Peter Hruschka
Kim Lauenroth
Markus Meuten
Gareth Rogers
Stefan Gärtner
Hans-Jörg Steffe

Handbuch
RE@Agile

Aus- und Weiterbildung

zum

IREB Certified Professional for Requirements Engineering

Advanced Level RE@Agile

Practitioner | Specialist

Version 2.0.0
Juli 2022

Nutzungsbedingungen

Wir haben in diesem Dokument bewusst auf die Verwendung gendergerechter Begriffe und Formulierungen verzichtet. Selbstverständlich befürworten wir bei IREB gendergerechte Formulierungen. Wir sehen aber auch die Notwendigkeit, komplexe Sachverhalte so zu formulieren, dass sie gut erfasst werden können.

Die deutsche Sprache hat leider den Nachteil, dass die Bezeichnung von Personen in der Form Feminin, Maskulin und Divers schnell zu langen und komplizierten Sätzen führt. Dadurch werden Texte weniger gut lesbar und somit schwerer zu erfassen. Das Ziel dieses Dokuments ist es aber gerade, Inhalte klar darzustellen und zu vermitteln.

Da wir die Leser*innen dabei unterstützten möchten, den inhaltlichen Fokus zu behalten, verwenden wir in diesem Dokument bewusst nur die maskuline Form von Personen.

Danksagung

Dieses Handbuch wurde erstmals 2018 von Bernd Aschauer, Peter Hruschka, Kim Lauenroth, Markus Meuten und Gareth Rogers erstellt.

Wir bedanken uns bei Rainer Grau für das intensive Review des RE@Agile Lehrplans, bei allen Reviewern dieses Dokuments sowie bei Stefan Sturm, Sibylle Becker und Ruth Rossi für ihre Ermutigung und Unterstützung. Mit Kommentaren von Sacha Reis und Sven van der Zee. Übersetzung aus dem Englischen ins Deutsche von Tanja Scheuermann. Deutsch Review durch Thomas Emmerich, Stefan Gärtner und Hans-Jörg Steffe.

Die Freigabe der englischen Version wurde am 18. Februar 2022 durch das IREB Council auf Empfehlung von Xavier Franch genehmigt.

Allen sei für ihr Engagement gedankt.

Inhaltsverzeichnis

Nutzungsbedingungen 2

Danksagung 2

Inhaltsverzeichnis 3

Vorwort 6

Versionshistorie 7

1. Was ist RE@Agile 8

1.1 Die Geschichte von Requirements Engineering und Agilität 8

1.2 Voneinander lernen 11

1.3 RE@Agile – eine Definition 13

2. Projekte erfolgreich starten 16

2.1 Visionen und Ziele 16

2.1.1 Grundlagen 16

2.1.2 Techniken zur Spezifikation von Vision und Ziel 18

2.1.3 Ändern der Vision und/oder Ziele 23

2.1.4 Festlegen der Systemgrenze Grundlagen 23

2.1.5 Dokumentation der Systemgrenze 26

2.1.6 Die Unvermeidbarkeit von Änderungen des Systemumfangs 31

2.2 Identifizierung und Management von Stakeholdern 31

2.2.1 Grundlagen 31

2.2.2 Identifizierung von Stakeholdern 32

2.2.3 Management von Stakeholdern 35

2.2.4 Andere Quellen für Anforderungen außer Stakeholdern 35

2.3 Zusammenfassung 36

2.4 Fallstudie und Übungen 37

3. Umgang mit funktionalen Anforderungen 39

3.1 Unterschiedliche Stufen der Anforderungsgranularität 39

3.2 Kommunizieren und Dokumentieren auf unterschiedlichen Stufen 42

3.3 Arbeiten mit User-Storys 45

3.3.1 Das 3-C-Modell 45

3.3.2 Eine Vorlage für User-Storys: 46

3.3.3 INVEST: Kriterien für „gute“ Storys 47
3.3.4 Ergänzende Storys mit anderen Anforderungsartefakten 48
3.4 Aufteilungs- und Gruppierungstechniken 48
3.5 Man sollte wissen, wann man aufhören sollte 51
3.6 Projekt- und Produktdokumentation von Anforderungen 52
3.7 Zusammenfassung 54
4. Umgang mit Qualitätsanforderungen und Randbedingungen 56
4.1 Die Bedeutung von Qualitätsanforderungen und Randbedingungen verstehen 57
4.2 Qualitätsanforderungen präzisieren 59
4.3 Qualitätsanforderungen und Backlog 64
4.4 Randbedingungen verdeutlichen 64
4.5 Zusammenfassung 67
5. Priorisieren und Schätzen von Anforderungen 68
5.1 Ermittlung des Geschäftswerts 68
5.2 Geschäftswert, Risiko 71
5.3 Äußern von Prioritäten und Sortieren des Backlogs 72
5.4 Schätzen von User-Storys und anderen Backlog Items 75
5.5 Auswählen einer Entwicklungsstrategie 80
5.6 Zusammenfassung 83
6. Skalierung von RE@Agile 85
6.1 Skalierung von Anforderungen und Teams 85
6.1.1 Organisation umfangreicher Anforderungen 87
6.1.2 Organisieren von Teams 89
6.1.3 Organisieren von Lebenszyklen/Iterationen 91
6.2 Kriterien für die Strukturierung von Anforderungen und Teams im Großen 91
6.2.1 Produktorientiertes Backlog 91
6.2.2 Selbstorganisierte Teams und kollaborative Entscheidungsfindung 93
6.2.3 Verständnis der merkmalbasierten Anforderungsaufteilung 93
6.2.4 Überlegungen, wenn eine Feature-basierte Aufteilung der Anforderungen nicht möglich ist 95
6.2.5 Beispiel eines Telekommunikationsunternehmens 96
6.3 Roadmaps und umfangreiche Planung 99
6.3.1 Darstellen von Roadmaps 100

6.3.2 Synchronisierung von Teams mit Roadmaps 103
6.3.3 Entwicklung von Roadmaps 104
6.3.4 Validierung von Roadmaps 105
6.4 Produkt-Validierung 106
Abkürzungsverzeichnis 108
Literaturverzeichnis 109

Vorwort

Dieses *Handbuch* ergänzt den Lehrplan für CPRE Advanced Level RE@Agile.

Es richtet sich an Ausbildungsanbieter, die Seminare oder Schulungen zum RE@Agile Practitioner und/oder Specialist gemäß IREB-Standard anbieten möchten. Zur weiteren Zielgruppe zählen Schulungsteilnehmer und interessierte Anwender, die einen detaillierten Einblick in den Inhalt dieses Advanced-Level-Moduls erhalten möchten. Weiterhin kann es bei der Anwendung von Requirements-Engineering-Methoden in einer agilen Umgebung nach dem IREB-Standard genutzt werden.

Das Handbuch ist kein Ersatz für Schulungen zu diesem Thema. Es stellt ein Bindeglied zwischen dem Lehrplan (in dem die Lernziele des Moduls aufgeführt und erläutert werden) und der umfangreichen Literatur dar, die zu diesem Thema veröffentlicht wurde.

Ausbildungsanbieter können den Inhalt dieses Handbuchs sowie die Verweise auf weiterführende Literatur zur Vorbereitung von Teilnehmern auf die Zertifizierungsprüfung nutzen. Das Handbuch bietet Schulungsteilnehmern und interessierten Anwendern eine Möglichkeit, ihre Kenntnisse über Requirements Engineering in einer agilen Umgebung zu vertiefen und den detaillierten Inhalt durch die empfohlene Literatur zu ergänzen. Darüber hinaus kann das Handbuch zur Auffrischung bereits vorhandener Kenntnisse in den verschiedenen Themen von RE@Agile genutzt werden, beispielsweise nachdem das RE@Agile Practitioner oder das RE@Agile Specialist Zertifikat erworben wurde.

Über Vorschläge für Verbesserungen oder Korrekturen freuen wir uns!

E-Mail-Kontakt: info@ireb.org

Wir wünschen Ihnen viel Freude beim Studium dieses Handbuchs und einen erfolgreichen Abschluss der Zertifizierungsprüfung zum IREB Certified Professional for Requirements Engineering Advanced Level RE@Agile - Practitioner - oder zum IREB Certified Professional for Requirements Engineering Advanced Level RE@Agile - Specialist.

Weitere Informationen zum IREB Certified Professional for Requirements Engineering Advanced Level Modul RE@Agile finden Sie unter: http://www.ireb.org.

Peter Hruschka

Kim Lauenroth

Markus Meuten

Gareth Rogers

Stefan Gärtner

Hans-Jörg Steffe

Versionshistorie

Version	Date	Comment	Author
1.0.0	11.9.2019	Erste Version, übersetzt auf Basis der englischen Version 1.0.1	Bernd Aschauer, Peter Hruschka, Kim Lauenroth, Markus Meuten und Gareth Rogers
1.0.1	17.12.2019	Umbenennung des Begriffs „Verfeinerungssitzung“ durch den englischen Begriff "Refinement Meeting".	Hans-Jörg Steffe
2.0.0	1.7.2022	Vollständige Neufassung des Kapitels 6; einheitliche Gestaltung der Abbildungen; Fehlerbehebung in den Kapiteln 1-5 (z.B. Ersetzen von "minimal" durch "minimum" in Minimum Viable Product und Minimum Marketable Product, Ersetzen von "Entwicklungsteam" durch "Entwickler"); Berücksichtigung des Advanced Level Splits in Practitioner und Specialist	Peter Hruschka, Kim Lauenroth, Markus Meuten, Gareth Rogers, Stefan Gärtner, Hans-Jörg Steffe

1. Was ist RE@Agile

Gutes Requirements Engineering ist unabhängig von der angewandten Entwicklungsmethodologie ein anerkannter Erfolgsfaktor für die Produkt- oder Systementwicklung.

In diesem Kapitel erhalten Sie einen Überblick über den Hintergrund und die Geschichte von Requirements Engineering und agilen Ansätzen (Kapitel 1.1). Sie erfahren mehr darüber, weshalb diese beiden Disziplinen zuweilen als inkompatibel angesehen werden – was ein gängiges Missverständnis ist. Sie lernen, dass die Techniken und Methoden der Requirements-Engineering-Disziplin – trotz ihrer Geschichte – in bestimmten Entwicklungsansätzen (wie Wasserfall oder Scrum) eingesetzt werden. Zudem erfahren Sie, dass gute Requirements-Engineering-Verfahren die Grundlage für agile Ansätze (wie Scrum, Lean Development und Kanban) sind, damit Produkte und Systeme erfolgreich geliefert werden können.

In Kapitel 1.2 werden die Stärken und Schwächen von Requirements-Engineering-Methoden und von agilen Ansätzen behandelt. Während beim Requirements Engineering das Ermitteln, Verstehen und Dokumentieren der Anforderungen der wichtigsten Stakeholder im Vordergrund steht, um die Entwicklung eines falschen Produkts oder Systems zu vermeiden, liegt bei den meisten agilen Ansätzen das Hauptaugenmerk auf einer vertrauensvollen Zusammenarbeit zwischen den Stakeholdern. In agilen Entwicklungsprozessen werden häufige Feedbackschleifen auf der Basis sichtbarer Ergebnisse eingesetzt, um zu vermeiden, dass falsche Annahmen getroffen werden oder dass sich Missverständnisse zu lange halten können.

Das IREB entwickelte das Advanced-Modul RE@Agile, um die Stärken beider Disziplinen zu vereinen. Wie Sie sich sicher vorstellen können, stehen die Ziele von Requirements-Engineering- und agilen Ansätzen NICHT im Widerspruch zueinander. Vielmehr ergänzen sie sich bei korrekter Anwendung der beiden Methoden.

Im letzten Kapitel 1.3 wird die Definition des IREB von RE@Agile vorgestellt. Kurzgefasst: Sie lernen, wie Ihre Entwicklungsprojekte von diesem integrierten Ansatz profitieren können.

1.1 Die Geschichte von Requirements Engineering und Agilität

Requirements-Engineering-Ansätze und agile Ansätze werden aufgrund ihrer unterschiedlichen Geschichte häufig separat und nicht gemeinsam in Betracht gezogen. Betrachten wir nun einige wichtige Meilensteine in der jeweiligen Geschichte, um die Ursachen für die Entstehung dieser Situation besser zu verstehen. Diese Meilensteine sind in einem Überblick in Abbildung 1 erfasst. (Beachten Sie, dass diese von den Autoren des Handbuchs ausgewählt wurden, um wichtige Quellen für das Advanced-Level-Modul hervorzuheben. In Bezug auf die Geschichte von Entwicklungsmethoden wird kein Anspruch auf Vollständigkeit erhoben.)

In den späten 1970er-Jahren hallte der Begriff „Softwarekrise“ in der gesamten IT-Community wider. Der zentrale Kritikpunkt: Produktentwicklung ist ein komplexer Prozess und die Produkte fallen häufig nicht zur Zufriedenheit der Benutzer aus. Als Antwort darauf entwickelten Wissenschaftler und Methodiker das Wasserfall-Modell (ursprünglich ausgearbeitet von Winston Royce, erlangte es erst durch Barry Boehm größere Bekanntheit). Eines der Mittel gegen die Softwarekrise war die Einführung einer „Anforderungsphase“ vor dem Design, dem Aufbau und dem Test von Systemen. Das Ziel bestand darin, eine Einigung unter den wichtigen Stakeholder darüber zu erreichen, was das Produkt realisieren oder liefern sollte, bevor es hergestellt wurde.

Anforderungsspezifikationen waren in dieser Zeit in erster Linie in natürlicher Sprache verfasste Dokumente. Im selben Zeitraum (Mitte bis Ende der 1970er-Jahre) gab es ebenfalls zahlreiche Vorschläge, Texte durch grafische Modelle zu ergänzen, um die Genauigkeit von Anforderungen zu verbessern und Inkonsistenzen zu vermeiden. 1975 sah Peter Chen in seinen Entity-Relationship-Modellen die Erfassung von geschäftlich relevanten Daten vor.

1978/1979 führten Douglas Ross und Tom DeMarco die strukturierte Analyse- und Design-Technik (SADT) ein, um geschäftliche Funktionen zu erfassen.

Mitte der 1980er-Jahre formulierte Barry Boehm das „Spiralmodell“. Mit der Einführung des Risikomanagements und von häufigeren Feedbackzyklen wurde das Requirements Engineering zu einer iterativen Technik.

Im Hinblick auf Methoden und Notationen war 1992 ein wichtiger Meilenstein für das Requirements Engineering: In diesem Jahr schlug Ivar Jacobson den „Use-Case-orientierten Ansatz“ vor. Er legte den Schwerpunkt auf „Akteure“ (oder Benutzer) im Kontext des Systems und dachte durchgängig an das gesamte Produkt. Diese Ideen waren nicht neu.

Auch McMenamin/Palmer (1984) und Hammer/Champy hoben in ihrer Methodologie „Business Process Reengineering“ diese Art von prozessbezogenem Denken hervor. Die Notation von Ivar Jacobson wurde jedoch sehr populär: einfache Strichmännchen und Ellipsen mit Beschreibungen der Use Cases in natürlicher Sprache.

Einen weiteren wichtigen Meilenstein für das Requirements Engineering bildete der „Unified Process“ von Ivar Jacobson, der als „Rational Unified Process“ (RUP) bekannt wurde. Der RUP erkennt das Requirements Engineering als eine „Disziplin“ an und sieht es nicht nur als eine „Phase“. Diese Disziplin umfasst all diese Phasen (mit unterschiedlicher Gewichtung).

Alle modernen Prozessmodelle haben diese Unterscheidung zwischen Disziplinen (wie Business Analyse, Anforderungen, Design, Implementierung, Testen) und Phasen (wie Konzeptualisierung, Entwurf, Konstruktion, Übergang - wie sie in der RUP-Terminologie bezeichnet werden) übernommen. Letztere ermöglichen machbare Meilensteine während Erstere sicherstellen, dass geeignete Techniken und Praktiken für die laufende Arbeit etabliert sind.

Die internationale Standardisierung der UML (Unified Modeling Language) im Jahr 1997 durch die OMG (Object Management Group) förderte eine größere Bekanntheit von Anforderungsspezifikationen, die Use-Case-Modelle, Aktivitätsdiagramme, Zustandsdiagramme usw. nutzten, vor allem, da viele Werkzeuge diese Notationen unterstützten.

Das Denken in durchgängigen Geschäftsprozessen verbreitete sich noch stärker durch die Standardisierung der BPMN (Business Process Model and Notation). Während die Use Cases von Ivar Jacobson oftmals falsch als „lediglich der IT-Teil der Geschäftsprozesse“ interpretiert wurden, sind die BPMN-Modelle näher am „Geschäft“. Damit wurde ein wichtiges RE-Problem aufgegriffen: die Abstimmung von Geschäft und IT.

Ein weiterer wichtiger Aspekt des Requirements Engineering kam bereits im Jahr 1986 mit der Einführung von FURPS durch HP zur Sprache: die Bedeutung von Qualitätsanforderungen. FURPS war einer der ersten Ansätze, der neben der Funktionalität auch Aspekte der Qualität betonte (FURPS steht für „Functionality“ – Funktionalität, „Usability“ – Benutzbarkeit, „Reliability“ – Zuverlässigkeit, „Performance“ – Effizienz und „Supportability“ – Änderbarkeit).

Dieser wurde durch die ISO/IEC-Norm 9126 verfeinert, mit der viele zusätzliche Qualitätskategorien etabliert wurden, die Systeme erreichen sollten. Die neueste Überarbeitung dieser Norm ist die ISO/IEC 25010 (auch als SQuaRE bekannt – Systems and Software Quality Requirements and Evaluation), in der die Bedeutung von Sicherheit in Systemen hervorgehoben wird.

