht sich nach Sichtung des ersten Prototyps, der von den Entwicklern selbstständig mit einfachen Bedienelementen umgesetzt wurde, nun in einer Webmaske eine Drag&Drop-Funktion. Dies wäre jedoch nur sehr aufwendig mit den zur Verfügung stehenden Technologien umzusetzen. Außerdem würde dadurch ein Teil seiner Systembenutzer ausgeschlossen, der eine ältere Browserversion verwendet oder auf mobilen Geräten arbeitet. Die Änderung nicht umzusetzen ist in diesem Fall die bessere Alternative für den Kunden. Sie stellt in der Bedienung nur unwesentlich mehr Aufwand dar, spart jedoch die hohen Zusatzkosten der Implementierungsänderung und berücksichtigt außerdem alle potenziellen Benutzergruppen. Für den einzelnen Kundenvertreter wäre die Änderung aus seiner Sicht durchaus mit einem Nutzen verbunden. Für den Kunden insgesamt und für den längerfristigen Nutzen aus dem System hätte sie jedoch nachteilige Auswirkungen.

In allen Projekten sollten Änderungen systematisch analysiert und bewertet werden, bevor sie akzeptiert und in das Backlog eingefügt werden. Der Kunde sollte auch immer eine Rückmeldung erhalten, ob und warum die Änderung von den anderen Beteiligten (Entwickler, IT-Betrieb, Marketing etc.) als Nutzen steigernd oder Nutzen verringernd eingeschätzt wird. Er kann ja auch im nicht sinnvollen Fall die Änderung trotzdem durchführen lassen – es ist ja schließlich sein Geld, das er dann mehr ausgibt! Diese Tatsache sollte ihm aber bewusst sein.

In der Praxis gibt es kaum Projekte, bei denen Zeit und Geld keine Rolle spielen und die Entwickler ohne Vorgaben einfach drauflosentwickeln oder Änderungen durchführen können. Es besteht daher in *jedem* Projekt die Notwendigkeit, zu Beginn zumindest eine grobe Planung und Spezifikation zu erstellen. Das Product Backlog und die Releaseplanung sind genau solche Pläne (siehe Abschnitte 6.2 und 6.3). So kann man *wissen* und nicht nur ahnen, ob bzw. wie weit man vom ursprünglich vorgesehenen Ziel und Rahmen abweicht.

Es gilt jedoch auch hier wieder das schon oben erwähnte Prinzip: Nur wenn die Kosten für die Erstellung und Pflege des Plans geringer sind als die Kosten, die sich durch eine unsystematische und schlecht abgestimmte Softwareentwicklung ohne Plan ergeben, sollte ein Plan erstellt und gewartet werden.

Dass Pläne sich ändern, ist ein Faktum. Gute Planer wissen dies und sehen in Änderungen auch nichts Negatives und schon gar kein Versagen ihrer planerischen Fähigkeiten.

Laufende Änderungen sind ein Naturgesetz in der Softwarewelt, das der gute Planer akzeptiert und berücksichtigt.

Ein Grundprinzip der Planung lautet daher: Entweder gut oder gar nicht! Wobei hier mit guter Planung nicht gemeint ist, dass das System vorab bis in die letzte Zeile Quellcode vorauszuplanen ist, sondern es geht hier darum, die Wegpunkte auf der Reise zum Ziel festzulegen, an denen man sich immer wieder orientieren kann. Wenn aber ein Plan erstellt wird, dann muss er auch bis zum Ende laufend gepflegt und konsistent gehalten werden. Wenn hier Nachlässigkeit aufkommt und der Plan von den Beteiligten als nicht konsistent angesehen wird, dann ist er nutzlos.

1.3 Scrum im Überblick

Scrum wurde von Ken Schwaber und Jeff Sutherland entwickelt und ist ein Framework für agiles Projektmanagement. Durch seine einfache Struktur ist es schnell verständlich und erleichtert dadurch die Umsetzung der täglich gelebten Prozesse (siehe Abb. 1–2).

Der Begriff »Scrum« wurde aus dem Rugby entlehnt und bedeutet »Gedränge«. Dies steht für den Teamgeist und die selbstorganisierende Weise, mit der die Teams ein gemeinsames Ziel erreichen. Offenheit, Respekt und Transparenz sind in Scrum die Basis, um gute Arbeitsergebnisse zu erzielen.

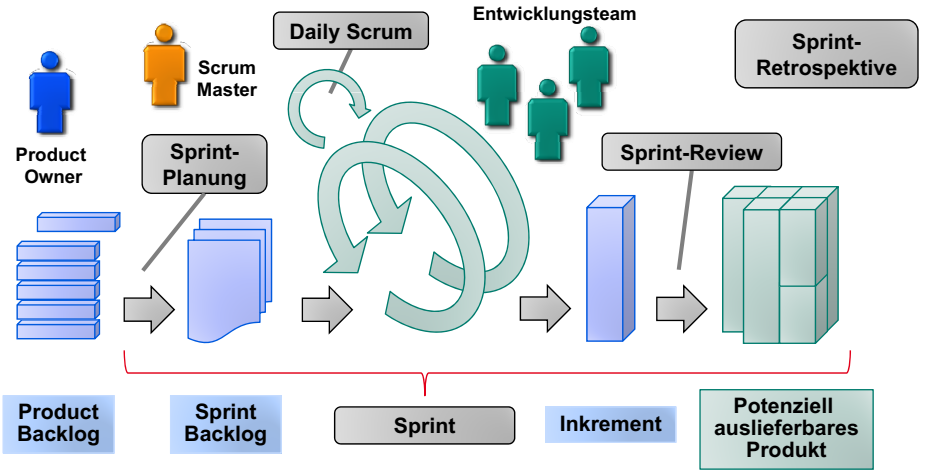

Abb. 1–2 *Überblick über Scrum*

Das Scrum-Regelwerk basiert auf der Theorie der empirischen Prozesssteuerung und verändert den in klassischen Vorgehensweisen angegebenen Planungsansatz grundlegend. Scrum nutzt kurze zeitlich festgeschriebene Iterationen (»**Sprint**«), die immer gleich lang sind (»**timeboxed**«). Typischerweise werden Sprint-Längen zwischen zwei und vier Wochen gewählt. Das Scrum-Team soll in dieser Zeit möglichst ungestört arbeiten können. Damit wird laufend messbarer Nutzen für den Kunden generiert und das Projektrisiko minimiert.

Der wesentliche Fokus in Scrum ist das **selbststeuernde Team**, bei dem es keinen Projektleiter gibt.

Das **Scrum-Team** besteht aus drei Rollen:

- Der **Product Owner** ist der Produktverantwortliche und zuständig für das Product Backlog. Er definiert, priorisiert und ändert Anforderungen (siehe Abschnitt 2.1).
- Der **Scrum Master** kümmert sich um die Einhaltung der vorgegebenen Regeln und hält Störfaktoren vom Team fern. Er mischt sich inhaltlich nicht ein, sondern sorgt dafür, dass das ganze Scrum-Team im Sinne der Gesamtorganisation möglichst produktiv ist[2] (siehe Abschnitt 2.3).

2. Im Verlauf des Buches wird anstelle des Begriffs »Scrum Master« der allgemeinere Begriff »Agile Master« verwendet (siehe Abschnitt 2.3).

- Das **Entwicklungsteam** setzt die Backlog-Einträge zu einem neuen Produktinkrement um. Es besteht aus mehreren Personen, die ihre Aufgaben im Team professionell wahrnehmen können. Das Entwicklungsteam hat alle Kompetenzen, die notwendig sind, um ein Produktinkrement fertigzustellen. Die Teamgröße sollte zwischen drei und neun Personen liegen. Innerhalb des Teams gibt es keine weitere Rollenunterteilung. Alle Teammitglieder werden als »Entwickler« bezeichnet (siehe Abschnitt 2.2).

Es gibt in Scrum folgende explizit definierte **Artefakte**:

- Im **Product Backlog** werden alle anstehenden Anforderungen an das umzusetzende System erfasst. Es wird nicht erwartet, dass alle Anforderungen schon von Beginn an bekannt sind und sich im Projektverlauf nicht mehr ändern. Das Product Backlog enthält die im Moment bekannten Anforderungen in der notwendigen Feinheit, die erforderlich ist, um die nächsten Sprints zu planen. Je weiter die Umsetzung einer Anforderung in der Zukunft liegt, desto gröber sind die Informationen. Die Anforderungen im Product Backlog sollen eindeutig gereiht (priorisiert) sein.
- Im **Sprint Backlog** werden die für einen konkreten Sprint ausgewählten Product-Backlog-Einträge festgehalten. Die Einträge werden nicht vom Product Owner, sondern vom Entwicklungsteam aus dem Product Backlog ausgewählt (»Pull«-Prinzip). Das Sprint Backlog macht die in einem Sprint umzusetzende Arbeit sichtbar und zeigt laufend den Status der Arbeiten im Sprint.
- Das **Produktinkrement** ist das Ergebnis der Arbeiten eines Sprints inklusive der Ergebnisse aus allen vorangegangenen Sprints. Das Inkrement muss am Ende des Sprints die »**Definition of Done**« erfüllen (siehe Abschnitt 3.2) und potenziell auslieferbar sein.

Der Ablauf eines Sprints ist durch folgende Ereignisse gekennzeichnet:

- **Sprint-Planung**
  Am Beginn jedes Sprints steht das Sprint-Planungs-Meeting, in dem im Wesentlichen zwei große Themen behandelt werden: Was wird im bevorstehenden Sprint umgesetzt? Und: Wie wird es umgesetzt? Die Antwort auf die erste Frage besteht aus der Auswahl von Einträgen aus dem Product Backlog, die zunächst ins Sprint Backlog übernommen werden. Diese Auswahl wird vom Entwicklungsteam eigenständig getroffen. Zum Teil bereits mit Designentscheidungen angereichert wird die zweite Frage der Planung behandelt, nämlich wie das Entwicklungsteam vorhat, die Anforderungen umzusetzen.
- **Daily Scrum**
  Dies ist ein tägliches Meeting für die Synchronisation im Entwicklungsteam. Von jedem Mitglied werden in insgesamt maximal 15 Minuten drei Fragen beantwortet: Was habe ich seit gestern fertiggestellt? Was werde ich heute machen, um dem Sprint-Ziel einen Schritt näherzukommen? Was behindert mich in meiner Arbeit?

- **Sprint-Review**
  Am Ende eines Sprints wird ein informelles Review-Meeting von maximal vier Stunden zur Analyse und Bewertung der erreichten Ergebnisse abgehalten. Wenn nötig wird auch das Product Backlog angepasst. Fokus sind die Ergebnisse des vergangenen Sprints. Wenn möglich erfolgt auch eine Vorführung des Produkts. Teilnehmer sind das Scrum-Team und die wichtigsten Stakeholder.
- **Sprint-Retrospektive**
  Dieses Meeting steht im Zeichen der Prozessverbesserung und findet zwischen Sprint-Review und nächster Sprint-Planung statt. Hier wird zurückgeschaut auf den Ablauf des vergangenen Sprints, um daraus zu lernen. Es werden Verbesserungspotenziale identifiziert und die Umsetzung der Verbesserungen eingeplant.

Die laufende Arbeit wird in Scrum gut visualisiert. In der Praxis hat sich für das operative Projektcontrolling in Scrum ein **Taskboard** bewährt (siehe Abschnitt 6.5). Ebenso hat sich eine grafische Veranschaulichung der Menge an insgesamt noch offenen Arbeiten im Verlauf des Sprints und im Projekt in einem **Burndown-Chart** als geeignet erwiesen.

Die Arbeitsplanung erfolgt durch jedes Teammitglied für sich selbst nach dem »Pull«-Prinzip. Das heißt, Teammitglieder bekommen keine Aufgaben von irgendjemandem zugeteilt, sondern jedes Teammitglied holt sich eigenständig seine Arbeit.

Zusammengefasst definieren Schwaber und Sutherland in ihrem Scrum Guide [Schwaber & Sutherland 2013] Scrum als »ein Rahmenwerk, innerhalb dessen Menschen komplexe adaptive Aufgabenstellungen angehen können, und durch das sie in die Lage versetzt werden, produktiv und kreativ Produkte mit dem höchstmöglichen Wert auszuliefern«.

Wichtig für die erfolgreiche Anwendung von Scrum ist auch, dass sich alle Beteiligten bewusst sind, dass Scrum kein fertiges Prozessmodell ist, sondern primär (Projekt-)Managementthemen fokussiert. Scrum macht keine Vorgaben für Themen wie Requirements Engineering, Risikomanagement, Architektur, Programmierung oder Testen. Diese sind bei komplexen Produktentwicklungen unbedingt notwendig und sollten explizit definiert und mit den Vorgaben von Scrum kombiniert werden. Gemeinsam mit diesen Ergänzungen bringt die Anwendung von Scrum nachhaltigen Nutzen für die Organisation.

1.4 Ein Blick auf das große Ganze

Der Begriff »Requirements-Spezifikation« wird oft mit Eigenschaften wie »komplex«, »überstrukturiert« assoziiert und mit dicken, von kaum jemand zu überblickenden Vertragswerken gleichgesetzt. In diesem Buch wird der Begriff »Requirements-Spezifikation« nicht einem bestimmten Umfang zugeordnet, da auch ein einzelnes Requirement eine Requirements-Spezifikation darstellen kann. Er wird auch keiner bestimmten Methode oder Zeitalter zugeschrieben, sondern ganz allgemein wie folgt verwendet:

> Ein »Requirement« ist *jede* Anforderung eines Stakeholders und jede Eigenschaft, die ein geplantes System besitzen soll. Eine »Requirements-Spezifikation« ist *jede* Repräsentation eines oder mehrerer Requirements, unabhängig davon, in welcher Form oder Granularität dies spezifiziert wird.