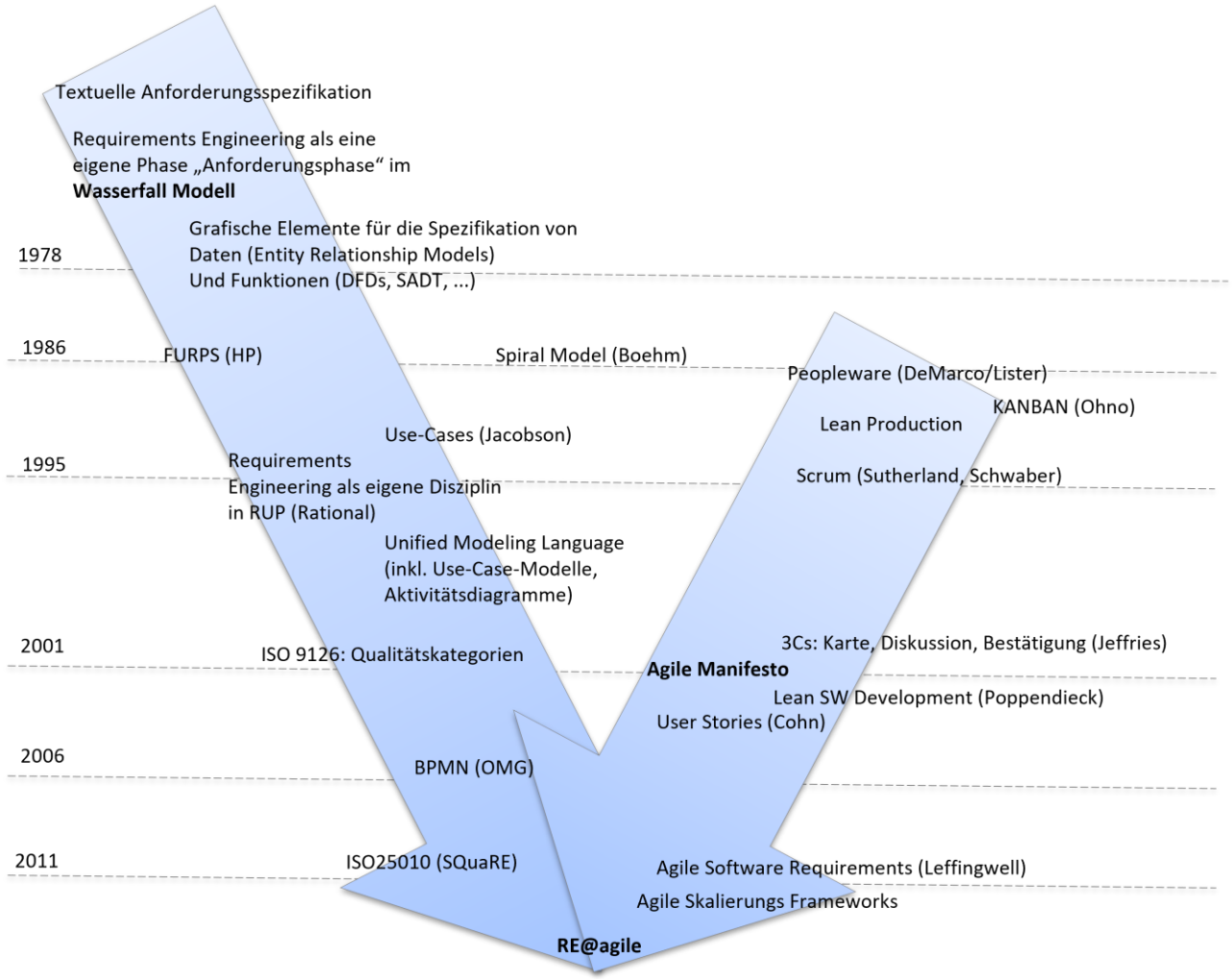

Abbildung 1: Ausgewählte Meilensteine im RE und in agilen Entwicklungsprozessen

Einige zentrale Ideen zu agilen Ansätzen wurden veröffentlicht, lange bevor das Manifest für agile Entwicklungsprozesse 2001 in Erscheinung trat.

1987 prägten Tom DeMarco und Tim Lister den Begriff „Peopleware“, um die Bedeutung der persönlichen Zusammenarbeit und von Teams zu betonen.

Toyota veröffentlichte in den späten 1980er-Jahren Erfolgsgeschichten mit Bezug zu Kanban und Lean Manufacturing (oder Lean Production). Beide Konzepte (Kanban und Lean) sind als Kerngedanken in den agilen Methoden von heute enthalten.

Scrum, „ein Framework zum Entwickeln und Erhalten komplexer Produkte“, wurde erstmals 1995 von Mike Beedle und Ken Schwaber veröffentlicht. Damit wurde die Rolle des „Product Owner“ eingeführt, der in einem Unternehmen für den Erfolg des Produkts verantwortlich ist. Der Product Owner[1] legt die Prioritäten von Anforderungen (häufig als Epics oder Storys bezeichnet) fest. Scrum fand weltweit sehr großen Anklang, was teilweise auf seine Einfachheit (3 Rollen, 4 Artefakte, 5 Meetings) zurückzuführen war.

2001 traf sich in Utah eine Gruppe von 17 Personen, Vertreter gängiger Ansätze wie Extreme Programming, Scrum, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development und Pragmatic Programming, die sich auf ein „Manifest“ einigten.

[1] Der Kürze wegen verwenden wir in diesem Handbuch die Rollenbezeichnung „Product Owner“, wenn wir uns auf die Person beziehen, die für das Management von Anforderungen verantwortlich ist. Eine Definition des Begriffs finden Sie im Glossar.

Dieses „Agile Manifesto“, oder Manifest für agile Entwicklungsprozesse, verlagerte mit zunehmender Bekanntheit den Schwerpunkt von der Systementwicklung, von Verträgen, Dokumenten und langfristiger Planung sowie Prozessen hin zu Zusammenarbeit, Offenheit für Veränderung und Feedback auf Basis häufiger Releases.

Im selben Jahr veröffentlichte Ron Jeffries, einer der Unterzeichner des Agilen Manifests, das 3C-Modell (Card, Conversation, Confirmation – Karte, Diskussion, Bestätigung), in dem er „soziale“ User-Storys von „dokumentarischen“ Requirements-Engineering-Verfahren wie Use Cases abgrenzte.

Wenige Jahre später schlug Mike Cohn ein Format für diese Karten vor: User-Storys. User-Storys betonen drei wichtige Punkte: Wer möchte was und warum. (Als <Art des Benutzers> möchte ich <ein Ziel>, um <ein Grund/Nutzen>).

Da Scrum in erster Linie für kleinere Teams entwickelt worden war (bis zu zehn Mitglieder), wurden ab 2010 immer mehr Skalierungs-Frameworks (beispielsweise SAFe, LeSS, DAD usw.) veröffentlicht, die Möglichkeiten der Zusammenarbeit in größeren oder verteilten Teams beinhalteten.

Dean Leffingwell [Leffingwell2010] prägte den Begriff „Agile Software Requirements“ in seinem 2011 veröffentlichten gleichnamigen Buch, aus dem sich dann der Begriff „Agile Requirements Engineering“ entwickelte. Wenngleich dieser Begriff für die Durchführung von Requirements-Engineering-Aufgaben nach den Prinzipien des Agilen Manifests sehr populär wurde, besteht doch die Gefahr, dass er zu dem Missverständnis führt, es gäbe zwei Arten von Requirements Engineering: klassisches Requirements Engineering und agiles Requirements Engineering. Nach Ansicht des IREB gibt es nur ein gutes oder ein schlechtes Requirements Engineering – in einem nicht-agilen oder einem agilen Umfeld. Deshalb bezeichnen wir den Ansatz des IREB als RE@Agile.

1.2 Voneinander lernen

Agilität und Requirements Engineering sind zwei Disziplinen mit unterschiedlicher Herkunft und klar abgegrenzten Zielen, die jedoch viel voneinander lernen können.

Lassen Sie uns zu Beginn einige zentrale Gedanken des Requirements Engineering betrachten und wie diese von der agilen Denkweise profitieren können. Anschließend sehen wir uns einige grundlegende agile Prinzipien an und besprechen, wie Requirements-Engineering-Techniken diese weiter verbessern können.

Das IREB definiert Requirements Engineering als einen systematischen und disziplinierten Ansatz für die Systemspezifikation mit den folgenden Zielen:

1. Kennen der relevanten Anforderungen, Erreichen eines Konsenses der Stakeholder über diese Anforderungen, das Dokumentieren der Anforderungen nach vorgegebenen Standards und das systematische Management der Anforderungen;
2. Verstehen und Dokumentieren der Wünsche und Bedürfnisse der Stakeholder;
3. Spezifizieren und Managen von Anforderungen, um das Risiko der Auslieferung eines Systems zu minimieren, das nicht den Wünschen und Bedürfnissen der Stakeholder entspricht.

Der erste Punkt – das Kennen der relevanten Anforderungen, bevor man sich mit der Lösung beschäftigt – ist unumstritten. Agilität impliziert jedoch eine sehr klare Auffassung davon, wie „relevant“ interpretiert werden sollte: just in time! Nicht alle Anforderungen sind zu Beginn eines Vorhabens relevant. Eine Ausarbeitung der Vision oder klare Ziele sind für den Anfang ausreichend. Bevor Teile der Lösung entwickelt werden, ist ein gründliches Verständnis dieser Teilmenge von Anforderungen erforderlich. Andere Anforderungen, die für das Unternehmen nicht so dringlich sind, können zunächst diffuser bleiben und zu einem späteren Zeitpunkt verfeinert werden.

Ein Ziel des Requirements Engineering ist es, einen Konsens unter allen Stakeholdern zu erzielen. Wer würde dieses Ziel infrage stellen? Bei einer agilen Herangehensweise soll dieser Konsens durch intensive, vertrauensvolle Zusammenarbeit erreicht werden: Die Stakeholder sollten sich so lange intensiv über die Anforderungen austauschen, bis jeder seine Perspektive verstanden sieht.

Ein weiterer agiler Mechanismus zum Erreichen eines Konsens der Stakeholder besteht in einem raschen Feedback durch nachweisliche Produktinkremente. Oftmals trägt ein sicht- oder greifbares (Teil-) Produktinkrement und die Möglichkeit, dieses zu nutzen, besser dazu bei, offene Probleme festzustellen, als dies mit dem Erstellen umfangreicher, präziser Dokumente, die häufig nicht einmal gelesen werden, möglich wäre.

Im Requirements Engineering sollen die Wünsche und Bedürfnisse der Stakeholder dokumentiert werden. Agilität warnt uns davor, Dokumente nur aus dem Grund der Erstellung von Dokumenten anzufertigen. Das Dokumentieren von Anforderungen (in einer für die Stakeholder adäquaten Form) sollte entweder (1) den Prozess des Erreichens eines Konsenes unterstützen, oder (2) extern auferlegte Randbedingungen (zum Beispiel rechtliche Randbedingungen oder Anforderungen hinsichtlich der Verfolgbarkeit) zufriedenstellen, oder (3) das Definieren von Anforderungen für das nächste Release erleichtern, ohne den Zwang, damit von Grund auf beginnen zu müssen.

Das letzte Ziel in der Definition von Requirements Engineering ist es, Anforderungen so zu managen, dass das Risiko der Auslieferung eines Systems, das nicht den Wünschen und Bedürfnissen der Stakeholder entspricht, minimiert wird. Um das zu erreichen, legen agile Prinzipien nahe, die Priorität und Schätzungen von Backlog Items (Anforderungen) kontinuierlich zu prüfen.

Die Prinzipien der Agilität helfen dabei, das Requirements Engineering in Bezug auf dessen Effizienz, Flexibilität und Zusammenarbeit neu auszurichten. Umgekehrt gibt es zahlreiche Erkenntnisse des Requirements Engineering, von denen auch agile Ansätze profitieren können.

Für agile Entwicklungsprozesse wird nachdrücklich eine vertrauensvolle Zusammenarbeit und Kommunikation zwischen allen relevanten Stakeholdern empfohlen. Dies bedeutet in vielen agilen Methoden in der Regel häufige und offene verbale Kommunikation zwischen Auftraggebern und Benutzern einerseits (denjenigen, die Bedürfnisse oder Anforderungen haben) und Entwicklern andererseits (denjenigen, die Lösungen für die Bedürfnisse und Anforderungen bereitstellen können).

Eine vertrauensvolle Kommunikation ist zwar eine ausgezeichnete Möglichkeit, um ein gemeinsames Verständnis von Anforderungen zu erlangen, aber dies ist bei Weitem nicht der einzige Weg zur Ermittlung von Anforderungen. Das Requirements Engineering hat einen umfassenden Wissensfundus von Ermittlungstechniken entwickelt (z. B. [RoRo2013]), der für die Anwendung in verschiedenen Umgebungen und unter bestimmten Randbedingungen geeignet ist. Zum Beispiel: Kreativitätstechniken wie das Brainstorming helfen etwa beim schnellen Erstellen von Product Backlog Items in innovativen Projekten; dank Produktarchäologie können bei der Arbeit an neuen Versionen bereits vorhandener Produkte schnellere Ergebnisse erreicht werden; Fragebögen können dabei helfen, schnell Feedback von einer großen Anzahl weit verstreuter Stakeholder zu erhalten, die man nie in einen Besprechungsraum bekommen würde.

Agile Product Owner können sehr von einer solchen Bandbreite an Ermittlungstechniken und dem Auswählen einer Reihe geeigneter Techniken profitieren, mit denen sich das Product Backlog schneller füllen lässt als „nur mitreden“.

Durch die Konzentration auf vertrauensvolle Kommunikation wird in agilen Ansätzen der Bedeutung einer präzisen Dokumentation häufig ein geringerer Stellenwert beigemessen. Vor allem bei User-Story-Ansätzen wird hervorgehoben, dass die Karten zum Anzeigen der Storys im Grunde eine Erinnerung an die Diskussion sind und kein Ersatz für präzise Anforderungen (siehe auch Kapitel 3.3). Wir stimmen darüber ein, dass eine rein natürliche Sprache (im Gegensatz zu stärker formalen Notationen von Anforderungen) häufig ein adäquater Weg der gegenseitigen Verständigung darstellt. Natürliche Sprache ist jedoch manchmal nicht präzise genug, um Fehlinterpretationen auszuschließen.

In den letzten Jahrzehnten wurden zahlreiche andere Anforderungsnotationen entwickelt – einschließlich vieler grafischer Notationen –, mit deren Hilfe Stakeholder die fehlende Präzision natürlicher Sprache überwinden können. Einige Geschäftsprozesse lassen sich unter Umständen einfacher diskutieren, wenn Techniken wie Aktivitätsdiagramme, Datenflussdiagramme oder die Business Process Model and Notation (BPMN) eingesetzt werden, anstatt Karten, auf denen die einzelnen Prozessschritte notiert werden. Manche zu behandelnde Objekte lassen sich mitunter einfacher mithilfe von Informationsmodellen entwerfen und einige zustandsorientierte Systeme können von Zustandsmodellen profitieren, wenn zu klären ist, welche Aktivitäten in welchem Zustand ausgeführt werden sollen. Um es noch einmal hervorzuheben: Product Owner und Entwickler sollten solche Notationen kennen – nicht um der Anwendung eines Formalismus willen, sondern um Diskussionen zu verkürzen.

Ein weiteres Credo der Agilität ist die häufige Bereitstellung funktionierender Software, also iteratives Arbeiten und Erstellen einer Folge von Produktinkrementen. Es ist allerdings nicht sinnvoll, mit iterativer Entwicklung zu beginnen, wenn das Team nicht auf eine gemeinsame Vision oder eine Reihe von Zielen eingestimmt ist. Für einen einzelnen Scrum Product Owner, der die volle Kontrolle über ein Produkt hat, mag es einfach sein, eine Vision oder eine Reihe von Zielen zu haben. Muss sich der Product Owner aber mit mehreren „wichtigen“ Stakeholdern abstimmen, dann sollte jeglicher Detailarbeit an Anforderungen eine Analyse der Stakeholder, ein Abgleich der Ziele und die Definition des Systemumfangs vorausgehen. Diese Aktivitäten sind in der Idee eines „erfolgreichen Projektstarts“ enthalten, der in Kapitel 2 vorgestellt wird.

Fasst man die Überlegungen dieses Kapitels zusammen, so lässt sich festhalten, dass Agilität uns dabei hilft, eine Kultur für erfolgreiche Produktentwicklung zu schaffen. Das Requirements Engineering macht sie flexibler und effizienter und sie bewirkt, dass der Zusammenarbeit mehr Bedeutung beigemessen wird. Aus Anforderungssicht sind die Eckpfeiler der Agilität eine vertrauensvolle Zusammenarbeit aller Stakeholder und das Streben nach kurzfristigen inkrementellen Ergebnissen. Techniken wie das Erfassen von User-Storys auf Story-Cards funktionieren gut. Es gibt jedoch viele andere Techniken der Anforderungsermittlung und -validierung, die über Jahrzehnte hinweg von der Requirements-Engineering-Forschung entwickelt wurden und die Product Owner und ihren Entwicklungsteams zu noch mehr Produktivität verhelfen können – natürlich bei korrekter Anwendung und ohne formalistische Übertreibung.

Eine unserer Schlussfolgerungen im RE@Agile Primer [Primer2017] lautete: „Der wichtigste Wert ist dem Requirements Engineering und Agilität gemein: die Endbenutzer des Produkts durch eine Lösung zufriedenstellen, die ihren Bedürfnissen entspricht oder die größten Probleme behebt.“

In diesem Advanced-Level-Modul gehen wir noch mehr ins Detail, um zu zeigen, wie Ideen aus beiden Welten zusammen zum Erreichen dieses Ziels genutzt werden können. Wir finden es hilfreich, in der folgenden Definition von RE@Agile zunächst unsere eigenen Grundprinzipien für den Rest dieses Handbuchs darzulegen.

1.3 RE@Agile – eine Definition

RE@Agile ist ein kooperativer, iterativer und inkrementeller Ansatz mit vier Zielen:

1. Die relevanten Anforderungen in einem angemessenen Detaillierungsgrad zu kennen (zu jedem Zeitpunkt während der Systementwicklung);
2. Eine ausreichende Einigung der relevanten Stakeholder über die Anforderungen zu erzielen;
3. Die Anforderungen gemäß den Rahmenbedingungen der Organisation zu erfassen (und zu dokumentieren);
4. Alle auf Anforderungen bezogenen Aktivitäten gemäß den Prinzipien des Agilen Manifests durchzuführen.

Wie zuvor erwähnt, verwenden wir gemäß Scrum-Terminologie den Begriff Product Owner als Rolle, die für den kooperativen Ansatz und damit für gutes Requirements Engineering verantwortlich ist.

Betrachten wir die folgenden zentralen Gedanken dieser Definition näher:

1. RE@Agile ist ein kooperativer Ansatz:
   „Kooperativ" betont die Idee der Agilität in Form einer intensiven Interaktion mit den Stakeholdern, welche durch regelmäßige Inspektionen des Produktstands und des hieraus resultierenden Feedbacks gekennzeichnet ist. Hierdurch wird eine kontinuierliche Nachschärfung und Klärung von Anforderungen ermöglicht, welche sich aus dem kontinuierlichen Lernen ergibt.
2. RE@Agile ist ein iterativer Prozess:
   Daraus lässt sich die Idee der „Just in Time"-Anforderungen ableiten. Die Anforderungen müssen vor Beginn der Arbeit am technischen Design und an der Implementierung nicht vollständig sein. Stakeholder können diese Anforderungen iterativ im erforderlichen Detaillierungsgrad definieren (und verfeinern), wenn die zu implementierenden Anforderungen zeitnah umgesetzt werden sollen.
3. RE@Agile ist ein inkrementeller Prozess:
   Vom ersten Inkrement an werden diejenigen Anforderungen implementiert, welche den größten Geschäftswert versprechen oder die höchsten Risiken mindern sollen. In den ersten Inkrementen wird die Erstellung eines Minimum Viable Product (MVP) oder eines Minimum Marketable Product (MMP) angestrebt. Ab diesem Zeitpunkt können dem Produkt die nächsten Inkremente hinzugefügt werden, wobei stets die Anforderungen herausgegriffen werden, die den größten Geschäftswert versprechen. Auf diese Weise wird der Geschäftswert des resultierenden Produkts konstant gesteigert.