Dies beginnt schon bei den inhaltlichen Zielen, die die Requirements auf höchster Ebene darstellen. Agile Artefakte wie User Stories oder Epics sind ebenfalls Requirements. Natürlich ist auch ein Use Case oder ein Szenario, egal ob es textuell oder mittels grafischer Darstellung spezifiziert wurde, ein Requirement, genauso wie wenn der Kunde die Länge eines Feldes oder die Farbe eines Textes nach seinen Wünschen definiert. All dies sind Requirements, jedoch in unterschiedlicher Detaillierungstiefe und Darstellungsart.

Zur Strukturierung und Einordnung von Requirements kann man grundsätzlich drei Ebenen unterscheiden:

- Die **High-Level-Sichtweise**, um den Überblick über das Vorhaben zu erlangen bzw. zu behalten
- Die **Strukturierungsebene,** auf der unterschiedliche Artefakte in einen Zusammenhang gebracht werden
- Die **Detailebene** mit den feingranularen Inhalten

In Abbildung 1–3 werden diese drei Ebenen dargestellt und mit den drei Sichten »Kunde«, »Entwickler« und »Management« kombiniert. In dieser Matrix werden die wichtigsten in der agilen Softwareentwicklung verwendeten Requirements-Artefakte im Überblick angegeben. Die Kapitel 4 und 6 widmen sich diesem Thema noch detaillierter.

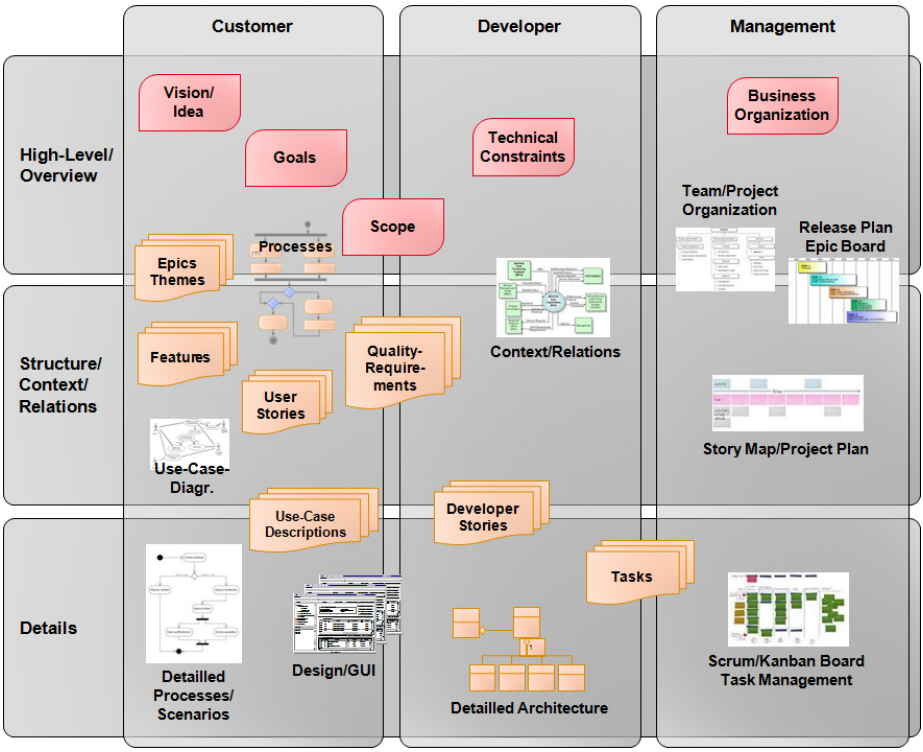

Abb. 1–3 *Überblick über Requirements-Artefakte im agilen Umfeld*

Da ein großer Teil der Projekte in einem komplexen Organisations- und Kundenumfeld stattfindet, wird angelehnt an [Leffingwell 2011] noch eine erweiterte Sichtweise auf die agile Organisation, die Rollen und Verantwortungsbereiche dargestellt (siehe Abb. 1–4).

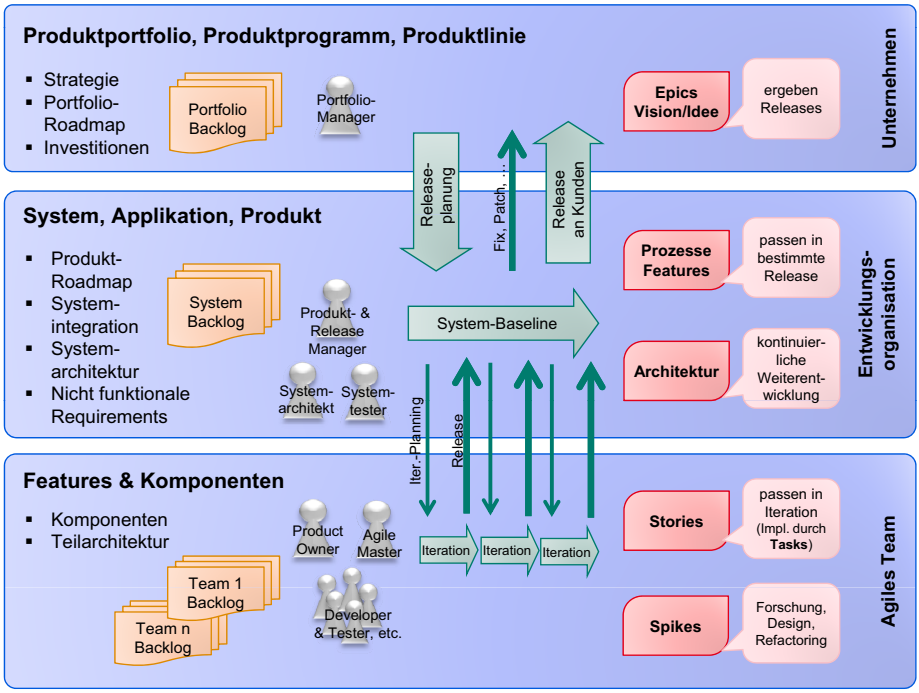

Abb. 1–4 *Agile Requirements- und Organisationsstrukturen in einem komplexen Umfeld*

Dabei ist insbesondere zu berücksichtigen, dass bei größeren Organisationen und Projekten, die mehr als nur ein agiles Team beschäftigen, übergeordnete Kommunikationskanäle und Koordinationsrollen notwendig sind. Es muss sichergestellt sein, dass die Teams übergreifend auf demselben Niveau, z. B. bezüglich Codequalität, Usability, sowie in dieselbe Richtung, was die Anforderungen und Ziele des Kunden betrifft, arbeiten.

Darüber hinaus ist es wichtig, die Gesamtsicht nicht zu vergessen. Dies ist vor allem bezüglich der Requirements und beim Testen von Bedeutung. Tests, die unkoordiniert in einzelnen Teams stattfinden, werden keine ausreichende Qualitätssicherung darstellen. Außerdem sind meist teamübergreifende Architekturüberlegungen, Integrationstests und übergeordnete Systemtests und Business-Prozess-Sichtweisen nötig, um aus den vielen Einzelergebnissen der Teams auch eine angemessene Qualität des Gesamtsystems im Sinne des Kunden zu erreichen.