Das erste Ziel („relevante Anforderungen, die in angemessenem Detaillierungsgrad bekannt sind") bezieht sich auch auf den iterativen Ansatz: Als „relevant" gelten die Anforderungen, die zeitnah implementiert werden sollen. Diese Anforderungen müssen sehr präzise verstanden worden sein (einschließlich der zugehörigen Akzeptanzkriterien) – insbesondere auf Seiten der Entwickler.

Sie müssen der „Definition of Ready" (DoR) genügen. Andere Anforderungen, die noch nicht die oberste Priorität haben, können zunächst auf einer allgemeineren Abstraktionsebene gehalten und erst detaillierter betrachtet werden, wenn sie an Bedeutung gewinnen.

Voraussetzung für das zweite Ziel („ausreichende Einigung der relevanten Stakeholder") ist es, alle Stakeholder und deren Relevanz für das zu entwickelnde System zu kennen. Die für die Anforderungen verantwortliche Person (gewöhnlich der Product Owner) muss die Anforderungen mit den relevanten Stakeholdern verhandeln und daraus die Implementierungsreihenfolge ableiten.

Bei agilen Herangehensweisen wird die intensive und laufende Kommunikation über Anforderungen mehr wertgeschätzt als die Kommunikation über die Dokumentation. Dennoch wird im dritten Ziel betont, wie wichtig Dokumentation in einem geeigneten Detaillierungsgrad ist (welcher von der jeweiligen Situation abhängt). Organisationen müssen die Dokumentation von Anforderungen unter Umständen aufbewahren (aus rechtlichen Gründen, zur Verfolgbarkeit, für Wartungszwecke usw.). In diesen Fällen muss bei agilen Ansätzen sichergestellt werden, dass die geeignete Dokumentation erstellt wurde. Diese sollte jedoch nicht vorab erstellt werden. Es kann Zeit und Aufwand sparen, die notwendige Dokumentation parallel zur Implementierung oder sogar erst nach der Implementierung zu erstellen. Außerdem ist es hilfreich, Artefakte wie Daten-, Aktivitäts- oder Zustandsmodelle als vorläufige Dokumentation zu erstellen, um die Diskussion über Anforderungen zu unterstützen.

Das Anforderungsmanagement umfasst sämtliche Aktivitäten, die auszuführen sind, wenn die Anforderungen sowie anforderungsbezogene Artefakte einmal vorliegen. In agilen Entwicklungsprozessen sind die meisten Aktivitäten des Anforderungsmanagements im Verfeinerungsprozess der Backlog Items enthalten.

Das klassische Anforderungsmanagement umfasst jedoch auch die Attribuierung von Anforderungen, das Versions- und Konfigurationsmanagement sowie die Nachverfolgbarkeit der Anforderungen und anderer Entwicklungsartefakte. Die Empfehlung gemäß RE@Agile ist, diesen Aufwand zu minimieren oder die Balance zwischen Aufwand und Nutzen zu halten:

- Umfangreiches Versionsmanagement lässt sich durch schnelle Iterationen zur Erstellung von Produktinkrementen ersetzen (der Änderungsverlauf von Anforderungen seit ihrem ersten Auftreten ist zum Beispiel von geringerem Interesse als ihr aktuell gültiger Status).
- Die iterative Ableitung von Sprint Backlogs umfasst das Konfigurationsmanagement (Baselining), d.h. das Gruppieren der Anforderungen, die aktuell den höchsten Geschäftswert versprechen.

Aus diesem Grund ersetzt man einige der zeitintensiven (und auch papierintensiven) Aktivitäten des Anforderungsmanagements nicht-agiler Ansätze durch Aktivitäten, die auf agilen Prinzipien beruhen. Andere Aktivitäten werden gut durch Werkzeuge unterstützt, die das automatische Verfolgen von Beziehungen zwischen Anforderungen und des Verlaufs erleichtern, ohne dass dafür zusätzlicher personeller Einsatz nötig ist.

In den nächsten Kapiteln dieses Handbuchs gehen wir näher auf die verschiedenen Aspekte von RE@Agile ein.

In Kapitel 2 behandeln wir die Voraussetzungen für eine erfolgreiche Systementwicklung: das Abwägen von Visionen und/oder Zielen, Stakeholdern und des Systemumfangs.

In den Kapiteln 3 und 4 gehen wir auf den Umgang mit funktionalen Anforderungen, Qualitätsanforderungen und Rahmenbedingungen auf verschiedenen Granularitätsstufen ein.

In Kapitel 5 besprechen wir die Prozesse des Schätzens, Ordnens und Priorisierens von Anforderungen zur Festlegung der Reihenfolge der Inkremente.

In den Kapiteln 2 bis 5 liegt der Schwerpunkt auf dem Umgang mit Anforderungen einer Gruppe von Entwicklern (von 3 bis 9 Personen).

In Kapitel 6 wird die Skalierung des Requirements Engineering für größere, potenziell verteilte Teams dargestellt, einschließlich der übergeordneten Produktplanung und Roadmap-Erstellung.

2. Projekte erfolgreich starten

Es hat sich in vielen Unternehmen bewährt, vor Beginn eines wichtigen Projekts einen Workshop vorzubereiten. Dazu zählt beispielsweise das Vorbereiten und Sammeln der benötigten Materialien, das Bereinigen und Sortieren der Werkzeuge, das Entfernen von Überflüssigem aus dem vorherigen Projekt und die Verständigung über die Eckpfeiler des anstehenden Projekts. Dieses Vorgehen mag aufgrund des immateriellen Wesens von Software unpassend oder altmodisch erscheinen. Doch das Gegenteil ist der Fall.

Der Großteil der Arbeit bei der Softwareentwicklung besteht aus geistiger Arbeit oder reinem Nachdenken. Das heißt, der überwiegende Anteil der Arbeit ist, verglichen mit herkömmlichem Handwerk, von außen nicht wahrnehmbar. In einer Werkstatt oder auf einer Baustelle hingegen sind Fehler für andere häufig sichtbar und können daher umgehend korrigiert werden. Ein Denkfehler fällt nur dann auf, wenn das Ergebnis des Denkvorgangs in irgendeiner Weise sichtbar ist. Das Ergebnis kann dann von einer Person oder einem System als falsch erkannt werden, was zu einem Verständnis oder der Identifikation des Denkfehlers führt.

Bei agilen Herangehensweisen ist man sich dieser Problematik häufig nicht bewusst. Die Beteiligten sind der Meinung, dass direkte Kommunikation und schnelle Feedbackzyklen ausreichen. Aber auch wenn sie wirklich wertvoll und nützlich sind, ausreichend sind sie nicht. Ein Beispiel: Wenn der Gesamtzusammenhang und andere sichtbare Artefakte bei Entwicklungsbeginn fehlen, lassen sich auch durch direkte Kommunikation und schnelle Feedbackzyklen keine mehrfachen Nacharbeiten verhindern.

Das in diesem Kapitel dargestellte Konzept eines erfolgreichen Projektstarts enthält wichtige Voraussetzungen für eine erfolgreiche iterative, inkrementelle Systementwicklung.

Sie erfahren, dass ein erfolgreicher Projektstart aus drei Aktivitäten bestehen sollte, die drei greifbare Ergebnisse hervorbringen, die zur Steuerung der iterativen Arbeit verwendet werden können:

- Definition der Vision des Systems und/oder der Ziele, die damit erreicht werden sollen
- Identifizierung des derzeit bekannten Systemumfangs und der Systemgrenze
- Identifizierung der relevanten Stakeholder und anderer wichtiger Anforderungsquellen

In den nächsten Kapiteln erfahren Sie Näheres zu den einzelnen Aktivitäten und den zugehörigen Techniken. Am Ende dieses Kapitels stellen wir die Fallstudie iLearnRE mit praktischen Übungen für den erfolgreichen Projektstart vor. Auf die Fallstudie greifen wir in den folgenden Kapiteln als fortlaufendes Beispiel für weitere Übungen zurück.

2.1 Visionen und Ziele

2.1.1 Grundlagen

Die Produktvision und/oder die Ziele des Produkts sind für jede Entwicklungsaktivität von höchster Bedeutung. Sie geben die allgemeine Richtung für die Entwicklung vor und dienen als Orientierung für sämtliche anderen Aktivitäten. Der Auslöser für die Vision und/oder die Ziele sind entweder Probleme oder unzufriedenstellende Umstände, die in der aktuellen Umgebung anzutreffen sind, Veränderungen in der Umgebung, die uns zu Reaktionen zwingen (beispielsweise die Einführung neuer gesetzlicher Vorschriften), oder innovative Ideen, die ein größeres oder besseres Geschäft versprechen.

Wir verwenden die beiden Begriffe Vision und Ziele synonym. Im Rahmen agiler Methoden wird häufig der Begriff „Vision“ bevorzugt, in Requirements-Engineering-Ansätzen ist dagegen gewöhnlich von „Zielen“ die Rede. Beide Begriffe können als abstrakteste Formulierung dessen betrachtet werden, was ein System erreichen sollte. Alle Teammitglieder und alle relevanten Stakeholder sollten die definierte Vision und die Ziele kennen, um zu verstehen, was das Team anstrebt.

Der Product Owner ist für die Formulierung der Vision und/oder der Ziele verantwortlich. Außerdem ist er dafür verantwortlich, den Teammitgliedern die Details näherzubringen. Die Tatsache, dass der Product Owner verantwortlich ist, bedeutet nicht, dass er Vision oder Ziele alleine definieren muss. In der Regel diskutiert er die Vision und/oder die Ziele mit den relevanten Stakeholdern und holt deren Input und Feedback ein.

Ein häufiger Stolperstein beim Definieren ist die Wahl der falschen Perspektive, das heißt, man formuliert eine Vision und/oder Ziele, die etwas über das aktuell entwickelte System oder einen Teil davon aussagen. Ein Beispiel dafür ist die folgende Aussage:

> „Erstellen einer Website zum Kaufen und Lesen von elektronischen Büchern und Hörbüchern."

Diese Vision beschreibt ein System (eine Website) zum Kaufen und Lesen von elektronischen Büchern. In Abhängigkeit von den Gegebenheiten kann die Entwicklung eines solchen Systems eine gute Idee sein, oder aber auch nicht. Die Aussage ist jedoch viel zu restriktiv, um eine gute Formulierung einer Vision zu werden, da sie das System charakterisiert, anstatt anzugeben, was durch die Entwicklung des Systems erreicht werden soll. Bei der folgenden Aussage wird eine andere Perspektive gewählt:

> „Verkauf von elektronischen Büchern und Hörbüchern an Personen weltweit (über eine Internetverbindung) und die Möglichkeit, das gekaufte Buch sofort zu lesen/anzuhören."

Diese Aussage ist im Vergleich zu der vorherigen aus mehreren Gründen besser:

1. Es wird definiert, was das System erreichen soll und nicht die Funktion des Systems.
2. Die Aussage konzentriert sich auf die Vorteile des Systems für Menschen (und die Nutzer).
3. Das Kaufen von elektronischen Büchern/Hörbüchern, wo immer die Nutzer sich befinden und das unmittelbare Lesen/Hören des Buchs.
4. Die Aussage definiert nicht vorab die Art des Systems.
5. Eine Website ist nicht unbedingt die richtige Lösung zum Lesen von elektronischen Büchern/Anhören von Hörbüchern.

Der größte Nachteil der Formulierung von Visionen und Zielen, die sich auf das System selbst konzentrieren anstatt auf dessen Auswirkungen, ist, dass solche Formulierungen den Lösungsspielraum für das Team direkt von Beginn des Projekts an einschränken. Es gilt als Faustregel, Referenzen auf das aktuell entwickelte System (und der Begriff „System" an sich) in einer Visions- oder Zielformulierung zu vermeiden.

Visionen und Ziele sind in der Regel an einen Zeithorizont gebunden. Dieser Zeithorizont legt den Zeitraum fest, in dem eine Vision oder ein Ziel erreicht werden soll. Aus diesem Grund empfehlen wir die Definition von Visionen und Zielen immer mit einem Zeitraum (oder auch einem bestimmten Datum) zu verbinden. Es ist nicht nötig, den Zeitraum direkt in die Formulierung der Vision oder Ziele aufzunehmen, er sollte lediglich allen Teammitgliedern und Stakeholdern klar sein.

In agilen Methoden wird die Definition von Visionen und/oder Zielen für die einzelnen Iterationen empfohlen. Daher kann es unterschiedliche Formulierungen für verschiedene Zeiträume geben. Eine System- oder Produktentwicklung kann langfristige Ziele (oder strategische Visionen) haben, zum Beispiel für die nächsten drei Jahre. Diese können wiederum in Ziele unterteilt werden, die in bestimmten Jahren zu erreichen sind. Und bei einer iterativen Entwicklung sollte man selbstverständlich auch Ziele haben, die in der nächsten Iteration erreicht werden sollen.

Das Definieren von Visionen oder Zielen mit einer langfristigen Perspektive hat den Vorteil, dass die Teammitglieder und sämtliche Stakeholder ein klares Verständnis vom Gesamtzusammenhang haben und von dem Zeitrahmen, in dem dieser erreicht wird. Dieser Vorteil lässt sich gut anhand des früheren Beispiels mit dem Buchshop verdeutlichen.

Die formulierte Vision kann man folgendermaßen unterteilen:

Gesamtvision: „Verkauf von elektronischen Büchern und Hörbüchern an Personen weltweit (über eine Internetverbindung) und die Möglichkeit, das gekaufte Buch sofort zu lesen/anzuhören.“

- Ende des sechsten Monats: Elektronische Bücher an Personen weltweit verkaufen und diesen ermöglichen, das gekaufte Buch sofort zu lesen.
- Ende des ersten Jahres: Hörbücher an Personen weltweit verkaufen und diesen ermöglichen, das Hörbuch sofort anzuhören.
- Ende des zweiten Jahres: Elektronische Bücher und Hörbücher kombiniert an Personen weltweit verkaufen und diesen Personen ermöglichen, das Buch sofort und zur selben Zeit zu lesen und anzuhören.

In den unterteilten Visionen ist ein klarer Zeitrahmen für das Projekt und die wichtige Information enthalten, dass es ein Paket aus elektronischen Büchern und Hörbüchern geben wird, welche der Leser/Hörer gleichzeitig anhören und auch lesen kann. Diese Informationen sind für das Team sehr wichtig, da es das System zum Lesen elektronischer Bücher so konzipieren soll, dass später im Prozess noch Hörbücher hinzugefügt werden können. Darüber hinaus kann das Team jetzt Feedback zur Beantwortung folgender Frage geben: Ist es realistisch, die drei Ziele innerhalb des definierten Zeitrahmens umzusetzen?

2.1.2 Techniken zur Spezifikation von Vision und Ziel

Im vorherigen Kapitel haben Sie die Grundlagen für die Definition von Vision und/oder Zielen kennengelernt. In diesem Kapitel erfahren Sie mehr über spezifische Techniken, die Ihnen bei der Entwicklung und Definition von Vision und/oder Zielen Unterstützung bieten. Welche Art und Weise man auch wählt: Jeder Stakeholder hat das Recht zu wissen, was das Team anstrebt. Daher muss die Definition der Vision und der anfänglichen Ziele zu Beginn einer Entwicklungstätigkeit erfolgen.

2.1.2.1 SMART

SMART ist ein Akronym, das sich auf eine vereinfachte Art des Verfassens von Zielen bezieht, das George T. Doran [Doran1981] 1981 entwickelt hat.

Doran zufolge steht das Akronym für:

- Specific (spezifisch) – zielt auf einen bestimmten Verbesserungsbereich ab;
- Measurable (messbar) – quantifiziert oder enthält zumindest einen Vorschlag für einen Fortschrittsindikator;
- Assignable (zuweisbar) – legt fest, wer es realisiert;
- Realistic (realistisch) – gibt an, welche Ergebnisse mit den verfügbaren Ressourcen realistischerweise erreicht werden können;
- Time-related (zeitbezogen) – gibt an, wann die Ergebnisse erzielt werden können.

Diese Originaldefinition wurde von der Agile-Community in verschiedener Weise angepasst. Aus Perspektive des Requirements Engineering eignet sich die folgende Definition:

- Specific (spezifisch) – zielt auf einen bestimmten Verbesserungsbereich ab;
- Measurable (messbar) – quantifiziert oder enthält zumindest einen Vorschlag für einen Fortschrittsindikator;

- Achievable (erreichbar), statt „Assignable“ – Festlegen eines Ziels, das für das Team erreichbar ist;
- Relevant, statt „Realistic“ – Festlegen eines Ziels, das für die Stakeholder relevant ist;
- Time-bound (zeitbezogen) – gibt an, wann die Ergebnisse erzielt werden können.

Diese Änderung berücksichtigt zwei Ideen der agilen Entwicklung:

1. Ziele sollten auf die Erreichbarkeit durch das Team ausgerichtet sein, ohne den Fokus auf die Ressourcen zu legen. Ressourcen werden nicht geplant; sie werden nach Priorisierung zugewiesen.
2. Die Relevanz des Ziels, das heißt, der mit dem Ziel verbundene Wert, ist wichtiger als die Frage der Erreichbarkeit in Bezug auf die verfügbaren Ressourcen.

Wir verwenden noch einmal das vorherige Beispiel, um die Anwendung von SMART zu verdeutlichen:

> „Verkauf von elektronischen Büchern und Hörbüchern an Personen weltweit (über eine Internetverbindung) und die Möglichkeit, das gekaufte Buch sofort zu lesen/anzuhören.“

Diese Aussage erfüllt die SMART-Kriterien wie folgt:

Kriterium	**Beispiel**
Specific	Die Erfahrung beim Kaufen und Konsumieren von elektronischen Büchern und Hörbüchern wird verbessert.
Measureable	Das messbare Ergebnis besteht darin, elektronische Bücher und Hörbücher weltweit zu kaufen (über eine Internetverbindung) und diese sofort zu lesen/anzuhören.
Achievable	Das Internet und die Mobilfunktechnologie können das gewünschte Ergebnis erbringen.
Relevant	Elektronische Bücher und Hörbücher sind bei vielen ein beliebtes Medium.
Time-bound	Der Zeitrahmen ist detailliert festgelegt (Details siehe oben).

Die SMART-Kriterien können entweder als Vorlage oder als Checkliste für die Formulierung von Zielen genutzt werden. Beim Ansatz mit Vorlage beschreiben Sie explizit jedes einzelne Element der SMART-Kriterien. Die obige Tabelle ist ein Beispiel für diesen Ansatz. Der Nachteil des Vorlagen-Ansatzes ist, dass dabei üblicherweise Redundanzen in der Formulierung erzeugt werden.

Beim Ansatz mit Checkliste prüfen Sie anhand der SMART-Kriterien, ob Ihre Zielformulierung alle Aspekte abdeckt.

Eine gute Kombination dieser beiden Ansätze sähe so aus: Sie treffen mithilfe der SMART-Vorlage eine Entscheidung und verwenden das Ergebnis dazu, ein präzises Ziel zu definieren, das einfach zu kommunizieren ist.

2.1.2.2 PAM

Bei PAM handelt es sich um ein alternatives Kriterienset für die Formulierung von Zielen, die von [Robertson2003] entwickelt wurde. Die Kriterien sind folgendermaßen definiert:

- Was ist der Zweck (Purpose – P)?
- Was ist der betriebliche Vorteil/Mehrwert (Business Advantage – A)?
- Wie würde man diesen Vorteil messen (Measure – M)?

Die PAM-Kriterien konzentrieren sich auf den Geschäftswert eines Ziels und lassen die zeitliche Perspektive der SMART-Kriterien außen vor. Verwendet man diesen Ansatz in einem frühen Stadium, hat das den Vorteil, dass man sich auf die verschiedenen Werte konzentriert, anstatt die zeitliche Perspektive in die Zieldefinition zu pressen.

Wenn wir noch einmal das zuvor angeführte Beispiel betrachten, sehen wir, dass die PAM-Kriterien dabei nicht vollständig erfüllt werden. Das wird in der folgenden Tabelle deutlich:

Kriterium	**Beispiel**
Zweck	Die Erfahrung beim Kaufen und Konsumieren von elektronischen Büchern und Hörbüchern wird verbessert.
Business Advantage	Der betriebliche Vorteil/Mehrwert ist nicht explizit angegeben.
Measure	Das messbare Ergebnis besteht darin, elektronische Bücher und Hörbücher weltweit zu kaufen (über eine Internetverbindung) und diese sofort zu lesen/anzuhören.

Der betriebliche Vorteil/Mehrwert ist in unserem Beispiel nicht klar. Ein betriebswirtschaftlicher Vorteil könnte beispielsweise so lauten:

- Konsumenten kaufen mehr elektronische oder Hörbücher, wenn diese umgehend verfügbar sind;
- Konsumenten kaufen mehr elektronische oder Hörbücher, wenn sie reisen, da die Bücher weltweit verfügbar sind.

Ebenso wie die SMART-Kriterien, können auch die PAM-Kriterien als Vorlage oder Checkliste für die Formulierung von Zielen genutzt werden.

2.1.2.3 Product Vision Box

Die Idee, die SMART und PAM zugrunde liegt, ist die Definition expliziter Kriterien, wodurch die Formulierung von Zielen unterstützt wird. Diese Kriterien sind nützlich, wenn Sie viele Informationen gesammelt haben und diese in richtige Ziele und/oder eine echte Vision strukturieren möchten.

Eine andere Möglichkeit, Visionen und Ziele zu definieren, ist die Product Vision Box, die von [Highsmith2001] entwickelt wurde. Der Product Vision Box liegt die Idee zugrunde, dass man ein *physisches Paket* für sein Produkt erstellt, das die zentralen Vorteile und Ideen des Produkts für potenzielle Kunden in einem Geschäft präsentiert.

Ein halbtägiger Workshop ist ein gängiges Format für die Product Vision Box. Laden Sie dazu die wichtigsten Stakeholder ein, wenn möglich aus dem gesamten Spektrum der an dem Produkt Beteiligten (beispielsweise Endbenutzer, Marketing, technische Mitarbeiter).

Stellen Sie den Workshop-Teilnehmern Pappkartons, verschiedene Arten von Materialien (beispielsweise Papier, Stifte, Buntstifte, Whiteboard-Stifte, Aluminiumfolie, Draht) und Medienmaterial (beispielsweise Zeitungen, Zeitschriften, Fotos) zur Verfügung.

Die Agenda des Workshops sollte aus abwechselnden Arbeits- und Präsentationsphasen bestehen. In der Arbeitsphase erstellt ein Team aus Workshop-Teilnehmern (3 bis 4 Personen) eine oder mehrere Boxen (Pakete).

In der Präsentationsphase werden den Teilnehmern die Boxen ohne jegliche Erklärung präsentiert. Jeder Teilnehmer kann sich seine Gedanken über die einzelnen Boxen machen. Anschließend präsentieren die Ersteller die Grundidee, die hinter ihrer Box steht, und es wird darüber diskutiert.

Personen, die nicht am Workshop teilgenommen haben, können optional zur letzten Präsentationsphase eingeladen werden. Auf diese Weise erhält die Gruppe externes Feedback, was die Auswirkungen von Gruppendenken verringert.

Der wichtigste Vorteil der Product Vision Box besteht darin, dass man über die Produktidee vom endgültigen Ergebnis ausgehend zurück nachdenkt. Ein Produktpaket enthält typischerweise Informationen über die wichtigsten Features oder Vorteile eines Produkts. Ein solcher Ansatz unterstützt implizit die Konzentration auf Ziele und die Gesamtvision. Den Teilnehmern bereitet diese Herangehensweise in der Regel großen Spaß, da sie ein greifbares Ergebnis hervorbringt, das später als eine Art Referenzpunkt für die weitere Diskussion genutzt werden kann.

Ein häufiger Einwand gegen die Product Vision Box ist, dass es Arten von Produkten gibt, die nicht in einfachen Paketen verkauft werden können. Diese können aber nach agilen Methoden entwickelt werden. Hierzu ein Beispiel von außerhalb des IT-Bereichs: Projekte zu organisatorischen Veränderungen betreffen verschiedene Problemgebiete und Bedürfnisse und erfordern daher mehrdimensionale Lösungen, die nicht in eine Box passen.

2.1.2.4 Nachrichten aus der Zukunft

Eine weitere Technik für die Formulierung von Visionen und Zielen ist das Verfassen eines kurzen Zeitungsartikels über Ihr Produkt, der aus der Zukunft stammt (siehe [HeHe2011]). Diese Technik wurde von Techniken für die Persönlichkeitsentwicklung abgeleitet, die Personen dazu anregen sollen, vom Lebensende ausgehend über ihr Leben nachzudenken, indem sie beispielsweise ihren eigenen Nachruf schreiben.

Mit den Neuigkeiten aus der Zukunft lassen sich verschiedene Themen und Titel abdecken. Gute Ausgangspunkte können etwa Folgendes sein:

- Erfolgreiche Produktpräsentation – schreiben Sie einen Artikel aus der Perspektive eines Journalisten, der an Ihrer erfolgreichen Produktpräsentation teilgenommen hat. Erwähnen Sie Features, Eindrücke oder Ideen, die diesem Journalisten besonders gut an Ihrem Produkt gefallen haben;
- Alles Gute zum 10. Jahrestag – stellen Sie sich vor, dass Ihr Produkt seinen 10. Jahrestag feiert und ein Journalist einen Zeitungsartikel darüber schreibt. Gehen Sie in dem Bericht auf die Hochs und Tiefs ein und beschreiben Sie, welche Auswirkungen das Produkt auf das Leben der Menschen oder auf Ihr Geschäft hatte;
- Bericht über negative Produktentwicklung – stellen Sie sich vor, Ihr Produkt scheitert und ein Journalist schreibt darüber. Erläutern Sie die Gründe, die zum Scheitern geführt haben, und denken Sie darüber nach, welches Wissen über Ihre Kunden Ihnen gefehlt hat, welche Features vermisst werden oder welche Qualitätsprobleme das Produkt haben könnte.

Der daraus resultierende Artikel kann analysiert werden, um potenzielle Visionen und Ziele zu identifizieren. Er dient auch als guter Ausgangspunkt für weitere Aktivitäten. Anschließend können Sie beispielsweise mithilfe der SMART- oder PAM-Kriterien eine präzise Formulierung der Vision und/oder Ziele erstellen.

Die Technik „Nachrichten aus der Zukunft“ kann von Einzelnen oder als Gruppenübung in einem Workshop durchgeführt werden. In der Gruppenübung sollten die Teilnehmer recht kurze Artikel schreiben, die im Rahmen des Workshops gelesen und besprochen werden können.

2.1.2.5 Vision Boards

Der Begriff „Vision Board“ bezieht sich auf eine Klasse von Techniken, mit denen eine strukturierte grafische Darstellung der Vision und/oder der Ziele auf einer physischen Tafel erstellt wird. Dahinter verbirgt sich die folgende Grundidee:

a) Die Tafel bzw. das Board bietet eine inhalts- oder zeitbezogene Struktur, um den gesamten Umfang der Vision und/oder der Ziele für die Stakeholder zu visualisieren;

b) Das Vision Board wird als lebende Einheit betrachtet, die ständig angepasst wird, um das aktuelle Verständnis aller Stakeholder abzubilden;

c) Es ist für alle Stakeholder der Single Point of Truth in Bezug auf die Vision und/oder Ziele.

Ein ganz einfaches Beispiel eines Vision Board besteht aus drei Spalten:

- Die kurzfristige Vision und zugehörige kurzfristige Ziele: Was möchten wir in der nahen Zukunft erreichen (beispielsweise in vier Wochen)?
- Die mittelfristige Vision und zugehörige mittelfristige Ziele: Was möchten wir mittelfristig erreichen (beispielsweise in sechs Monaten)?
- Die langfristige Vision und zugehörige langfristige Ziele: Was möchten wir langfristig erreichen (beispielsweise in drei Jahren)?

Ein zweites, strukturorientiertes Beispiel eines Vision Board ist das von [Pichler2016] definierte „Product Vision Board“. Es besteht aus den folgenden Elementen:

- Vision: Was motiviert Sie zur Herstellung des Produkts? Welche positive Veränderung sollte es herbeiführen?
- Zielgruppe: Welchen Markt oder welches Marktsegment bedient das Produkt? Wer sind die Zielkunden und Benutzer?
- Bedürfnisse: Welche Probleme löst das Produkt? Welche Vorzüge bietet es?
- Produkt: Um welches Produkt handelt es sich? Wodurch hebt es sich von anderen Produkten ab? Ist die Entwicklung des Produkts realisierbar?
- Geschäftliche Ziele: Wie profitiert das Unternehmen von diesem Produkt? Welche geschäftlichen Ziele werden damit verfolgt?

2.1.2.6 Canvas-Techniken

Der Begriff „Canvas-Technik“ bezeichnet eine Reihe von Techniken, mit denen eine strukturierte Übersicht über die verschiedenen Aspekte eines Produkts bereitgestellt werden soll. Canvas-Techniken ähneln Vision-Board-Techniken, haben aber in der Regel einen größeren Umfang und konzentrieren sich nicht ausschließlich auf die Vision und/oder Ziele des Produkts. Dennoch sind die Vision und/oder die Ziele immer Teil eines Canvas und sie werden in Verbindung mit den anderen, durch das Canvas abgedeckten Aspekten entwickelt.

Aufgrund dieses breiteren Umfangs gibt es bei der Anwendung von Canvas-Techniken mehr Slots als bei einem Vision Board, weshalb für die Dokumentation aller Aspekte bei Canvas-Techniken mehr Platz benötigt wird. Dies hat zur Verwendung des Begriffs „Canvas“ (Leinwand) geführt, da ein Canvas viel größer als ein Board (eine Tafel) sein kann. Dennoch ähnelt die allgemeine Idee, die Canvas-Techniken zugrunde liegt der von Vision Boards.

Ein häufiges Beispiel für die Canvas-Technik ist das „Business Model Canvas“ von [OsPi2011]. Die Idee dahinter ist, das Wertversprechen eines Unternehmens oder Produkts, die Infrastruktur, die Kunden und die Finanzen zu beschreiben.

Ein weiteres Beispiel ist das Canvas für Chancen („Opportunity Canvas“) von [Patton2015]. Bei diesem Canvas geht man von einem bereits vorhandenen Produkt aus, das verbessert werden muss.

2.1.3 Ändern der Vision und/oder Ziele

Im Laufe einer Entwicklungstätigkeit können sich Ziele aufgrund neuer Stakeholder oder eines neuen oder veränderten Verständnisses des Systems oder des Kontexts ändern. Daher sollte die Dokumentation von Vision und/oder Zielen regelmäßig aktualisiert werden. Techniken wie das Vision Board ermöglichen eine physische Darstellung der Vision und/oder Ziele und sie ermöglichen eine einfache Kommunikation der geänderten Ziele.

Änderungen der Vision oder der Ziele sollten explizit dokumentiert werden, einschließlich der Beweggründe für deren Änderung. Eine formale Dokumentation dieser Änderungen ist nicht nötig. Es gibt folgende schlanke Möglichkeiten zur Dokumentation von Änderungen:

- Ein Zeitplaner oder Tagebuch (analog oder mit einem Werkzeug) für die Vision und/oder Ziele, in dem sämtliche Änderungen mit Datum und Begründung dokumentiert werden.
- Ein Foto des Vision Board (oder eine andere Darstellung), einschließlich Notizen zu den Änderungen.

Diese Dokumentation sollte als allgemeine Erinnerung der Vision und/oder Ziele betrachtet werden. Sie ist nützlich, um Änderungen abzubilden und die Häufigkeit von Änderungen zu erkennen. Die Häufigkeit ist eine wichtige Metrik: Treten Änderungen vor allem in späteren Phasen der Produktentwicklung zu häufig auf, sollte das als Indikator angesehen werden, dass die Produktentwicklung insgesamt in Gefahr ist, da es keine klare Gesamtrichtung für das Produkt gibt.

2.1.4 Festlegen der Systemgrenze Grundlagen

Das Konzept „Systemgrenze“ umfasst eine Reihe von Begriffen, die präzises Denken und eine unmissverständliche Dokumentation des Umfangs und Kontexts des Systems ermöglichen. Für ein richtiges Verständnis des Begriffs „Systemgrenze“ ist zunächst das Verständnis der Begriffe „Systemumfang“ und „Kontext“ erforderlich.

Die folgenden Definitionen sind auch im Glossar des IREB enthalten:

- Systemgrenze: die Grenze zwischen einem System und dem umgebenden (System-) Kontext.
- Kontext: der Teil einer Systemumgebung, der für die Anforderungen des geplanten Systems relevant ist.
- Systemumfang: Die gesamte Bandbreite von Aspekten, die bei der Entwicklung eines Systems geformt und konzipiert werden können.

Manchmal darf der Kontext eines Systems nicht verändert werden und die Systemgrenze ist nicht verhandelbar.

Typische Beispiele sind:

- Die Ersetzung einer technischen Komponente innerhalb eines größeren bestehenden Systems. Beispielsweise muss in einer vorhandenen Kfz-Produktionslinie aufgrund veränderter gesetzlicher Bestimmungen die Softwarekomponente einer eingebetteten Steuerungseinheit ausgetauscht werden. Die Fahrzeuge werden bereits genutzt und die Komponente muss zu den vorhandenen Schnittstellen und der Hardware passen. Eine Änderung dieser Aspekte ist nicht möglich.
- Die Entwicklung eines Systems innerhalb eines vorhandenen Systems. Ein Versicherungsunternehmen betreibt zum Beispiel ein Webportal für Kunden, über das es Versicherungsverträge verwaltet. Das Unternehmen hat entschieden, eine Smartphone-App als zweiten Kanal für seine Kunden zu entwickeln. Die App soll dieselben Funktionen bieten wie das Webportal. In dem Entwicklungsprojekt für die App dürfen keine Änderungen am Portal oder an den Schnittstellen zu anderen Systemen vorgenommen werden.

In vielen Entwicklungsprojekten sind der Systemumfang und die Systemgrenze jedoch verhandelbar. Das heißt, dass Bestandteile des Kontexts im Laufe der Entwicklungstätigkeit gegebenenfalls tatsächlich geändert werden. Diese Aussage mag abstrakt und theoretisch erscheinen, sie hat allerdings eine enorme Auswirkung auf jede Entwicklung. Von Beginn an muss klar sein, welche Elemente des Systemkontexts angepasst werden können und welche unverändert bleiben müssen.

Eine typische Situation ist die Verbesserung eines Geschäftsprozesses durch ein neues System. Eine Bank möchte beispielsweise die papierbasierte Beantragung neuer Konten durch eine Webportal-Lösung ersetzen[2]. Bei dem bestehenden Prozess sendet der potenzielle Kunde das Antragsformular per Post an die Bank. Ein Bankmitarbeiter genehmigt den Antrag und richtet das Bankkonto ein, indem er die Daten in das Banksystem eingibt. Das neue System bietet potenziellen Kunden webbasierte Anträge: Die Kunden geben ihre Daten in das Formular ein und senden diese an die Bank. Sofort nach der Datenübermittlung erhalten die Kunden eine Bestätigung ihres Antrags per E-Mail. Das ist die beabsichtigte Änderung im Systemkontext (potenzielle Kunden füllen kein Papierformular mehr aus, sie verwenden stattdessen die webbasierte Anwendung).

Der interessantere Teil dieses Beispiels ist der Prozess im Backoffice. Hier sind drei Szenarios denkbar:

1. Die Antragsdaten werden per E-Mail an den Bankmitarbeiter gesendet. Der Bankmitarbeiter führt den bestehenden Genehmigungsprozess durch und gibt die Daten manuell in das System der Bank ein.

[2] *Anmerkung*: Die Beschreibung dieses Beispiels ist absichtlich unvollständig. Auf den folgenden Seiten decken wir weitere fehlende Aspekte auf, um die Vorteile der verschiedenen Techniken darzustellen. Wenn Sie denken, dass Sie bereits einige fehlende Aspekte entdeckt haben, notieren Sie diese und sehen Sie im weiteren Verlauf, ob wir Ihre Ansicht teilen.

2. Der Bankmitarbeiter führt den Genehmigungsprozess im neuen Webportal durch. Er prüft die Antragsdaten im Webportal. Wenn er den Antrag akzeptiert, nutzt das Webportal eine neue Schnittstelle zum Banksystem, damit das Bankkonto automatisch eingerichtet wird.
3. Der Genehmigungsprozess wird automatisch vom neuen Webportal ausgeführt. Das Webportal wird mit einer regelbasierten Genehmigungs-Engine ausgestattet, die eine automatische Genehmigung von Standardanträgen ermöglicht. Wird der Antrag genehmigt, richtet das Webportal automatisch das Bankkonto ein. Falls er nicht genehmigt wird, muss er von einem Bankmitarbeiter geprüft werden.