Drei bis vier übergeordnete Rollen sind in größeren Organisationen dazu sinnvoll: ein übergreifender Produkt- und Releasemanager bzw. Product Owner, ein Gesamtsystemarchitekt sowie ein übergeordneter Testmanager und evtl. bei sehr großen bzw. mehreren parallel zu koordinierenden Entwicklungsvorhaben noch ein Portfoliomanager.

Die Zusammenhänge im Requirements Engineering sind fast immer komplexer, als es auf den ersten Blick aussieht (siehe Abb. 1–5). Um teamübergreifende

Anforderungen, Einsatzszenarien bei den Anwendern und das große gemeinsame Ziel optimal im Blick behalten zu können, müssen die oft verwendeten User Stories sinnvoll um weitere Requirements-Artefakte ergänzt werden. Jedes Artefakt steht dabei mit vielen verschiedenen anderen Artefakten in Beziehung.

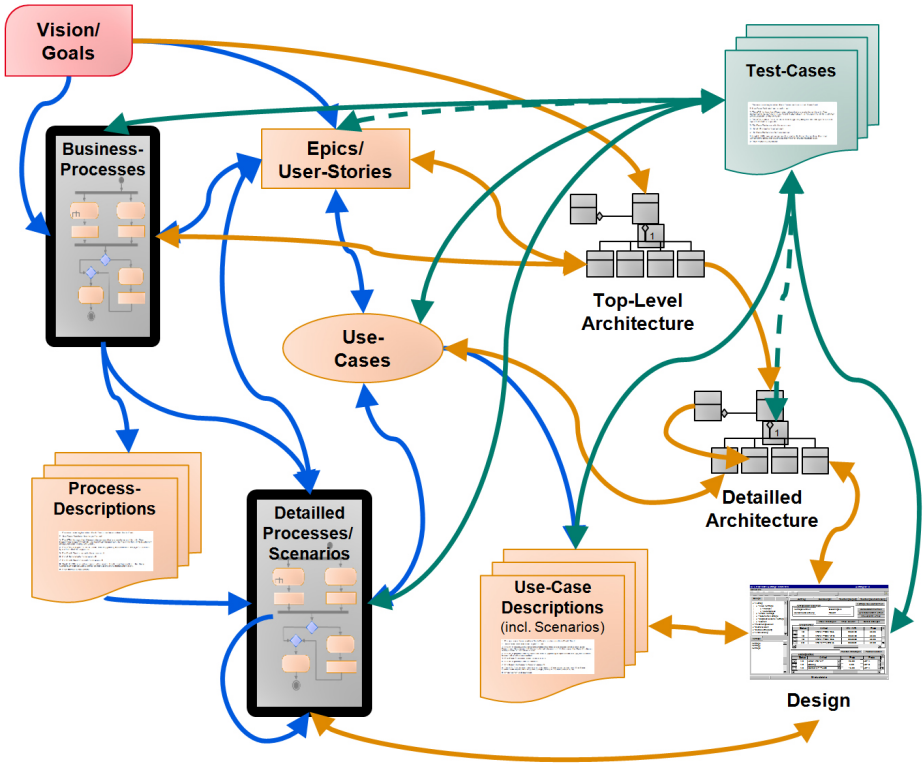

Abb. 1–5 *Komplexe Zusammenhänge im Requirements Engineering*[3]

Diese Artefakte und Beziehungen sollten in jedem Projekt berücksichtigt und gemanagt werden. Passiert dies nicht, sind sie zwar nicht mehr explizit sichtbar, aber leider immer noch vorhanden. Das Nichtbeachten dieser Situation führt dann oft dazu, dass Ineffizienzen wie z. B. Leerlaufzeiten oder auch Fehler, z. B. vergessene Anpassungen und Testfälle oder Inkonsistenzen zwischen Artefakten, entstehen. Die Folge ist, dass dann z. B. der Tester nicht oder erst verzögert informiert wird, wenn sich an den Anforderungen etwas ändert, und daher die Testfallerstellung erst verspätet beginnen kann oder die falschen Testfälle erstellt werden.

3. Aus Gründen der Übersichtlichkeit sind in dieser Grafik nur die wichtigsten Requirements-Artefakte und Beziehungen dargestellt. Code-Artefakte, externe Rechtsgrundlagen und andere gegebenenfalls noch relevante Artefakte wurden weggelassen, müssen in der Praxis jedoch ebenfalls berücksichtigt werden.

Unabhängig davon, welche Vorgehensweise in einem Projekt angewendet wird: Die komplexen Beziehungen zwischen den Projektartefakten bestehen in jedem Projekt und müssen entsprechend berücksichtigt werden.

In vielen Projekten versucht man, diese Zusammenhänge und Komplexitäten nicht durch entsprechende Dokumentation in passenden Artefakten und Verwaltung in geeigneten Tools zu beherrschen, sondern durch intensive persönliche Kommunikation. Die Kommunikationstheorie hat jedoch festgestellt – und die Praxis hat dies bestätigt –, dass sich der Kommunikationsaufwand mit der Anzahl der Kommunikationsknoten und Beziehungen exponentiell erhöht. Schon bei einer einfachen und nicht alle Beziehungen eines Projekts umfassenden Abbildung wird ersichtlich, dass es nicht einmal mit sehr großem Kommunikationsaufwand möglich ist, alle Zusammenhänge zwischen den Kundenanforderungen und den restlichen Projektartefakten zu bewahren und konsistent zu halten. In der Praxis wird dies in keinem Projekt ausschließlich auf Basis der persönlichen Kommunikation funktionieren.

Was in Projekten ohne explizite Dokumentation der Artefaktbeziehungen oft passiert, ist, dass ein Großteil der eigentlich für eine nachhaltige Entwicklung notwendigen Informationen nur unzureichend oder inkonsistent vorhanden ist. Speziell für Nichtteammitglieder sind diese Informationen nicht nutzbar, da sie zwar in den Köpfen von Teammitgliedern vorhanden sind, jedoch nur mit großem Aufwand an andere weitergegeben werden können. Außerdem können sie nicht strukturiert analysiert und geprüft werden. Das Wissen ist in den Köpfen »eingebunkert«. In vielen Projekten führt dies dann zu Problemen.

Die wesentlichen Beziehungen und Inhalte der durchgeführten Kommunikation sollten daher auch dokumentiert werden, um dem Wissensverlust entgegenzuwirken (siehe Grundprinzip 3 in Abschnitt 1.5). Des Weiteren sollten Aufwände, die zur Aufrechterhaltung der Beziehungen zwischen den Artefakten nötig sind, durch den Einsatz von passenden Werkzeugen (Requirements Management, Modellierung, Codeverwaltung, Testmanagement, Prozessautomatisierung etc.) reduziert oder vermieden werden.