Das ist natürlich ein stark vereinfachtes und unvollständiges Beispiel, es zeigt aber die Auswirkungen der Entscheidung zum Systemumfang. Im ersten Szenario ist der Systemumfang auf das Webportal, den neuen Antragsprozess und die E-Mail-Übertragung von Antragsdaten an den Mitarbeiter begrenzt. Im zweiten Szenario umfasst der Systemumfang außerdem die Art und Weise, wie der Bankmitarbeiter im Backoffice arbeitet und wie die Datenübertragung an das Banksystem funktioniert. Die Entscheidung über den Antrag bleibt jedoch beim Mitarbeiter. Im dritten Szenario ist sogar der Entscheidungsprozess zum Bestandteil des Systemumfangs für das Projekt geworden.

Welches der drei Szenarios für die betreffende Bank geeignet ist, geht aus unserem Beispiel nicht hervor. Dies hängt von verschiedenen Faktoren ab, die im Zuge der Entwicklungstätigkeit herausgearbeitet werden müssen.

Am Beispiel des Banksystems wird deutlich, dass ein gemeinsames Verständnis des Systemumfangs und des -kontexts die Voraussetzung für eine effektive und effiziente Entwicklung sind. Missverständnisse in Bezug auf die Systemgrenze oder den Systemumfang können zu Folgendem führen:

- Entwicklung von Funktionalitäten oder Komponenten, die nicht im Verantwortungsbereich der Entwicklungstätigkeit lagen. Unser Bankprojekt hat beispielsweise mit der Entwicklung der regelbasierten Genehmigungs-Engine begonnen (Szenario 3), die Stakeholder haben eine solche Genehmigungs-Engine jedoch nie vereinbart. Wenn die Stakeholder beschließen, dass die Genehmigungs-Engine nicht erforderlich ist, dann ist der Entwicklungsaufwand dafür verloren.
- Die falsche Annahme, dass Funktionalitäten oder Komponenten, die tatsächlich Bestandteil des Systems sind, außerhalb des Systems entwickelt werden sollten (der angenommene Systemumfang war zu klein). In unserem Beispielprojekt des Banksystems wurde etwa die E-Mail-Übertragung von Antragsdaten an Mitarbeiter implementiert (Szenario 1), der Stakeholder hatte aber erwartet, dass die Genehmigung im Webportal ausgeführt werden würde (Szenario 2).

Die System- und die Kontextgrenze lassen sich durch die Diskussion folgender Aspekte definieren:

a) *Welche Features oder Funktionalitäten muss das System und welche der Kontext bereitstellen?* Diese Frage zielt auf die Systemgrenze ab, indem sie die Diskussion über konkrete Funktionsmerkmale des Systems in Gang setzt. Darf das System in unserem Bankprojekt beispielsweise einen Antrag automatisch genehmigen oder nicht (Szenario 2 oder 3)? Ein weiteres Diskussionsthema könnte die Einrichtung eines neuen Bankkontos sein: Soll das neue System diese Aufgabe durchführen oder nicht (Szenario 1 oder 2)?

b) *Welche technischen Schnittstellen oder Benutzeroberflächen muss das System dem Kontext bereitstellen?* Diese Frage zielt auf die Systemgrenze ab und ist eng mit der vorangehenden Frage zu Feature/Funktionalität verbunden. Für viele Funktionalitäten sind Schnittstellen zu Benutzern oder anderen Systemen nötig. In unserem Bankprojekt ist beispielsweise eine Schnittstelle zum Banksystem erforderlich, damit neue Bankkonten automatisch eingerichtet werden können (Szenario 2). Für die Genehmigung des Bankmitarbeiters im Webportal (Szenario 2) ist zudem eine Benutzeroberfläche zum Anzeigen und Genehmigen der Antragsdaten erforderlich.

c) *Welche Aspekte des Kontexts sind für das System relevant/irrelevant?* Diese Frage zielt auf die Kontextgrenze ab, indem sie explizit Aspekte des Kontexts aufgreift, die bei der Systementwicklung geprüft werden müssen. In unserem Bankprojekt zählen beispielsweise das Antragsformular und der Prozess zum Senden der Daten an die Bank definitiv zum Kontext. Die Einrichtung des Bankkontos kann im Rahmen des Kontexts (Szenario 2 und 3) oder außerhalb des Rahmens liegen (Szenario 1).

d) *Welche Aspekte des Systemkontexts können während der Systementwicklung geändert werden?* Diese Frage zielt auf den Systemumfang ab, indem festgelegt wird, welche Aspekte des Kontexts geändert bzw. nicht geändert werden dürfen. Es ist zu beachten, dass ein Element, das im Systemumfang enthalten ist, per definitionem Teil des Systemkontexts ist. In unserem Bankprojekt ist es beispielsweise denkbar, dass die Genehmigungsentscheidung beim Bankmitarbeiter liegen soll (wodurch Szenario 3 unmöglich wird).

Alle vier Fragen stehen selbstverständlich in engem Zusammenhang und müssen zusammen diskutiert werden. Beachten Sie, dass die Kontextgrenze immer unvollständig ist, da sie sich nur durch die Faktoren definieren lässt, die explizit vom Systemkontext ausgeschlossen werden. In ähnlicher Weise ist der Systemumfang niemals endgültig und er kann sich während der Entwicklungstätigkeit verändern. Eine wichtige Erkenntnis aus Perspektive des Requirements Engineering ist, dass Änderungen an Systemumfang, Systemgrenze und Kontextgrenze allen relevanten Stakeholdern explizit mitgeteilt werden müssen.

2.1.5 Dokumentation der Systemgrenze

Der Systemumfang und die Systemgrenze lassen sich mithilfe verschiedener Techniken klären und dokumentieren. In diesem Kapitel stellen wir vier davon dar: Kontextdiagramme, natürliche Sprache, Use-Case-Diagramme und Story-Maps.

2.1.5.1 Kontextdiagramm

Das Kontextdiagramm ist ein Element der strukturierten Systemanalyse (Essential System Analysis [McPa1988]), für das Diagramme zur Darstellung des Kontexts genutzt werden. Es dokumentiert das System, Aspekte des Kontexts und deren Beziehung. Die Notation eines Kontextdiagramms besteht aus drei Modellierungselementen:

- Das betrachtete System (Kreis);
- Aspekte des Kontexts (Rechtecke);
- Pfeile zur Darstellung der Verbindungen zwischen den Elementen. Die Richtung eines Pfeils stellt den Informationsfluss dar.

Die folgende Abbildung zeigt die Kontextdiagramme für alle drei Szenarios des Portals für Bankkontoanträge.

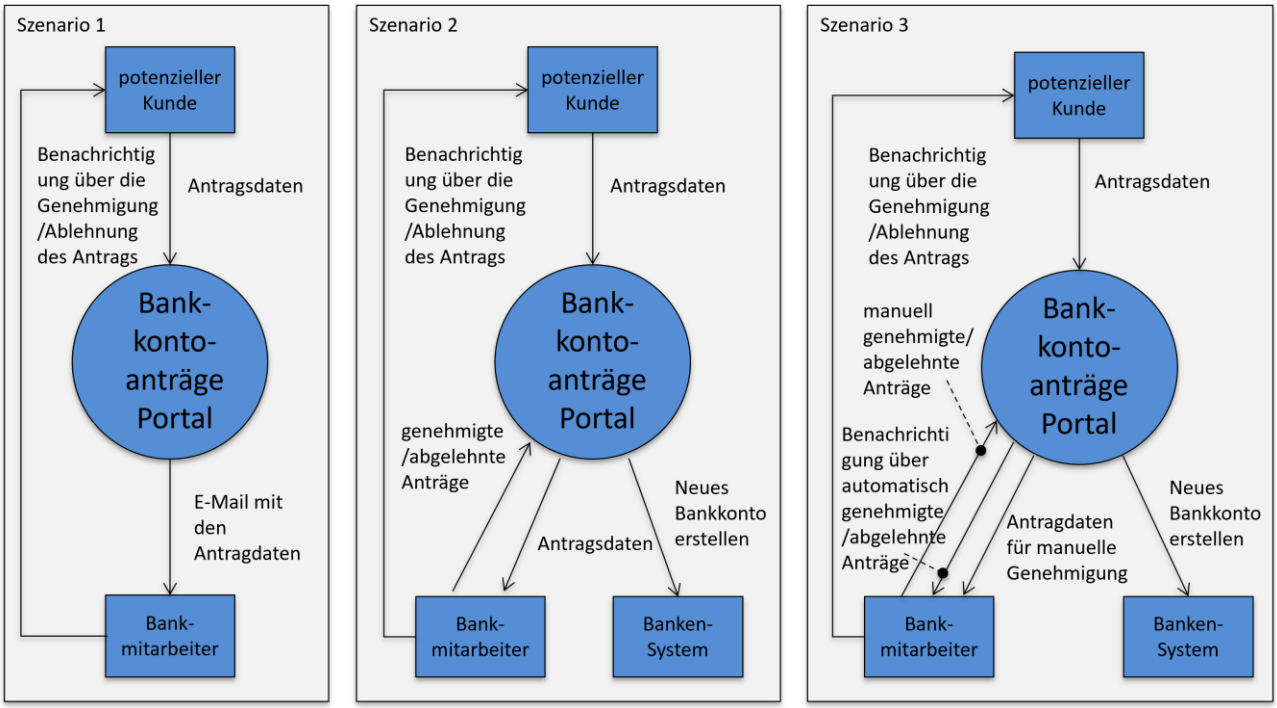

Abbildung 2: Drei Kontextdiagramme für unser Beispiel des Portals für Bankkontoanträge

Alle drei Szenarios berücksichtigen die Beziehung zwischen dem potenziellen Kunden und dem Portal (der Kunde sendet die Antragsdaten an das Portal) und die Beziehung zwischen dem Bankmitarbeiter und dem potenziellen Kunden (der Bankmitarbeiter sendet dem potenziellen Kunden eine Benachrichtigung über die Genehmigung/Ablehnung des Antrags). Das Dokumentieren der zweiten Beziehung (zwischen dem potenziellen Kunden und dem Bankmitarbeiter) ist nicht Teil des ursprünglichen Kontextdiagramms. Es ist jedoch nützlich, diese Beziehung in der Praxis zu dokumentieren, da dies klar kennzeichnet, dass das System nicht für das Senden dieser Benachrichtigung verantwortlich ist.

Die Kontextdiagramme für die Szenarios 2 und 3 enthalten beide die Beziehung mit dem Banksystem, um im Falle einer Genehmigung das neue Bankkonto einzurichten. Die Beziehung ist kein Bestandteil von Szenario 1, da in diesem Fall der Bankmitarbeiter das Konto manuell einrichtet.

Man könnte argumentieren, dass die Beziehung zwischen dem Bankmitarbeiter und dem Banksystem ebenfalls im Kontextdiagramm für Szenario 1 dokumentiert werden könnte. Es gibt Argumente dafür und dagegen:

- Dafür spricht: Das Einrichten des Bankkontos ist Teil des gesamten Geschäftsprozesses (Eröffnen eines Bankkontos). Deshalb muss dies dokumentiert werden, damit der Gesamtkontext klar wird.
- Dagegen spricht: Das Einrichten des Bankkontos wurde für das Szenario 1 als nicht im Systemumfang enthalten definiert. Daher sollte es nicht dokumentiert werden.
- Beide Argumente sind nachvollziehbar und berechtigt. Die Entscheidung für oder gegen die Dokumentation solcher Beziehungen hängt vom Gesamtkontext des Projekts ab.

Der Hauptunterschied zwischen den drei Szenarios ist die Beziehung zwischen Bankmitarbeiter und Portal. Im Szenario 2 erhält der Bankmitarbeiter alle Antragsdaten und muss diese genehmigen oder ablehnen. Im Szenario 3 erhält der Bankmitarbeiter nur die Anträge, über die nicht automatisch entschieden werden kann. Zusätzlich bringt das Kontextdiagramm für Szenario 3 einen neuen und bislang fehlenden Aspekt zum Vorschein: Der Bankmitarbeiter erhält eine Benachrichtigung über automatisch genehmigte/abgelehnte Anträge. Diese Information benötigt er, um dem potenziellen Kunden eine Benachrichtigung zu senden.

Obwohl das Beispiel mit dem Portal stark vereinfacht ist, unterscheiden sich die Kontextdiagramme für alle drei Szenarios erheblich und bieten einen einfachen Überblick über das System und den Kontext.

2.1.5.2 Natürlichsprachige Dokumentation von Systemumfang und Systemgrenze

Natürliche Sprache ist die flexibelste und benutzerfreundlichste Technik für die Dokumentation von Systemumfang und Systemgrenze. Sie müssen lediglich eine Liste der System-Features/Funktionalitäten und eine Liste mit weiteren Aspekten erstellen, um den Systemkontext zu dokumentieren (denken Sie daran, auch Aspekte zu dokumentieren, die nicht im Umfang enthalten sein sollen). Dokumentieren Sie den Systemumfang mithilfe einer zusätzlichen Liste.

Die Dokumentation von Systemumfang und Systemgrenze von Szenario 1 des Bankprojekts ließe sich durch die folgende Liste darstellen.

Systemumfang und Systemgrenze des Portals für Bankkontoanträge (Szenario 1)

Features/Funktionalitäten des Systems:

- webbasiertes Formular für die Beantragung eines Bankkontos
- Senden einer E-Mail an den Kunden zur Empfangsbestätigung des Antragsformulars
- Senden der Antragsdaten per E-Mail an das Backoffice der Bank

Im Kontext enthaltene Aspekte:

- Kunde, der ein Bankkonto beantragen möchte

Im Systemumfang enthaltene Aspekte:

- Prozess des Ausfüllens eines Antragsformulars (ausgeführt vom Kunden)
- Prozess der Übermittlung von Antragsdaten an den Bankmitarbeiter

Aspekte außerhalb des Kontexts:

- Bankmitarbeiter im Backoffice, der den Antrag genehmigt (oder ablehnt)
- Einrichtung des Bankkontos (bei Genehmigung des Antrags)
- Senden der Antragsgenehmigung an den Kunden (wenn der Antrag genehmigt wird)
- Senden der Ablehnung des Antrags an den Kunden (wenn der Antrag nicht genehmigt wird)

Vergleicht man diese Liste mit der Beschreibung von Szenario 1 in Kapitel 2.2.1, so tritt ein neuer Aspekt in Erscheinung, der bis dahin noch nicht behandelt wurde: Die Beschreibung enthält keine Informationen zur Genehmigung oder Ablehnung. Es ist unklar, wie der Kunde über den genehmigten oder abgelehnten Antrag informiert wird.

Der vorangehenden Liste zufolge ist dieser Aspekt nicht im Kontext enthalten. Die Entwicklung muss sich daher nicht um dieses Thema kümmern. Ohne diese explizite Aussage, ist die Wahrscheinlichkeit sehr hoch, dass verschiedene Stakeholder unterschiedliche Erwartungen daran haben, wie eine Genehmigung oder Ablehnung mit dem neuen System gehandhabt würde.

2.1.5.3 Use-Case-Diagramm

Das Use-Case-Diagramm ist Teil der UML. Mit dieser Art von Diagramm lassen sich die Akteure und Use Cases eines Systems modellieren. In einem Use Case wird das Verhalten eines Systems aus der Perspektive eines Benutzers (oder anderer externer Akteure oder beispielsweise anderer Systeme) festgelegt: In jedem Use Case wird eine Funktionalität beschrieben, die das System für die beteiligten Akteure bereitstellen muss.

Bei Use-Case-Diagrammen liegt das Augenmerk detailliert auf Akteuren und deren zugehörigen Funktionen. Dies ist für die Klärung des Systemumfangs und -kontexts sehr nützlich. Die folgenden Notationselemente von Use-Case-Diagrammen sind für die Modellierung des Systemkontexts hilfreich:

- Systemgrenze (Rechteck mit der Systembezeichnung)
- Akteur (Strichmännchen mit dem Namen darunter oder Rechteck mit dem Namen)
- Use Case (Ellipse mit der Use-Case-Bezeichnung)
- Beziehung zwischen Use Case und Akteur (Linie)

Use-Case-Diagramme bieten außerdem Notationselemente für die Modellierung von Beziehungen zwischen Use Cases (beispielsweise zur Erweiterung und Aufnahme von Beziehungen). Mithilfe der Notationselemente lassen sich detaillierte Beziehungen zwischen Use Cases dokumentieren. Dieser Detaillierungsgrad ist in der Regel für die anfängliche Klärung des Systemkontexts nicht sehr hilfreich. Die folgende Abbildung zeigt Use-Case-Diagramme für alle drei Szenarios des Portals für Bankkontoanträge.

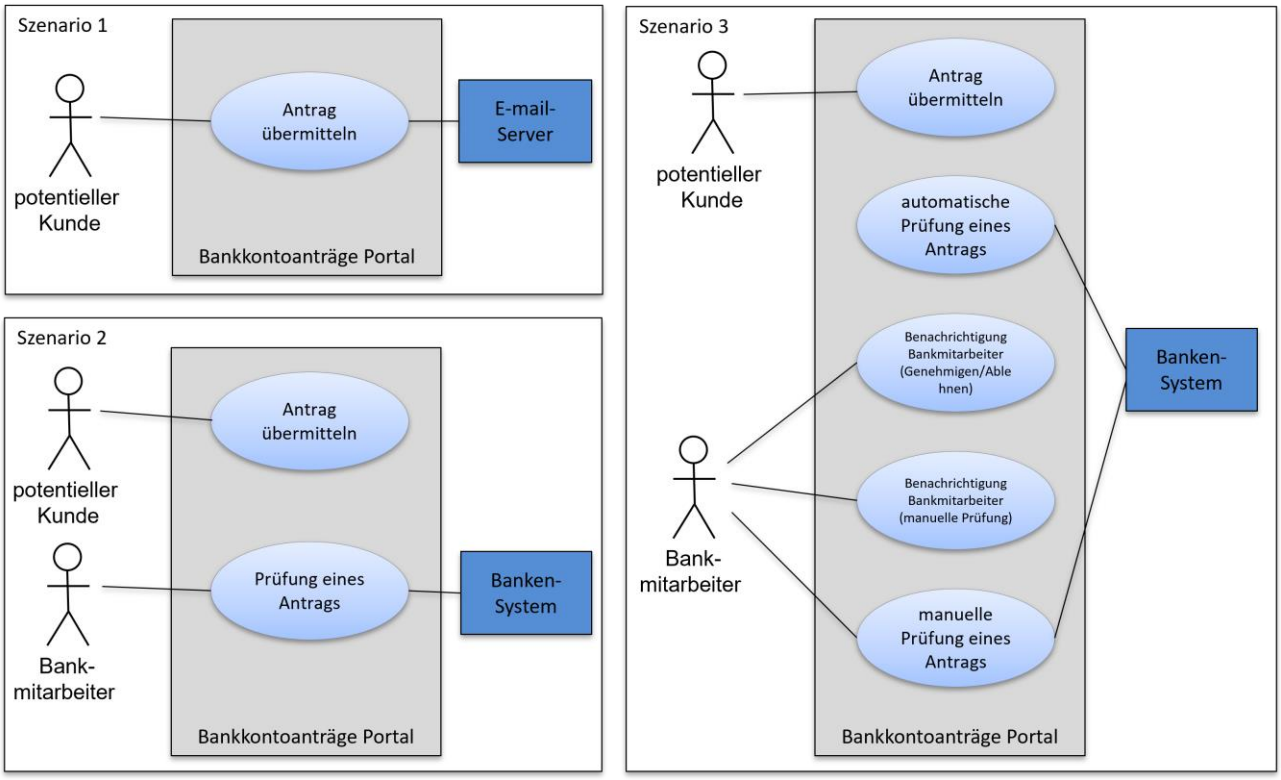

Abbildung 3: Drei Use-Case-Diagramme für das Beispiel des Portals für Bankkontoanträge

Die Use-Case-Diagramme bieten auf den ersten Blick eine Übersicht darüber, wie die funktionale Komplexität der drei Szenarios zunimmt. Szenario 1 ist sehr einfach gehalten (ein Use Case), Szenario 3 ist dagegen am komplexesten (fünf Use Cases).

Die Bezeichnungen der Use Cases vermitteln die zentralen Informationen des Use-Case-Diagramms. Daher ist es sehr wichtig, sorgfältig geeignete Bezeichnungen für die Use Cases auszuwählen. Vergleicht man die Use-Case-Diagramme für die Szenarios 2 und 3, dann erkennt man, dass die Use Cases für die „Prüfung eines Antrags“ in Szenario 3 mit den Adjektiven „automatisch“ und „manuell“ präzisiert werden, um klarzustellen, wer die jeweilige Aufgabe ausführt. Diese Klarstellung fällt bei Szenario 2 weg, da der Bankmitarbeiter für die Prüfung sämtlicher Anträge verantwortlich ist.

Der wichtigsten Unterschiede zwischen den drei Szenarios in Bezug auf die Systemgrenze sind deutlich erkennbar:

- In Szenario 1 ist der Bankmitarbeiter kein Bestandteil des Systemkontexts, da er kein Akteur des Portals ist; der Mitarbeiter erhält den Antrag per E-Mail.
- In den Szenarios 2 und 3 ist der Bankmitarbeiter Teil des Systemkontexts, da er auf verschiedene Weise mit dem System interagiert.
- In den Szenarios 2 und 3 ist das Banksystem ein Akteur, da zwischen dem Portal und dem Banksystem eine Interaktion möglich sein muss, damit Bankkonten eingerichtet werden können.

Ein Aspekt des Prozesses bleibt in den Diagrammen unerwähnt: Die Benachrichtigung des Kunden im Falle einer Genehmigung oder Ablehnung. Wenn diese Benachrichtigung Teil des Banksystems ist, dann sind die Diagramme korrekt und die Benachrichtigung ist nicht im Systemumfang enthalten. Ist sie jedoch Bestandteil des Antragsportals, dann müssen die Diagramme um die Benachrichtigung erweitert werden.

Vergleicht man die Use-Case-Diagramme und das Kontextdiagramm (siehe Abbildung 2), dann erkennt man die Hauptunterschiede zwischen den beiden Notationen:

- Im Kontextdiagramm zum Szenario 1 ist der Bankmitarbeiter dokumentiert, da er ein Element des Systemkontexts ist, der Informationen aus dem Portal erhält (per E-Mail). Im Use-Case-Diagramm für das Szenario 1 ist der Bankmitarbeiter nicht enthalten, da er in Bezug auf das Portal kein Akteur ist.
- Use-Case-Diagramme erlauben keine Dokumentation von Beziehungen zwischen Akteuren außerhalb des Systems. Akteure können nur dokumentiert werden, wenn sie im Systemkontext enthalten sind. In Kontextdiagrammen ist die Dokumentation des Informationsflusses zwischen Kontextelementen möglich (die Benachrichtigung potenzieller Kunden über die Genehmigung/Ablehnung des Mitarbeiters).
- Mit dem Use-Case-Diagramm wird das System anfänglich funktional zerlegt (in Use Cases). Diese funktionale Zerlegung ist in Kontextdiagrammen nicht ersichtlich.

Diese Unterschiede sind auf die Notationselemente beider Diagramme zurückzuführen und sollten nicht als Vor- oder Nachteile des einen Diagrammtyps gegenüber dem anderen betrachtet werden. Nach Möglichkeit sollte man parallel ein Kontextdiagramm und ein Use-Case-Diagramm erstellen, um von den Stärken beider Diagramme zu profitieren. Muss man sich zwischen Kontext- und Use-Case-Diagramm entscheiden, so ist folgende Faustregel hilfreich: Wenn das betrachtete System in einen komplexen Kontext mit verschiedenen wichtigen Interaktionen außerhalb des Systems eingebettet ist, dann ist ein Kontextdiagramm vorzuziehen. Umfasst das betrachtete System eine Reihe komplizierter Funktionalitäten und Interaktionen mit verschiedenen Benutzern und/oder verbundenen Systemen, dann ist ein Use-Case-Diagramm vorteilhafter.

2.1.5.4 Story-Map

Eine Story-Map [Patton2015] ist eine Technik zum Dokumentieren und Managen der Produktentwicklung während des gesamten Produktentwicklungsprozesses. Ihre Hauptstruktur besteht aus einer zweidimensionalen Anordnung von User-Storys. In der horizontalen Dimension liegt der Schwerpunkt auf dem Backbone, das heißt auf dem narrativen Ablauf des Systems (oder auf dem durch das System bereitgestellten Gesamtprozess). Die vertikale Dimension enthält Details zu den einzelnen Teilen des narrativen Ablaufs sowie eine Trennung von Aspekten gemäß der Entwicklungsreihenfolge der Software.

Deshalb stellt die Story-Map ein nützliches Modell dar, mit dem sich die Funktionalität des Systems verstehen und der Kontext/Systemumfang detailliert beschreiben lässt. In Kapitel 3.4 werden Story Maps detaillierter dargestellt.

2.1.6 Die Unvermeidbarkeit von Änderungen des Systemumfangs

Der initiale Systemumfang (einschließlich Systemkontext) muss zu Beginn einer Entwicklungstätigkeit definiert werden. Ohne klar abgegrenzten Systemumfang hat das Team keinen Rahmen für die Entwicklung. Und ohne Kenntnis des Kontexts versteht das Team nicht, wo das System angesiedelt werden soll und wo es Informationen zum Entwicklungsgegenstand recherchieren kann.

Dennoch sind Systemumfang und Systemkontext niemals endgültig und stabil. Tatsächlich wäre das einzige Ereignis, das einen stabilen Systemkontext und Systemumfang zur Folge hätte, die Außerbetriebnahme des Systems! Es gibt zahlreiche Gründe dafür, den Systemumfang und/oder -kontext anzupassen. Der Kunde bittet vielleicht um Änderungen und benötigt neue Funktionalitäten; oder es sind Änderungen aufgrund einer neuen oder geänderten Gesetzgebung erforderlich.

Der häufigste Grund für Änderungen des Systemumfangs und/oder -kontexts ist jedoch das weiterentwickelte Verständnis der Entwickler und/oder der Stakeholder. Im Allgemeinen stellt jede Entwicklungstätigkeit eine erhebliche Änderung des Systemkontexts dar und diese Änderungen sind nicht vollständig vorhersehbar. Es ist ganz natürlich, dass in solchen Situationen Neues in Erfahrung gebracht wird. Und die neuen Erkenntnisse wirken sich oftmals auf den Systemumfang und/oder -kontext aus.

Diese Situation ist jedoch keine Ausrede, um keine richtige Definition von Systemumfang und Systemkontext erstellen zu müssen. Aus der Perspektive des Requirements Engineering ist das tatsächlich der Hauptgrund, den Systemumfang und -kontext systematisch zu definieren. Ohne gründliche Kenntnis des aktuellen Systemumfangs und -kontexts vor Entwicklungsbeginn ist es nur eine Frage des Zufalls, ob der Bedarf einer späteren Anpassung überhaupt erkannt wird. Die in diesem Kapitel dargestellten Techniken sind schlank und benutzerfreundlich. Die richtige Nutzung dieser Techniken erfordert nur einen geringen Aufwand und sie bietet für jede Entwicklung enorme Vorteile.

2.2 Identifizierung und Management von Stakeholdern

2.2.1 Grundlagen

Gemäß IREB-Glossar ist ein Stakeholder eine Person oder eine Organisation, die einen (direkten oder indirekten) Einfluss auf die Anforderungen eines Systems hat. Unter den indirekten Einfluss fallen außerdem Situationen, in denen eine Person oder eine Organisation durch das System beeinflusst wird.

Diese Definition betont die Bedeutung von Stakeholdern, der richtigen Identifizierung und des Managements von Stakeholdern während der Entwicklung. Die Aussage „das Reagieren auf Veränderungen bewerten wir höher als das Befolgen eines Plans" aus dem Manifest für agile Entwicklungsprozesse wird häufig falsch verstanden und dient als willkommene Ausrede dafür, die gründliche Ermittlung von Stakeholdern zu Beginn einer Entwicklungstätigkeit zu überspringen.

Die Identifizierung eines neuen Stakeholders ist eine unvermeidliche Veränderung, auf die das Team reagieren muss.

Wenn man es versäumt, einen wichtigen Stakeholder zu identifizieren und in die Entwicklung einzubeziehen, kann das erhebliche Auswirkungen haben: Wichtige Anforderungen werden möglicherweise (zu) spät (oder gar nicht) erkannt. Es kann kostspielige Änderungen spät(er) im Prozess nach sich ziehen oder gar zu einem unbrauchbaren System führen. Die Identifizierung und das Management von Stakeholdern ist eine wichtige Investition in die Minimierung des Risikos, dass wichtige Stakeholder und deren Anforderungen fehlen.

2.2.2 Identifizierung von Stakeholdern

In diesem Kapitel stellen wir das Zwiebelschalenmodell („Onion Model“) vor: eine einfache Technik zum Identifizieren und Klassifizieren von Stakeholdern. Zudem behandeln wir die Bedeutung von Benutzern als zentralen Stakeholdern und die Bedeutung, die indirekten Stakeholdern zukommt.

2.2.2.1 Das Zwiebelschalenmodell („Onion Model“) zum Identifizieren und Klassifizieren von Stakeholdern

Mithilfe des Zwiebelschalenmodells von Ian Alexander [Alexander2005] lassen sich Stakeholder einfach ermitteln und klassifizieren. Das Modell umfasst drei Arten von Stakeholdern („Onion Layers“ oder Zwiebelschichten), die systematisch nach Stakeholdern durchsucht werden können:

- Stakeholder des Systems: Diese Stakeholder sind direkt von dem neuen oder geänderten System betroffen. Typische Beispiele für diese Klasse sind Benutzer, Wartungspersonal und Systemadministratoren.
- Stakeholder des umgebenden Kontexts: Diese Stakeholder sind indirekt von dem neuen oder geänderten System betroffen. Typische Beispiele für diese Klasse sind Manager von Benutzern, Project Owner, Sponsoren oder Owner von verbundenen Systemen (beispielsweise Systeme mit einer Schnittstelle zu dem aktuell entwickelten System, siehe Kapitel 2.2.4).
- Stakeholder aus dem weiteren Kontext: Diese Stakeholder haben eine indirekte Beziehung zu dem neuen oder geänderten System oder zum Entwicklungsprojekt. Typische Beispiele für diese Klasse sind Gesetzgeber, Einrichtungen, die Standards festlegen, Wettbewerber, Nichtregierungsorganisationen (NGOs), Gewerkschaften, Umweltschutzbehörden usw.

Stakeholder des Systems bezeichnet man auch als *direkte Stakeholder* („direct stakeholders“). Stakeholder des umgebenden Kontexts und aus dem weiteren Umfeld werden auch als *indirekte Stakeholder* („indirect stakeholders“) bezeichnet.

Das Zwiebelschalenmodell kann in unterschiedlichen Settings zur Identifizierung von Stakeholdern angewendet werden:

- Denkwerkzeug: Verwenden Sie das Zwiebelschalenmodell, um systematisch über das aktuell entwickelte System nachzudenken und notieren Sie alle möglichen Stakeholder, die Ihnen für die einzelnen Schichten in den Sinn kommen.
- Interview-Richtlinie: Nutzen Sie das Zwiebelschalenmodell als Orientierung für kurze Interviews mit Stakeholdern. Bei einem solchen Interview kann der Stakeholder nach potenziellen Stakeholdern in jeder Zwiebelschicht gefragt werden.

- Workshop-Richtlinie: Nutzen Sie das Zwiebelschalenmodell zur Strukturierung eines Workshops für die Identifizierung von Stakeholdern. Das Modell kann als Visualisierungswerkzeug verwendet werden (beispielsweise auf einer Tafel oder einem Flipchart). Gemeinsam mit den Workshop-Teilnehmern können Sie die einzelnen Schichten der Zwiebel analysieren. Jeder Stakeholder schreibt beispielsweise die Namen von Stakeholdern auf eine Karte. Alternativ können die einzelnen Schichten auch im Rahmen einer Brainstorming-Session ausgearbeitet werden.

Als Faustregel gilt, dass bei der Identifizierung von Stakeholdern auf eine große Bandbreite von Quellen zurückgegriffen werden sollte. Ein einziges Interview mit einer Person reicht in der Regel nicht aus, um die wichtigsten Stakeholder zu ermitteln. Der Product Owner sollte stattdessen mehrere Interviews und/oder Workshops für die Identifizierung von Stakeholdern einplanen. Werden bestimmte Namen mehrmals genannt (beispielsweise ist Maria in verschiedenen Geschäftsthemen als sehr fachkundig anerkannt), dann sollte diese Redundanz als wichtiges Zeichen und nicht als verschwendete Zeit interpretiert werden.

2.2.2.2 Die Bedeutung des Benutzers als direkter Stakeholder

Wenn ein System menschliche Benutzer hat, zählen diese zu den wichtigsten direkten Stakeholdern. Der Erfolg eines Systems beruht schließlich auf der Akzeptanz des Systems durch seine Benutzer. Selbst wenn die Features eines Systems perfekt implementiert sind, ist das System wertlos, wenn es die Benutzer nicht verwenden möchten.

Eine einfache Klassifizierung in Bezug auf Stakeholder ist die Trennung zwischen offenen und geschlossenen Umgebungen:

- In einer offenen Umgebung haben die Benutzer Alternativen, aus denen sie auswählen können. Ein Unternehmen möchte beispielsweise eine neue Bürosoftware entwickeln (etwa für die Textverarbeitung und für Präsentationen). Für diese Art von Produkt sind bereits mehrere Alternativen auf dem Markt. Die Stakeholder-Analyse muss sich daher auf Informationen konzentrieren, die Benutzer zum Wechsel von ihrem bestehenden auf das neue System veranlassen können.
- In einer geschlossenen Umgebung sind die Benutzer „gezwungen“, ein neues System zu verwenden. Ein Unternehmen entwickelt zum Beispiel ein neues System für die kaufmännische Verwaltung, um sein Geschäft zu managen, und jeder Mitarbeiter muss das neue System verwenden, da es mit sämtlichen Unternehmensbereichen verbunden ist. In einer solchen geschlossenen Umgebung kommt der Identifizierung von Stakeholdern unter Umständen nicht genügend Aufmerksamkeit zu, da die Benutzer keine andere Wahl haben, als das System zu verwenden. Bei diesem Verhalten wird die Macht des „Immunsystems“ von Unternehmen unterschätzt: Wenn die Benutzer das neue System nicht akzeptieren, dann wird das unternehmerische Immunsystem einen Weg finden, dessen Einführung zu verhindern.

Die Benutzer eines Systems (sowohl in offenen als auch in geschlossenen Umgebungen) decken in der Regel ein weites Spektrum von Personen mit unterschiedlichen Erwartungen, Haltungen und Voraussetzungen ab. Daher ist es ein wichtiger erster Schritt, das Spektrum von Benutzern eines Systems zu verstehen.

Ist die Anzahl von Benutzern gering, dann ist es ratsam, diese (oder deren Vertreter) durch persönliche Interviews kennenzulernen. Dabei kann man den Benutzern direkt anforderungsbezogene Fragen stellen.

Ist die Anzahl von Benutzern groß oder gar unbekannt (dies ist bei offenen Umgebungen die Regel), dann sollte das Spektrum von Benutzern mit anderen Mitteln erfasst werden. In einem solchen Fall ist die Persona-Technik ein geeignetes Werkzeug [Cooper2004]. Eine Persona stellt einen Beispielbenutzer mit klar abgegrenzten Eigenschaften dar. Eine solche Persona wird üblicherweise detailliert beschrieben, erhält einen echten Namen (beispielsweise Jim), ein oder mehrere Bilder, einen kurzen Lebenslauf und eine Liste mit Hobbys. Ziel der Beschreibung ist es, die Persona so realistisch wie möglich zu veranschaulichen und ihr anforderungsbezogene Fragen zu stellen (beispielsweise: Welche Art von Suchfunktion würde Jim bevorzugen?). Für eine Entwicklungstätigkeit ist eine einzige Persona normalerweise nicht ausreichend. Als Faustregel gilt, dass für ein Projekt drei bis fünf Persona mit verschiedenen Hintergründen entwickelt werden sollten. Es empfiehlt sich besonders, Persona mit klar abgegrenzten Positionen zu entwickeln (z. B. einen Berufsanfänger und eine Fachkraft). Wenn für diese ausgewählten, klar umrissenen Benutzerprofile, die für die extremen Nutzungsszenarios des Produkts stehen, neue Software entwickelt wird, dann wird auch das Gros der Benutzer (beispielsweise durchschnittliche oder erfahrene Benutzer) das System akzeptieren.

Die Persona-Technik ist in den Designprozess für neue Software eingebettet. Ein alternativer, stärker messungsorientierter Ansatz ist die Anwendung von Datenanalysen, Google Analytics und Big Data: Das Verhalten von Onlinebenutzern lässt sich häufig direkt analysieren, indem man solche Technologien in bereitgestellte Produktinkremente integriert. Der Hauptvorteil dieser Techniken ist, dass sie konkrete Daten zum Benutzerverhalten bereitstellen. Ihr größter Nachteil besteht darin, dass sie detailliert geplant und in das Software-Inkrement implementiert werden müssen. Deshalb sind bei diesen Techniken klare Messziele entscheidend, da die Erfassung der entsprechenden Daten teuer ist.