1.5 Die fünf Grundprinzipien des Requirements Engineering in der agilen Softwareentwicklung

Für die Handhabung von Requirements in der täglichen Projektpraxis werden nachfolgend fünf Grundprinzipien beschrieben (siehe Abb. 1–6). Diese gehen auf die spezifischen Situationen im agilen Umfeld ein und berücksichtigen zusätzlich anerkanntes Requirements-Engineering-Wissen:

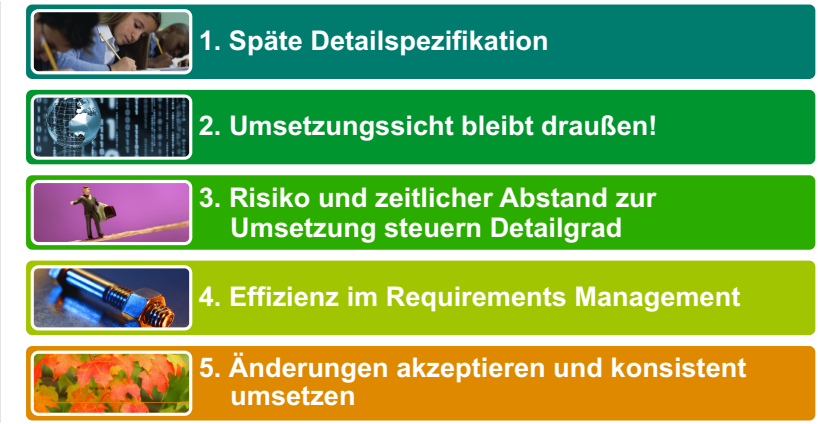

Abb. 1–6 *Die fünf Grundprinzipien des Requirements Engineering in agilen Projekten*

Grundprinzip 1: Späte Detailspezifikation

Die schriftliche Spezifikation zum spätest sinnvollen Zeitpunkt erstellen

Der spätestmögliche sinnvolle Zeitpunkt ist jener Zeitpunkt, zu dem der Aufwand für das Erstellen und *nachträgliche Ändern* der Spezifikation möglichst klein ist gegenüber dem Aufwand, der mit anderen Kommunikationsformen aufgewendet würde, wenn die Spezifikation nicht schriftlich erstellt wird. Ziel ist immer, mit einer schriftlichen Spezifikation genauso viel Informationsgehalt für die effiziente Erstellung des Systems zwischen den Beteiligten zu transportieren, dass dabei das Risiko und die Kosten über den gesamten Lebenszyklus minimiert werden.

Ein Erstellen der **Spezifikation *vor* diesem Zeitpunkt** führt zu einem unnötigen Mehraufwand in der Dokumentation, z. B. weil sich in der Zwischenzeit schon die Voraussetzungen geändert haben oder weil aufgrund unklarer Vorstellungen noch ein laufender Diskussions- und Änderungsprozess stattfindet.

Ein Erstellen der **Spezifikation *nach* diesem Zeitpunkt** ist aus Informationssicht unnötig, weil in vielen Bereichen die Kommunikation dann schon auf andere Art und Weise stattgefunden hat, die entsprechenden Systemteile schon programmiert sind und damit kein Informationsgewinn durch die Spezifikation erreicht wird. Der Spezifikationsaufwand wäre damit verlorener Aufwand. Trotzdem könnte eine nachträgliche Spezifikation bzw. Dokumentation eventuell aus rechtlicher Sicht erforderlich sein, um z. B. einen vorgeschriebenen Standard oder ein Gesetz einzuhalten oder das Haftungsrisiko zu minimieren.

Um die benötigten Spezifikationsteile zum spätest sinnvollen Zeitpunkt zu erstellen, kann die Testspezifikation als Detailspezifikation genutzt und so Detailanforderungen dann z. B. erst kurz vor Beginn des Umsetzungs-Sprints als Testfälle formuliert werden. Für vertraglich besonders kritische Themen wie z. B.

K.-o.-Kriterien (siehe Abschnitt 7.2) müssen alle relevanten Details, die notwendig sind, um die Machbarkeit der Anforderungen festzustellen, noch vor Vertragsabschluss geklärt sein. Bei größeren Projekten oder wenn mehrere Teams zusammenarbeiten, muss außerdem noch berücksichtigt werden, dass Detailspezifikationen (z. B. bei Schnittstellen) aus einer koordinativen Sicht auch schon zu einem früheren Zeitpunkt sinnvoll sein können, um Reibungsverluste durch abweichendes Vorgehen der Teams zu vermeiden.

Möglichst viele Details in die Testspezifikation verlagern

Oft werden Details, die am Anfang festgelegt wurden, sehr schnell wieder geändert oder gar überflüssig. Der Aufwand in der Anfangsphase des Projekts kann daher reduziert werden, wenn hier keine Details, sondern nur die grundlegenden Ziele und Anforderungen spezifiziert werden.

Da sich Anforderungsspezifikation und Testspezifikation zu einer Gesamtspezifikation ergänzen, ist es sinnvoll, die Details nicht in der Anforderungsspezifikation, sondern in einer Testspezifikation festzuhalten. Tests sind hier nichts anderes als eine Requirements-Spezifikation aus einer etwas anderen Sicht. Modellbasierte oder logische Testfallspezifikationen sind sehr ähnlich zu einer herkömmlichen Requirements-Spezifikation. Beispielsweise ist eine testdatenbasierte konkrete Testfallspezifikation vergleichbar mit Specification by Example (siehe Abschnitt 4.8.1).

Auch in der Testspezifikation können die grundlegenden Requirements-Spezifikationstechniken angewendet werden. Beispielsweise können im Rahmen des Model Based Testing Diagramme und Prozesse spezifiziert und dann durch entsprechende Testsichten und Testdaten ergänzt werden.

Der Vorteil der Testspezifikation gegenüber der herkömmlichen Requirements-Spezifikation ist, dass Testfälle mit einer größeren Wahrscheinlichkeit konsistent gehalten werden, da hier die Abweichungen zum Code bei der Testdurchführung sofort auffallen und dann angepasst werden müssen. Voraussetzung dafür ist, dass die Tests bei jeder Änderung an den dazugehörenden Codeteilen ausgeführt werden. Darüber hinaus kann man durch geschickte Gestaltung der Requirements- und Testspezifikation daraus auch die Benutzerdokumentation (halb-)automatisch erstellen und damit den Aufwand zusätzlich verringern.

Grundprinzip 2: Umsetzungsdetails bleiben draußen!

Nur das spezifizieren, was einen zusätzlichen Informationsgehalt für den Kunden bringt. Das WIE möglichst den Entwicklern überlassen.

Wichtig für eine Anforderungsspezifikation sind jene Punkte, die dem Auftraggeber wichtig sind, sowie die Hintergründe, die zu deren Verständnis erforderlich sind. Schriftlich sollen nur diejenigen Inhalte festgehalten werden, die auch einen Mehrwert und zusätzlichen Informationsgehalt für den Kommunikationsprozess zwischen Kunde und Entwickler bringen.

Das WIE – also Entwicklungs- und Umsetzungsvorgaben, z.B. wie die Datenbank intern aufgebaut sein soll – sollte in der Anforderungsspezifikation des Auftraggebers vermieden werden. Es schränkt die Entwickler nur unnötig ein und verhindert gute alternative Lösungsansätze. Außerdem kann und will der Kunde technische Details meist nicht beurteilen.

Grundprinzip 3: Risiko und zeitlicher Abstand zur Umsetzung steuern Detailgrad

Den Detaillierungsgrad passend zum Haftungsrisiko und potenziellen Wissensverlust wählen

Menschen vergessen in kurzer Zeit sehr viel von dem, was besprochen wird. Nach fünf Tagen sind nur mehr ca. 25–50 % des Wissens vorhanden [Wikipedia]. Daher ist es notwendig, die wichtigen Dinge schriftlich festzuhalten, um nachträglich unnötige Probleme oder Diskussionen, wie das denn nun wirklich vereinbart war, zu vermeiden. Sinnvoll ist es, nicht nur aufzuschreiben, WAS der Auftraggeber haben möchte, sondern auch das WARUM festzuhalten.

Auch bei Themen, die haftungsrelevant sein können, z.B. Funktionen, die eine Gefährdung von Personen bewirken oder einen großen finanziellen Schaden verursachen können, ist es ratsam, die Wünsche und dahinterliegenden Gründe des Auftraggebers schriftlich festzuhalten. Dies gilt insbesondere dann, wenn aus Kostengründen sicherheitsrelevante Funktionen oder Schritte geändert, reduziert oder gar weggelassen werden.

Grundprinzip 4: Effizienz im Requirements Management

Die Beziehungen zwischen Artefakten effizient verwalten!

Abbildung 1–5 auf Seite 14 zeigt, dass in jedem Projekt viele Abhängigkeiten zwischen den Artefakten vorhanden sind. Die Abhängigkeiten bestehen zumindest zwischen Kundenanforderungen, Testfällen und Code. Zusätzlich kommen je nach Dokumentationsgrad noch Prozess- und Architekturmodelle oder andere Artefakte dazu. Neben diesen zwingenden Abhängigkeiten ergeben sich oft auch Abhängigkeiten innerhalb einer Artefakt-Art, also zwischen verschiedenen Requirements oder einzelnen Architektur- oder Codeelementen.

Wenn man sich dafür entscheidet, Artefakte und deren Beziehungen zu erstellen und zu verwalten, dann ist es zwingend nötig, bei allen Änderungen auch alle Abhängigkeiten zu aktualisieren.

Sobald auch nur eine Änderung in den Beziehungen zwischen Projektartefakten nicht aktualisiert wurde, ist das gesamte Beziehungsgeflecht wertlos, da die Beteiligten nicht mehr darauf vertrauen können, dass die gespeicherten Informationen noch gültig sind.

Exkurs: Passende Detailliertheit der Beziehungsdokumentation

Implizit wird durch die oben stehende Aussage der passende Grad der Detailliertheit gesteuert, bis zu dem es sinnvoll ist, die Beziehungen zu verwalten:

Auf sehr **oberflächlicher Ebene** macht es keinen Sinn, weil hier der Informationsgewinn, der in einer konkreten Beziehung steckt, nicht gegeben ist. Zum Beispiel haben die Beteiligten recht wenig davon, wenn sie wissen, dass eine sehr grobe Anforderung (siehe auch Abschnitt 4.1.2) wie »Die Zeiterfassung wird effiziente Erfassungs- und Auswertungsmöglichkeiten für alle Zeitarten bereitstellen« mit der Testspezifikation »Test aller Zeiterfassungsarten« zusammenhängt.

Viel eher würde es einen Nutzen darstellen, wenn die Betroffenen wissen, dass die Story »Als Mitarbeiter möchte ich meine Tagesarbeitszeit und die Pausen erfassen, damit ich …« mit den Testfällen »Kommt-/Geht-Zeit«, »Pauseneingabe«, »Tagesarbeitsdauerberechnung« und »Interne Arbeitszeitregelungen« zusammenhängt. Damit wissen dann der Entwickler und Tester, was bei einer Änderung an dieser Story neu zu testen ist bzw. welche Testfälle evtl. angepasst werden müssen. Wenn nun noch die dazugehörigen Codestellen verknüpft werden, dann hat auch der Entwickler zusätzliche Infos, die er verwerten kann.

Insgesamt wird der Nutzen in einer Zeitersparnis bestehen, die bei allen Betroffenen erzielt werden kann, indem Fragen, Unklarheiten und zusätzlicher Rechercheaufwand reduziert werden.

Auf zu **detaillierter Ebene** macht das Dokumentieren der Beziehungen jedoch auch keinen Sinn. Der Aufwand für die Erstellung und Pflege der Beziehungsdokumentation steigt hier exponentiell an und kostet dann schnell mehr als der Nutzen, den die detaillierte Dokumentation bringt.

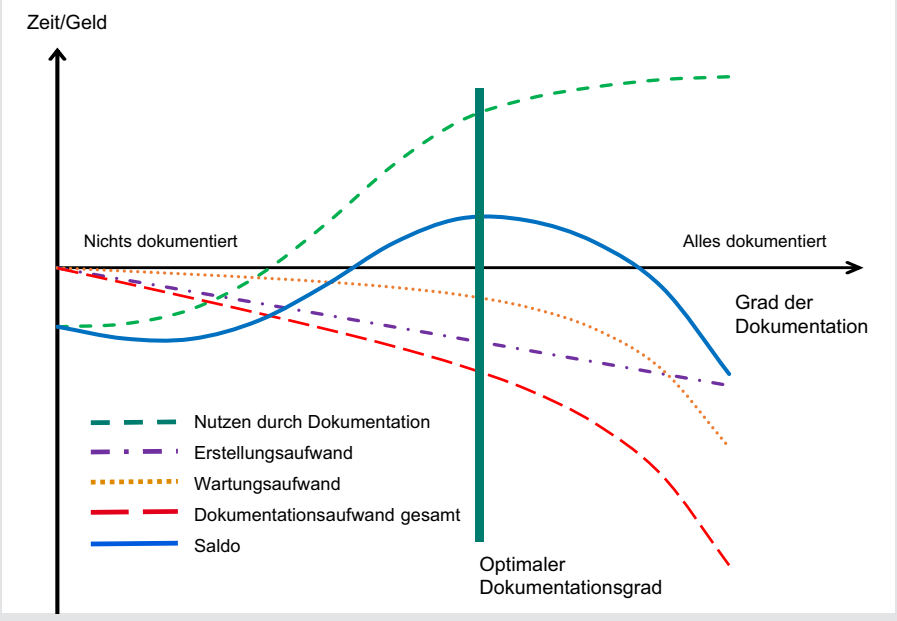

Abb. 1–7 *Nutzen der Dokumentation abhängig vom Detaillierungsgrad*

Die Kosten für die explizite Beziehungsdokumentation steigen exponentiell mit dem Grad der Detailliertheit. Dabei steigt der Erstellungsaufwand ungefähr linear mit dem Umfang der zu dokumentierenden Elemente, der Wartungsaufwand steigt jedoch exponentiell mit zunehmendem Dokumentationsumfang und verursacht so den starken Anstieg der Kosten in der Dokumentation.