2.2.2.3 Die Bedeutung indirekter Stakeholder

Im umgebenden Kontext des Systems gibt es indirekte Stakeholder, die für eine Entwicklung ebenso wichtig sein können, wie die Benutzer selbst. Der Begriff „indirekte Stakeholder“ ist absichtlich sehr weit gefasst, da sich diese Stakeholder für verschiedene Arten von Systemen erheblich unterscheiden. Die generelle Idee hinter indirekten Stakeholdern ist, dass nach Stakeholdern gesucht wird, die sich auf den Erfolg eines Systems auswirken – sei es positiv (unterstützend) oder negativ.

Unterstützung kann auf verschiedene Weise geboten werden. Ein Stakeholder kann wichtige Informationen in Bezug auf die Domäne liefern (zum Beispiel wichtige Geschäftsregeln oder Benutzerbedürfnisse) oder zu künftigen Entwicklungen in der Domäne (etwa eine neue Produktart, ein neues Gesetz, das sich auf das Geschäft auswirkt). Ein Stakeholder kann während der Entwicklung und Einführung des Systems auch politische Unterstützung bieten (beispielsweise ein wichtiger Manager aus einer verbundenen Abteilung).

Nicht zuletzt können auf verschiedene Weise negative Auswirkungen auf den Erfolg eines Systems entstehen. Ein wichtiger Aspekt ist beispielsweise die formale Aufnahme eines Systems in regulierten Umgebungen (z. B. medizinische Systeme): Werden für die Aufnahme eines Systems relevante Stakeholder nicht frühzeitig im Entwicklungsprozess involviert, so erfüllt das neue System unter Umständen wichtige Aufnahmekriterien nicht. Die politische Dimension einer Entwicklungstätigkeit ist ein weiterer Aspekt (beispielsweise kann ein Manager einer Abteilung mit einem konkurrierenden Produkt die Entwicklung behindern). Die negativen Auswirkungen sind nicht auf die Entwicklungstätigkeit beschränkt. Andere Arten von indirekten Stakeholdern, die leicht unterschätzt werden, sind NGOs oder Personen, die nur losen Bezug zu einem System haben. Eine NGO, die beispielsweise im Bereich Datenschutz für persönliche Gesundheit arbeitet, hat eine eindeutige Meinung zur Speicherung bestimmter Arten von persönlichen gesundheitsbezogenen Daten. Wenn Sie ein System in diesem Bereich entwickeln, zieht möglicherweise eine NGO mit einer Kampagne gegen Sie ins Feld.

Die Investition von Zeit in die Identifizierung von indirekten Stakeholdern sollte als ein Mittel betrachtet werden, um zusätzliche Informationen für den Entwicklungsprozess zu sammeln, die dazu dienen, das Risiko des Scheiterns zu reduzieren.

Als Faustregel gilt, dass ein Product Owner, der für das Requirements Engineering verantwortlich ist, einen umfassenden Überblick über die indirekten Stakeholder entwickeln sollte. Das Gespräch mit indirekten Stakeholdern ist oftmals nützlich, selbst wenn ein indirekter Stakeholder keine neuen Erkenntnisse bringt; auch die Bestätigung bereits bekannter Informationen ist häufig hilfreich.

2.2.3 Management von Stakeholdern

Die systematische Identifizierung von zentralen Stakeholdern muss zu Beginn einer Entwicklung als vorbereitende Aktivität stattfinden. Während der Entwicklungstätigkeit müssen diese identifizierten Stakeholder dann kontinuierlich gemanagt werden. Auch wenn dies sehr kostenintensiv erscheinen mag, so genügt in den meisten Kontexten eine einfache Auflistung mit Kontaktdaten und relevanten Attributen (wie Kompetenzbereiche oder Verfügbarkeit). Wird in einem Projekt ein Wiki zum Verwalten der Dokumentation verwendet, so lässt sich die Stakeholder-Liste einfach im Wiki erstellen und verwalten.

Die Liste kann sich jederzeit ändern, entweder weil ein Stakeholder zunächst übersehen wurde oder weil Änderungen im Kontext aufgetreten sind, wie etwa die Etablierung einer neuen Nichtregierungsorganisation. Sobald ein neuer Stakeholder identifiziert wurde, sollte er kontaktiert werden, um die Anforderungen für das neue System zu ermitteln und sonstige nützliche Informationen zu erheben.

Aufgrund der großen Bandbreite von möglichen Stakeholdern sollten sich alle an einer Entwicklung beteiligten Personen (z. B. die Entwicklung und der Product Owner) an der Identifizierung fehlender Stakeholder beteiligen. Der erste Schritt dabei ist, unter den Entwicklern ein Bewusstsein für die Bedeutung von Stakeholdern zu schaffen und nach Zeichen von neuen oder fehlenden Stakeholdern zu suchen.

2.2.4 Andere Quellen für Anforderungen außer Stakeholdern

Abhängig vom System und von der Domäne, können auch die bereits vorhandene Dokumentation, angrenzende Systeme mit Schnittstellen zum entwickelten System, Altsysteme oder sogar Systeme von Konkurrenten eine wichtige Quelle für Anforderungen darstellen. Nachfolgend sind einige Beispiele aufgeführt:

- Wenn es für das aktuell entwickelte System eine Vorgängerversion gibt, können die Dokumentation (sofern vorhanden) und der Quellcode dieses Altsystems wichtige Anforderungen bereitstellen (beispielsweise detaillierte Anforderungen zu Datenstrukturen);
- Hat das aktuell entwickelte System Schnittstellen zu anderen vorhandenen Systemen (beispielsweise im Kontext eines großen Unternehmens), dann bietet die Dokumentation der Schnittstellen wichtige Anforderungen für die Interaktion zwischen dem aktuell entwickelten System und diesen Systemen. Selbstverständlich sind die Benutzer, Entwickler usw. dieser vorhandenen Systeme wichtige Stakeholder;
- Für fast jedes System gibt es ein oder mehrere ähnliche Systeme, das heißt Systeme, die ähnliche Aufgaben in anderen Kontexten ausführen. Solche ähnlichen Systeme werden als Quelle für Anforderungen und Ideen häufig unterschätzt. Wenn Sie zum Beispiel ein Einkaufssystem für ein hoch spezialisiertes Produkt entwickeln, sollten Sie vorhandene Online-Shops und deren Funktionen sichten, um zu prüfen, ob diese auch für Ihre Systeme nützlich sein könnten;

- Bei der Entwicklung eines sehr innovativen Systems könnten auch neuere Forschungen auf dem entsprechenden Gebiet eine wichtige Anforderungsquelle darstellen. Es gibt mehrere Internetdatenbanken, in denen man nach Forschungsergebnissen suchen kann (z. B. Google Scholar).

Wenn Ihre Entwicklungstätigkeit von zusätzlichen Quellen für Anforderungen profitieren kann, sollten diese systematisch identifiziert und ähnlich wie Stakeholder gemanagt werden. Detaillierte Informationen zum Management anderer Anforderungsquellen werden im Modul IREB Advanced Level Elicitation bereitgestellt.

2.3 Zusammenfassung

Die Definitionen von Visionen und Zielen, Stakeholdern, Systemumfang und -kontext sind voneinander abhängig:

- Die relevanten Stakeholder formulieren die Vision und die Ziele. Aus diesem Grund kann sich die Identifizierung eines neuen Stakeholders auf die Vision oder die Ziele auswirken.
- Mit den Visionen und Zielen lässt sich die Identifizierung von neuen Stakeholdern lenken, indem man die Frage stellt: Welcher Stakeholder könnte am Erreichen der Vision und/oder der Ziele Interesse haben oder ist vom Erreichen der Vision und/oder der Ziele betroffen?
- Visionen und Ziele können zum Definieren eines initialen Systemumfangs verwendet werden. Dazu muss man folgende Frage beantworten: Welche Elemente sind zum Erreichen der Vision und/oder der Ziele nötig?
- Eine Änderung der Systemgrenze (und dadurch des Systemumfangs) kann sich auf die Vision und/oder die Ziele auswirken. Werden Aspekte aus dem Systemumfang entfernt, verfügt das System möglicherweise nicht mehr über ausreichende Mittel zum Erreichen der Vision und/oder der Ziele. Umgekehrt können bei einer Erweiterung des Systemumfangs neue Mittel zur Erfüllung der Vision/und oder der Ziele hinzukommen.
- Stakeholder schlagen den Systemumfang vor. Deshalb kann sich die Identifizierung eines neuen relevanten Stakeholders auf den Systemumfang auswirken. Ein wichtiger Manager entscheidet beispielsweise, dass der Projektumfang erweitert werden kann.
- Für Änderungen am Systemumfang (beispielsweise die Erfüllung eines neuen oder geänderten Ziels) ist die Zustimmung der relevanten Stakeholder erforderlich.

Diese starken wechselseitigen Abhängigkeiten machen es erforderlich, alle drei Elemente im Gleichgewicht zu halten und die Auswirkungen der Veränderung eines der drei Elemente auf die anderen zu untersuchen. Ein Bewusstsein für diese wechselseitigen Abhängigkeiten ist der erste Schritt, um gemeinsam an den Visionen und Zielen, an der Ermittlung von Stakeholdern und am Systemumfang zu arbeiten. Aufgrund dieser engen Verflechtungen empfehlen wir, diese Elemente zusammen anzugehen.

Bevor Sie mit der iterativen Entwicklung beginnen, empfehlen wir, dass Sie eine kohärente, initiale Spezifikation mit folgenden Inhalten erstellen:

- Vision und/oder Ziele
- Systemumfang und Systemkontext
- Anfängliche Liste mit Stakeholdern (und möglicherweise mit anderen Quellen)

Die in diesem Kapitel dargestellten Methoden und Werkzeuge können zur Erstellung einer schlanken Spezifikation verwendet werden. Ein guter, schlanker Ausgangspunkt ist ein halbtägiger Workshop, in dem alle drei Elemente auf der Agenda stehen. Als Vorbereitung auf den Workshop sollte jeder Teilnehmer die folgenden Fragen beantworten:

- Was ist Ihre Vision für das System? Welche Ziele sind für Sie am wichtigsten?
- Wie verstehen Sie den Systemkontext und den Systemumfang?
- Welche Stakeholder und anderen Quellen (Dokumente, Systeme) müssen für das Projekt berücksichtigt werden?

Wenn die Workshop-Teilnehmer nicht mit der Terminologie vertraut sind, stellen Sie ihnen die Definitionen als Hintergrundinformationen zur Verfügung. Das Ergebnis dieses Workshops ist der Ausgangspunkt für die Erstellung einer anfänglichen Spezifikation mithilfe der in diesem Kapitel vorgestellten Methoden und/oder Techniken.

Die anfängliche Spezifikation sollte als „lebendes“ Dokument betrachtet werden, das regelmäßig zu prüfen und zu aktualisieren ist. Die Gepflogenheiten und Techniken der agilen Entwicklung bieten verschiedene Möglichkeiten für eine schlanke Pflege dieser Dokumentation. Die Einbeziehung einer Gegenprüfung auf Basis der Dokumentation von Kontext/Systemumfang in die Definition of Ready stellt einen pragmatischen Ansatz dar. Wenn der Systemumfang beispielsweise anhand eines Use-Case-Diagramms beschrieben wird, würde jede User-Story mit einem Use Case und einem Akteur verknüpft werden.

2.4 Fallstudie und Übungen

Wir greifen im gesamten Handbuch auf eine Fallstudie zurück. Stellen Sie sich vor, Sie erstellen ein Lernsystem. Das Lernsystem soll Studierende dabei unterstützen, Requirements Engineering zu erlernen. Um zu beurteilen, ob ein Student die verschiedenen Themen verstanden hat, sollten kurze Videosequenzen mit Fragen angeboten werden. Die Plattform sollte auf allen Geräten nutzbar sein, über die sich die Studenten mit dem Internet verbinden können, das heißt Smartphones, Tablets, Laptops usw. Für den Leiter einer größeren Gruppe sollte die Plattform Informationen zum Fortschritt der einzelnen Studenten bieten. Als Namen für die Plattform schlagen wir „iLearnRE“ vor.

Übungsvorschläge:

Wenn Sie den erfolgreichen Projektstart üben möchten, können Sie dazu gerne die Fallstudie „iLearnRE“ nutzen. Als anfängliche Liste für die Vision und/oder Ziele haben wir folgende Aussagen definiert:

- Online-Videoschulungsportal, das Informationen zum Requirements Engineering enthält und das zur Vorbereitung auf die IREB Prüfung dient
- Verfügbarkeit auf verschiedenen Plattformen, auch für Internetverbindungen mit niedriger Bandbreite
- Umfasst einen Chatroom/ein Diskussionsforum, über die Probleme mit anderen Studenten diskutiert werden können
- Ein Management-Dashboard zur Kontrolle des Fortschritts von Studenten in Ihrem Team

Weiterhin haben wir die folgende Liste von Benutzern definiert:

- Studenten
- Administrator des Portals
- Teamleiter (von Studenten)
- Verfasser von Fragen

Mit diesen Informationen können Sie an den folgenden Übungen arbeiten:

1) Verwenden Sie die Techniken aus Absatz 2.1.2, um die Visions-/Zielaussagen umzuformulieren
2) Erstellen Sie ein Kontextdiagramm für iLearnRE.
3) Erstellen Sie ein Use-Case-Diagramm für iLearnRE.
4) Denken Sie über zusätzliche Stakeholder für iLearnRE nach

3. Umgang mit funktionalen Anforderungen

In Kapitel 2 haben Sie Informationen zum erfolgreichen Projektstart erhalten, beispielsweise über wichtige Voraussetzungen, die Sie erfüllen sollten, bevor Sie mit einer iterativen, inkrementellen Systementwicklung beginnen.

In diesem Kapitel geht es um die Ermittlung, Diskussion und Erfassung von funktionalen Anforderungen. Die beiden anderen Kategorien von Anforderungen (Qualitätsanforderungen und Randbedingungen) werden in Kapitel 4 behandelt. Viele der Ideen in diesem Kapitel sind auch für die beiden anderen Kategorien relevant.

In diesem Kapitel erfahren Sie, dass es ganz normal ist, dass Stakeholder sich ständig auf unterschiedlichen Granularitätsstufen unterhalten. Manchmal werden sie Sie um sehr abstrakte Dinge bitten, für die Sie als Product Owner unter Umständen mühevoll recherchieren müssen, um alle relevanten Details herauszufinden. Und ein anderes Mal werden Sie um sehr kleine, präzise Aspekte bitten, die bereits ganz nah an das heranreichen, was die Entwickler verstehen und implementieren können. Ihre Aufgabe als Product Owner ist es, mit all diesen Granularitätsstufen umzugehen. Ein hohes Level an Abstraktion ist nicht schlecht, wenn die Features noch nicht sehr bald benötigt werden. Für Features, die in einer der nächsten Iterationen implementiert werden müssen, ist mehr Präzision erforderlich.

Im agilen Umfeld werden grobgranulare Anforderungen häufig als Epics, Themen oder Features bezeichnet. In diesem Kapitel wird veranschaulicht, wie Sie diese in INVEST-User-Storys umwandeln können, sodass sie ausreichend präzise sind, um von den Entwicklern bearbeitet zu werden.

Wenn Sie einmal die Idee akzeptiert haben, dass Anforderungen auf verschiedenen Granularitätsstufen existieren, wirft das in der Regel einige Fragen auf:

- Wie gehen wir mit verschiedenen Granularitätsstufen um?
- Welche Kriterien können und sollten angewendet werden, um große, abstrakte Topics in kleinere Blöcke zu untergliedern?
- Ist es manchmal erforderlich, viele kleine Anforderungen in größere Blöcke zusammenzufassen, sodass wir einen „größeren Zusammenhang“ zur besseren Orientierung haben?
- Wie präzise müssen wir sein, bevor die Entwickler mit der Implementierung beginnen können?
- Ist es notwendig oder ratsam, verschiedene Anforderungsniveaus beizubehalten oder können wir abstrakte Aussagen verwerfen, sobald wir konkretere Anforderungen haben?
- Müssen wir all das schriftlich festhalten oder genügt es, wenn wir einfach darüber sprechen?

Auf all diese Fragen gehen wir in diesem Kapitel ein. Wie bereits erwähnt, konzentrieren wir uns dabei auf funktionale Anforderungen. In Kapitel 4 diskutieren wir Qualitätsanforderungen und Randbedingungen. In Kapitel 5 werden die Schätzung, Sortierung und Priorisierung von Anforderungen diskutiert. In diesem Kapitel 3 geht es rein um das Management komplexer funktionaler Anforderungen und deren Verfeinerung auf ein Niveau, mit dem die Entwickler arbeiten können.

3.1 Unterschiedliche Stufen der Anforderungsgranularität

Nehmen wir einige Beispiele aus unserer Fallstudie „iLearnRE“. Eines unserer Ziele können wir wie folgt formulieren: „Als Student möchte ich im Rahmen eines Online-Videokurses mehr über Requirements Engineering erfahren, damit ich keine Präsenzschulung besuchen muss.“

Nehmen wir an, dass einer Ihrer Stakeholder nun um folgendes Feature bittet:

> „Als Abteilungsleiter möchte ich in der Lage sein, den Lernfortschritt all meiner Mitarbeiter zu prüfen."

Das ist keine sehr präzise Aussage, da wir nicht notwendigerweise verstehen, was mit „Fortschritt" gemeint ist. Außerdem wissen wir nicht, wie das Ergebnis einer solchen Prüfung aussehen sollte. Es handelt sich hierbei jedoch um eine relevante Anforderung. Wir würden dies als eine grobgranulare Anforderung charakterisieren.

Angenommen, einer der Studierenden präsentiert folgende Anforderung:

> „Beim Abspielen eines Videoclips möchte ich die Möglichkeit haben, die restliche Laufzeit in Sekunden zu sehen."

Das ist eine präzisere Anforderung, für deren Implementierung aber noch einige Details nötig sind (Position, Größe, Farbe der Laufzeitanzeige). Diese Details kann der Product Owner hinzufügen, was zu einer Lösung durch den Product Owner führen wird, aber nicht unbedingt die beste Lösung ist. Alternativ bittet der Product Owner das Team während des Refinement Meetings um Optionen in Bezug auf die Details und trifft auf Basis der verfügbaren Informationen eine Entscheidung.

Stakeholder sprechen ständig in unterschiedlichen Granularitätsstufen mit uns. Als Product Owner können und sollten Sie sie nicht dazu zwingen, strukturierter zu sein. Es gehört zu Ihren Aufgaben als Product Owner, gemeinsam mit den am Requirements-Engineering-Prozess Beteiligten mit diesen verschiedenen Anforderungsstufen zu arbeiten und sie zu strukturieren.