Wenn **keine Dokumentation** vorhanden ist, bedeutet das nicht, dass dies nichts kostet. Es werden bei Nichtdokumentation erhöhte Kosten verursacht durch Fragen, Unklarheiten und zusätzlichen Rechercheaufwand, der im Team nötig wird. Ebenso muss berücksichtigt werden, dass das Vergessen und evtl. Wiederherstellen nicht dokumentierter Informationen auch Aufwand verursacht. Diese Aufwände sind in der Grafik als negativer Nutzen dargestellt. Bei Fehlen der Beziehungen zwischen Anforderungen und Testfällen wird zusätzliche Zeit für nicht notwendige Tests verschwendet.

Der **Nutzen** verhält sich leider nicht exponentiell, sondern ist ein Art Hysterese-Funktion (siehe Abb. 1–7). Das heißt, er steigt am Anfang langsam an (sehr geringer Detailgrad bringt wie oben erwähnt kaum einen Nutzen), hat dann einen Bereich, in dem er stark zunimmt, und flacht dann ab einer gewissen Detailliertheit wieder ab, weil durch die stärkere Detailliertheit kaum ein zusätzlicher Zeit- und Informationsgewinn für die Betroffenen erreicht wird und zusätzlich der Aufwand für das Überblicken und Nachvollziehen der vielen Detailzusammenhänge für den Betroffenen auch zu groß wird.

Es gibt daher ein Gesamtoptimum für den Dokumentationsgrad.

Der (zugegeben unscharf formulierte) beste Zeitpunkt, um die weitere Detaillierung der Dokumentation zu beenden, ist aus Sicht des Autors dann gegeben, wenn der Wartungsaufwand für die Beziehungen und abhängigen Artefakte einen Wert von ca. 20% des Erstellungsaufwands eines Artefakts überschreitet.

Wenn man z.B. eine Anforderung und deren Beziehungen zu anderen Artefakten mit einem Aufwand von einer Stunde erstellt und danach bei einer Änderung für die Anpassung des Requirements inkl. seiner Beziehungen maximal ca. 10–15 Minuten benötigt, wird dies noch angemessen sein. Wenn der Aufwand für die Anpassung des Requirements und seiner Abhängigkeiten in diesem Beispiel deutlich mehr als 15 Minuten in Anspruch nimmt, sollte man darüber nachdenken, ob es nicht besser ist, die Dokumentationstiefe des Requirements und der Beziehungen wieder zu reduzieren.

Jede *nicht* explizit verwaltete Beziehung bedeutet eine Aufwandseinsparung. Andererseits hilft jedoch auch jede explizit dokumentierte Beziehung, Aufwand zu sparen, indem man z.B. schneller und zum Teil sogar automatisiert erkennen kann, welche Codeteile oder Testfälle bei einer Anforderungsänderung anzupassen sind. Eine Möglichkeit der Aufwandsreduktion ist, schon möglichst wenige Beziehungen entstehen zu lassen. Das ist mitunter nicht ganz so einfach. In manchen Unternehmen artet dies dazu aus, dass dann nur mehr die Überschriften bzw. große Gruppen von Requirements verlinkt werden. Zum Beispiel steht dann das Thema »Reporting« mit der Testfallgruppe »Reporting« in Beziehung. Das ist zwar sehr wartungsarm, bringt jedoch praktisch keinen Nutzen mehr und

macht daher auch wenig Sinn. Andererseits gelingt es meist nicht, alle Abhängigkeiten vollständig darzustellen, da dann der Wartungsaufwand der Beziehungen den Nutzen übersteigt.

Hier hilft eine pragmatische und risikobasierte Vorgehensweise: Für Hochrisiko-Requirements sollte versucht werden, möglichst alle Abhängigkeiten zu erfassen und darzustellen. Für alle anderen Risikoklassen sollte dies entsprechend reduziert durchgeführt werden, z. B. nur Anforderungen zu Testfällen und Anforderungen zu Quellcode.

Im Bereich der Beziehungsverwaltung gilt der Grundsatz, dass alles, was einfach, automatisch und toolunterstützt verwaltet und gemanagt werden kann, auch einen Mehrnutzen für das Projekt bedeutet. Beziehungen, die manuell gepflegt werden müssen, sind oft inkonsistent und den investierten Aufwand nicht wert. In diese Kategorie fallen Abhängigkeiten, die z. B. mittels Word oder Excel gepflegt werden. Eine Traceability-Matrix als Excel-Dokument oder direkt eingefügte Verweise in Word- oder Excel-Dokumenten stimmen meist nicht und können nur mit unverhältnismäßig großem Aufwand konsistent gehalten werden.

Es ist daher zu empfehlen, alle Projektartefakte strukturiert in passenden Werkzeugen abzulegen, die auch das Beziehungsmanagement zwischen den Artefakten einfach ermöglichen.

Alle Artefakte und deren Beziehungen sollten in geeigneten datenbankgestützten Werkzeugen verwaltet werden, da dies die Wartung deutlich vereinfacht und eine effiziente Handhabung überhaupt erst möglich macht.

Grundprinzip 5: Änderungen akzeptieren und konsistent umsetzen

Änderungen an Spezifikationen zulassen!

Im Laufe des Projekts ändern sich Anforderungen und eventuell auch Rahmenbedingungen. Lässt man Änderungen zu, ist dies meist mit zusätzlichem Aufwand verbunden. Dies zu ignorieren oder sich dagegenzustellen, hilft leider nicht. Im Gegenteil: Spätestens wenn das Ergebnis dem Auftraggeber präsentiert wird oder das System in Betrieb geht und die Benutzer feststellen, dass es nicht so funktioniert, wie sie sich das vorgestellt haben, werden die Änderungen wieder präsent.

Je später eine Änderung erkannt und umgesetzt wird, desto mehr Aufwand entsteht. Der Versuch, Änderungen nicht zuzulassen, wird meist nicht gelingen bzw. dann zulasten der Beziehung zwischen Auftraggeber und Lieferant gehen. Der Grundsatz kann also nur lauten: Änderungen sind eine gute Sache, wenn wir professionell und effizient mit ihnen umgehen! Solange noch nicht mit der Umsetzung begonnen wurde, sind Änderungen an der Spezifikation ja auch vergleichsweise kostengünstig.

Bei Änderungen alle abhängigen Artefakte konsistent halten

Änderungen sind unvermeidlich. Die Konsequenz daraus ist, dass sie möglichst effizient behandelt werden müssen. Wenn zum Zeitpunkt einer Anforderungsänderung noch nichts programmiert wurde, ist es meist einfach, die Konsistenz herzustellen. Problematisch wird es, wenn Änderungen an schon umgesetzten Teilen erfolgen. Hier wurde schon spezifiziert, programmiert, getestet, eine Benutzerdokumentation erstellt etc. Das bedeutet, dass mit einem nachträglichen Änderungswunsch nicht mehr nur die Spezifikation angepasst werden muss, sondern auch die Architektur, der Code, die Testfälle und die Benutzerdokumentation.

1.6 Umfang des Requirements Engineering – agil vs. klassische Softwareentwicklung

In der klassischen Softwareentwicklung nach V-Modell oder Wasserfall treten die Requirements-Engineering-Aufwände sehr stark konzentriert am Anfang auf und in weiterer Folge wird noch ein gewisser Teil für das Änderungswesen aufgewendet (siehe Abb. 1–8). Insgesamt geht man von einem empfohlenen Aufwandsanteil von 15–20 % des gesamten Projektaufwands aus (in kritischen Projekten auch höher), um das Requirements Engineering auf einem angemessenen Niveau zu betreiben.

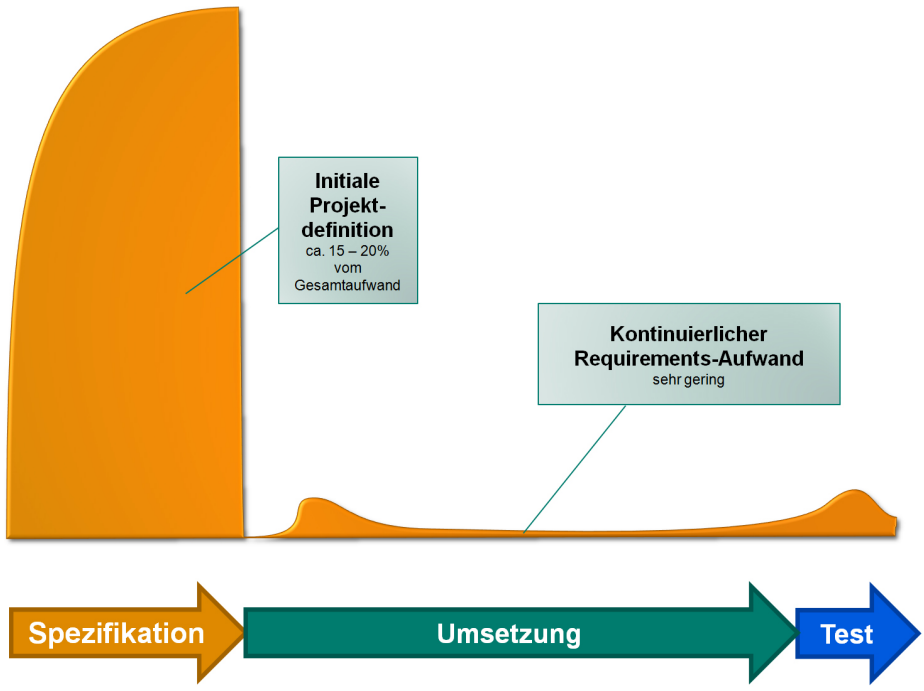

Abb. 1–8 *Requirements-Aufwandsverteilung in klassischen Projekten*

In vielen klassischen Projekten wird diese Empfehlung leider ignoriert und der Aufwand für Requirements Engineering oft auf unter 5 % gedrückt. Das ist eine der Hauptursachen vieler Projektmisserfolge.

In agilen Projekten verteilt sich der Requirements-Aufwand in der Regel anders: Man geht von einer geringfügigen Spitze am Beginn des Projekts für Projektsetup und Definition der groben Anforderungen aus (siehe auch Abschnitt 6.2.4). Oft werden je nach Komplexität und Größe des Vorhabens ein bis drei Startiterationen angesetzt, die zu einem guten Teil für Anforderungsklärung verwendet werden. Um nicht in das klassische Schema »möglichst alles vorab spezifizieren« zu kommen, sollte der initiale Aufwand für Requirements Engineering und Projektsetup bewusst niedrig gehalten werden und einen Anteil von ca. 5–10 % des Gesamtaufwands nicht übersteigen.[4]

Der restliche Requirements-Aufwand ist über alle folgenden Iterationen gesehen ziemlich gleichverteilt, da in jedem Sprint Definition, Klärung und Änderung von Anforderungen stattfinden (siehe Abb. 1–9). Hier sind sowohl die Analyse und Spezifikation der Anforderungen für den bzw. die kommenden Sprints gemeint als auch die Requirements-Aufwände in der Sprint-Planung sowie die dann im laufenden Sprint selbst stattfindende Kommunikation zwischen Product Owner und Team zur Klärung der vielen auftauchenden Fragen. Es wird geschätzt, dass sich das gesamte Team ca. 10 % seiner Zeit über die gesamte Projektlaufzeit mit der Analyse neuer und geänderter Anforderungen beschäftigt.

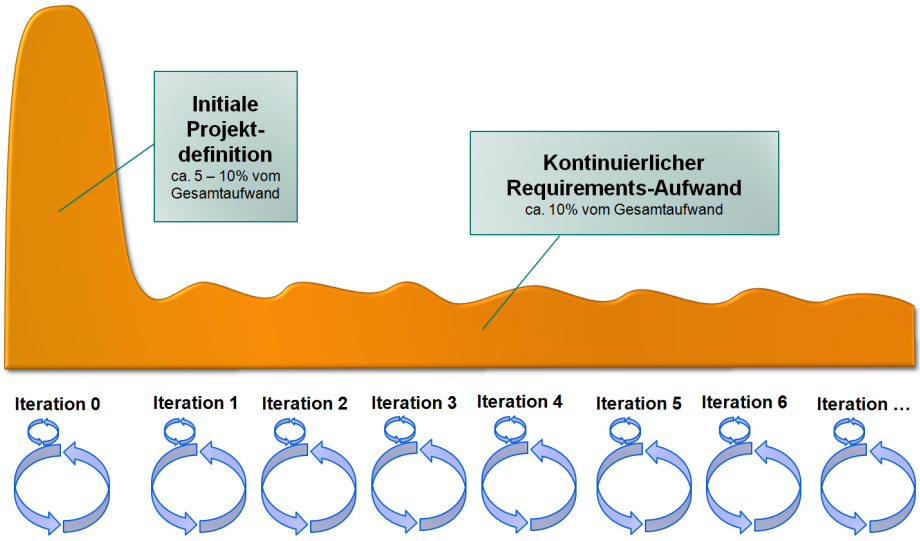

Abb. 1–9 *Requirements-Aufwandsverteilung in agilen Projekten*

4. Wenn es sich nicht um ein Projekt, sondern um eine kontinuierliche Produktentwicklung handelt, kann man von einer gleichmäßigeren Aufwandsverteilung ausgehen. Jedoch gibt es auch hier Aufwandsspitzen z. B. bei großen Umstellungsvorhaben und Neuentwicklungen.

Auch in agilen Vorgehensweisen kann davon ausgegangen werden, dass man durch die Anfangsiterationen und die kontinuierlichen Aufwände für die Requirements-Themen über die gesamte Projektlaufzeit auf einen Aufwand von ca. 15-20% des Gesamtprojektaufwands kommt. Das ist in etwa gleich oder etwas geringer als in klassischem Vorgehen empfohlen. Der wesentliche Unterschied ist, dass die Aufwände anders verteilt sind und zielgerichtet genau dort anfallen, wo sie den größten Nutzen haben.