Wie in Abbildung 4 ersichtlich, weist jedes System Anforderungen auf verschiedenen Granularitätsstufen auf, die sich unter der obersten Ebene der Vision und/oder der Ziele befinden. Sie als Product Owner streben zwei Ziele an:

1. Sie möchten den Überblick über sämtliche derzeit bekannten funktionalen Anforderungen haben. Dies ermöglicht die Auswahl der wertvollsten Anforderungen für eine frühzeitige Implementierung, um den größeren Zusammenhang nicht aus den Augen zu verlieren usw.;
2. Das Verstehen von Anforderungen in ausreichender Genauigkeit, damit sie von den Entwicklern für die Implementierung übernommen werden können.

Bei einigen Methoden erhalten die Anforderungsstufen spezielle Bezeichnungen. Bei SAFe beispielsweise werden die großen Blöcke als „Epics" bezeichnet, mittelgroße Anforderungen als „Features" [3] und Anforderungen auf einer niedrigeren Stufe als „User-Storys". Andere gängige Bezeichnungen für abstraktere Anforderungen sind „Topics" oder „Themen".

In der Agile-Community gibt es keinen Konsens über die Terminologie für abstraktere Anforderungen. Wir behandeln diese Begriffe in den Kapiteln 3.2 und 3.6.

Im Prozess der Anforderungsermittlung und -dokumentation kann diese *Granularitätshierarchie* auf verschiedene Weise erstellt werden. Wie zuvor angemerkt, nennen die Stakeholder üblicherweise ihre Wünsche auf unterschiedlichen Stufen. Sie können also versuchen, „top-down" zu arbeiten (von Visionen und/oder Zielen ausgehend zu Anforderungen einer niedrigeren Stufe), „bottom-up" (Gruppierung von Anforderungen mit hohem Detaillierungsgrad in größere Blöcke) oder „middle-out" (beginnend mit Anforderungen der mittleren Stufe, wobei einige detaillierter unterteilt und andere zusammengefasst werden).

[3] SAFe hat auch die optionale Stufe „Capabilities" zwischen Epics und Features.

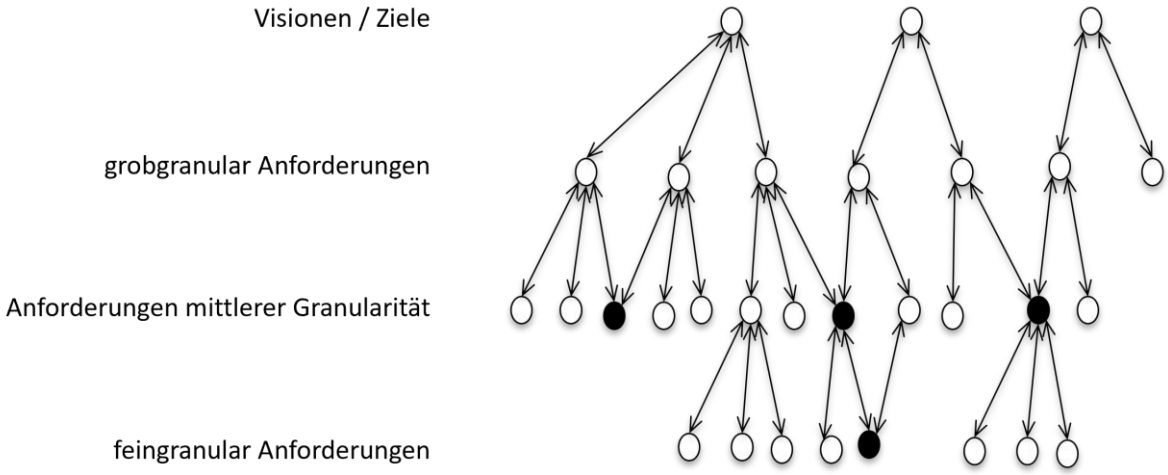

Abbildung 4: Anforderungen auf verschiedenen Granularitätsstufen

Als Product Owner sollten Sie die Beziehungen (Verfolgbarkeit oder Verknüpfungen) zwischen allen Anforderungen pflegen. Dadurch erhalten Sie nicht nur einen besseren Überblick, sondern können auch nicht zielorientierte Anforderungen verwerfen. So können Sie vermeiden, dass Anforderungen sich schleichend entwickeln und sich auf diejenigen konzentrieren, die unbedingt erreicht werden sollen.

Beachten Sie, dass manche detaillierten Anforderungen Bestandteil von mehreren Anforderungen auf höherer Stufe sein können, wie in Abbildung 4 durch die schwarzen Punkte dargestellt. Eine detaillierte Aktivität kann beispielsweise als Teil von zwei oder mehr Geschäftsprozessen durchgeführt werden.

Abbildung 5 zeigt einige Anforderungsbeispiele aus der Fallstudie iLearnRE, einschließlich deren Verknüpfungen.

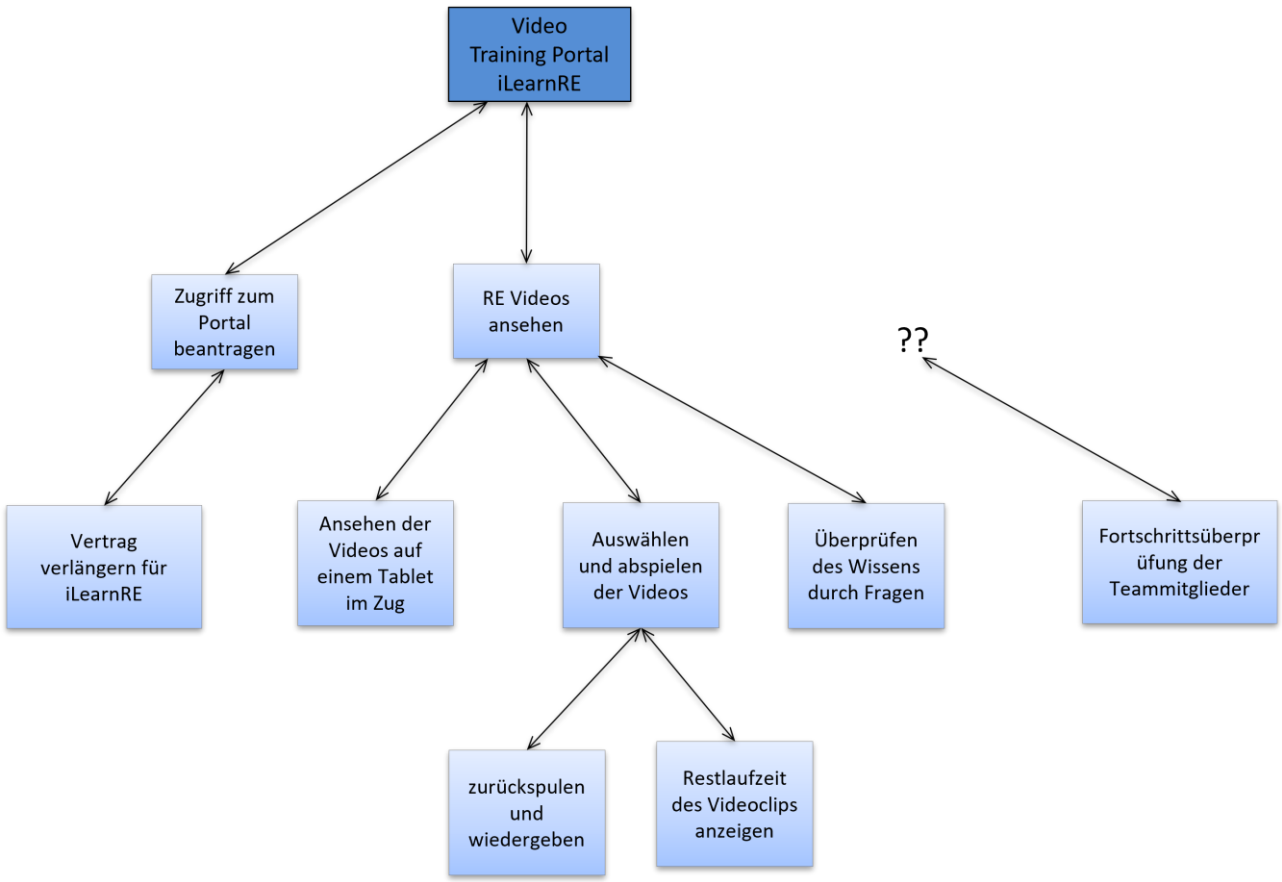

Abbildung 5: Anforderungsbeispiele aus der Fallstudie

Dank einer solchen strukturierten Hierarchie von Anforderungen, läuft der Product Owner (und alle anderen Stakeholder) in einem größeren Projekt nicht so leicht Gefahr, den Überblick zu verlieren. Die Stufen in dieser Hierarchie können genutzt werden, um Schätzungen vorzunehmen und Anforderungen zu priorisieren. Dies wird in Kapitel 5 detaillierter behandelt.

Außerdem besprechen wir in den nächsten Kapiteln Kriterien für die Zusammenfassung oder Aufteilung von Anforderungen, nützliche Notationen zu deren Erfassung sowie Werkzeuge und Techniken, die es erleichtern, den Überblick zu behalten.

3.2 Kommunizieren und Dokumentieren auf unterschiedlichen Stufen

Um funktionale Anforderungen zu erhalten, die an die Entwickler kommuniziert und von diesen implementiert werden können, ist eine präzisere Formulierung der Vision und/oder der Ziele nötig.

Auf Basis des Prinzips „teile und herrsche“ müssen wir ein großes System oder Produkt in kleinere Teile zerlegen. Abbildung 6 veranschaulicht diese Herangehensweise. Wir behandeln entsprechende Strategien und Taktiken dazu, wie wir dieses Ziel erreichen können.

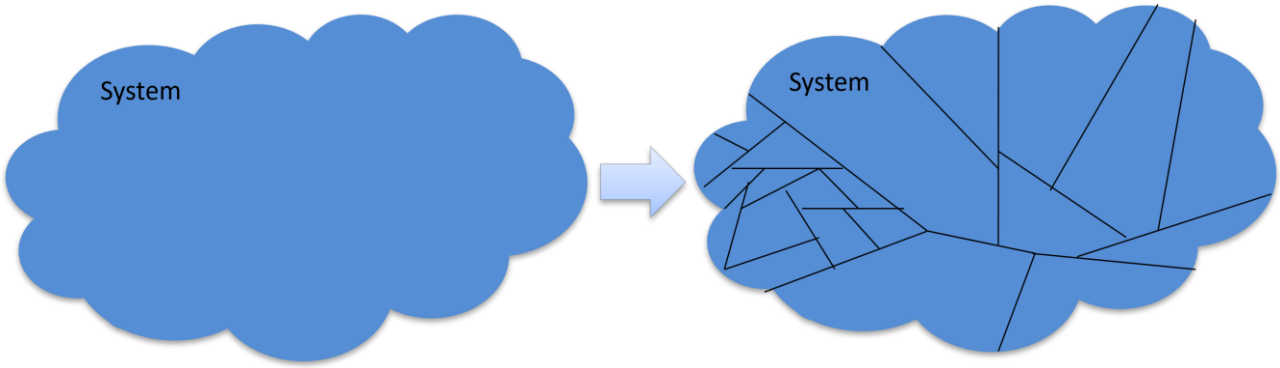

Abbildung 6: Zerlegen funktionaler Anforderungen

Hier sind einige Ansätze für das Zerlegen eines großen Systems beschrieben (einschließlich Beispielen aus der Fallstudie iLearnRE):

1. Aufteilung in logische Funktionen (auch als Features, Epics oder Themen bezeichnet):

   Zum Beispiel: Erstellen eines Vertrages für E-Learning, das Ansehen von Videos, Wissenstests mit Fragen oder das Überprüfen des Lernfortschritts

2. Verwendung der Historie, z. B. die Struktur eines vorhandenen Produkts, als Aufteilungsthema:

   Da es kein Vorgängerprojekt zu unserer Fallstudie gibt, funktioniert diese Strategie hier nicht.

3. Aufteilung nach organisatorischen Aspekten (d. h. Teile, die verschiedenen Abteilungen oder Benutzergruppen nutzen):

   Zum Beispiel: Software für Studenten, Software, die den Teamleiter unterstützt, Software für die Administratoren des Produkts iLearnRE

4. Aufteilung nach Hardware:

   Zum Beispiel: ein iLearnRE-Desktop mit responsivem Design, eine native iPhone-App, eine native Android-App

5. Aufteilung nach geografischer Verteilung:

   Zum Beispiel: iLearnRE für das Land mit der höchsten Anzahl von potenziellen Benutzern, Erweiterung auf andere Länder mit unterschiedlicher Gesetzgebung.

6. Aufteilung nach Daten (Geschäftsobjekten):

   Zum Beispiel: Funktionen für Videos oder Funktionen zu Fragen, Verträgen und Rechnungen

7. Aufteilung in extern angestoßene, wertschaffende Prozesse.

All diese Ansätze resultieren in kleineren Blöcken, die dann separat analysiert werden können.

Bei den ersten sechs Herangehensweisen stehen die *internen* Strukturen des Systems im Mittelpunkt: Funktionen, historische Struktur, organisatorische Aufteilung, Hardware, geografische Verteilung oder dessen Geschäftsobjekte.

Nur der letzte Ansatz (wertschaffende Prozesse) beginnt in einem Kontext außerhalb unseres Systemumfangs. Dabei werden externe Auslöser betrachtet, auf die unser System reagieren sollte.

Diese Auslöser können verschiedene Quellen haben: menschliche Benutzer, die etwas vom System brauchen, andere Softwaresysteme, die Daten senden und eine Systemaktion anfordern, Hardwaregeräte (wie Sensoren), die eine Aktion in unserem System auslösen.

Das Kontextdiagramm ist eine wertvolle Quelle beim Identifizieren von externen Auslösern, da es alle angrenzenden Systeme aufzeigt, die irgendetwas von dem betrachteten System anfordern könnten.

Diese wertorientierte Aufteilung wurde in den vergangenen Jahrzehnten von zahlreichen anderen Autoren vorgeschlagen: [McPa1988] bezeichnete sie als „ereignisorientierte Zerlegung“ („event-oriented decomposition“), [Jacobson1992] verwendete dafür „Use-Case-Zerlegung“ („use case decomposition“), [HaCh1998] prägte dafür den Begriff „Geschäftsprozesse“ und [Cohn2010] schließlich „User-Storys“. Sie alle vermitteln unterschiedliche Notationen zum Erfassen der Ergebnisse dieser Zerlegung. Abbildung 7 zeigt eine solche Zerlegung in zwei dieser Notationen: Use Cases und User-Storys.

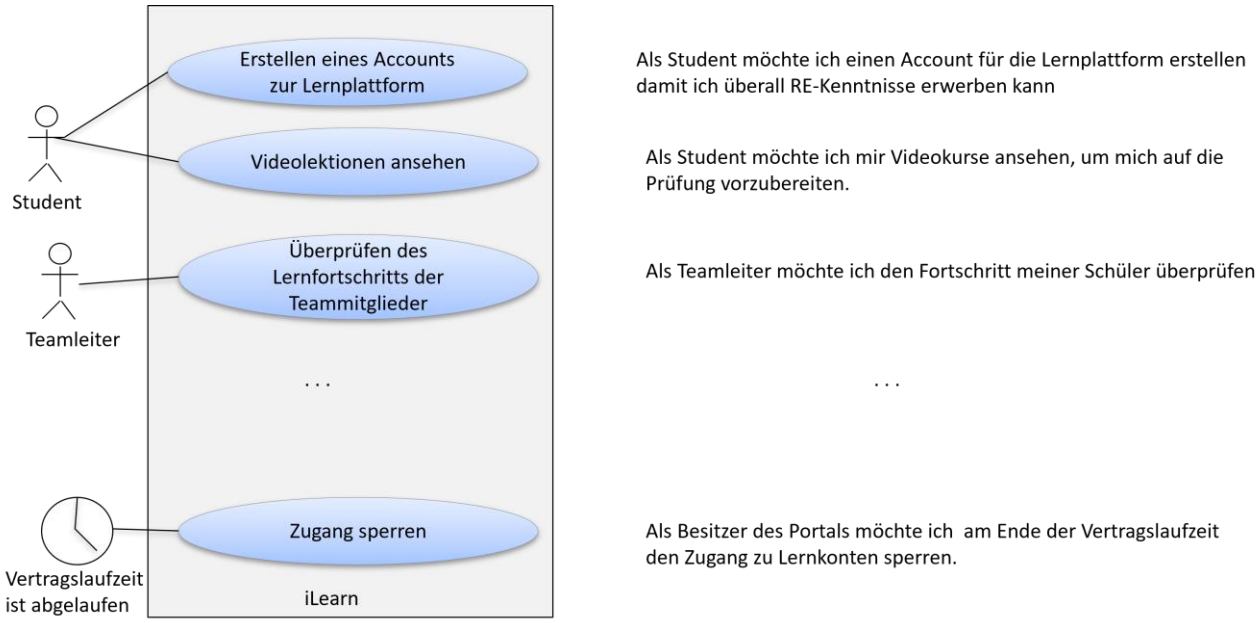

Abbildung 7: Wertorientierte Systemzerlegung in Prozesse mit verschiedenen Notationen

Lassen wir die Notationen für einen Moment außer Acht und sehen uns die Merkmale dieser Zerlegungen an. Agilitätsexperten erkennen in diesen Kriterien die ersten drei INVEST-Kriterien von Bill Wake [Wake2003] wieder.

Alle resultierenden Prozesse sind:

I: Independent [4], also unabhängig voneinander. Das bedeutet, sie sind autonom und minimieren wechselseitige Abhängigkeiten. Ihre Konzepte sollten sich nicht überlappen und wir wären gerne in der Lage, sie in beliebiger Reihenfolge einzuplanen und zu implementieren.

N: Negotiable, also verhandelbar, das heißt, sie stellen noch keinen festen Vertrag dar, lassen jedoch Raum für die Diskussion der Details.

V: Valuable, das heißt wertvoll: Sie schaffen für den Anforderer, also eine Person oder ein anderes System in dem Kontext, einen echten Mehrwert.

Die anderen Kriterien von INVEST werden im nächsten Kapitel über User-Storys behandelt.

Die zuvor für die Zerlegung genannten Ansätze können ebenfalls verwendet werden. Vor allem beim Verfassen von Anforderungen für ein bestehendes System stellt die aktuelle Komponenten- oder Subsystemstruktur oftmals einen guten Ausgangspunkt für die Ermittlung neuer Anforderungen dar. Es besteht jedoch die Gefahr von Lücken oder Überschneidungen dieser Teile in der Spezifikation (siehe Abbildung 8). Da sämtliche Backlog-Einträge diskutiert und verhandelt werden, würden Sie die Lücken und Überschneidungen wahrscheinlich bemerken. Durch das von Mehrwert schaffenden Prozessen ausgehende Denken (ganz gleich, mit welcher Notation) umgeht man diese Gefahren jedoch von Anfang an.

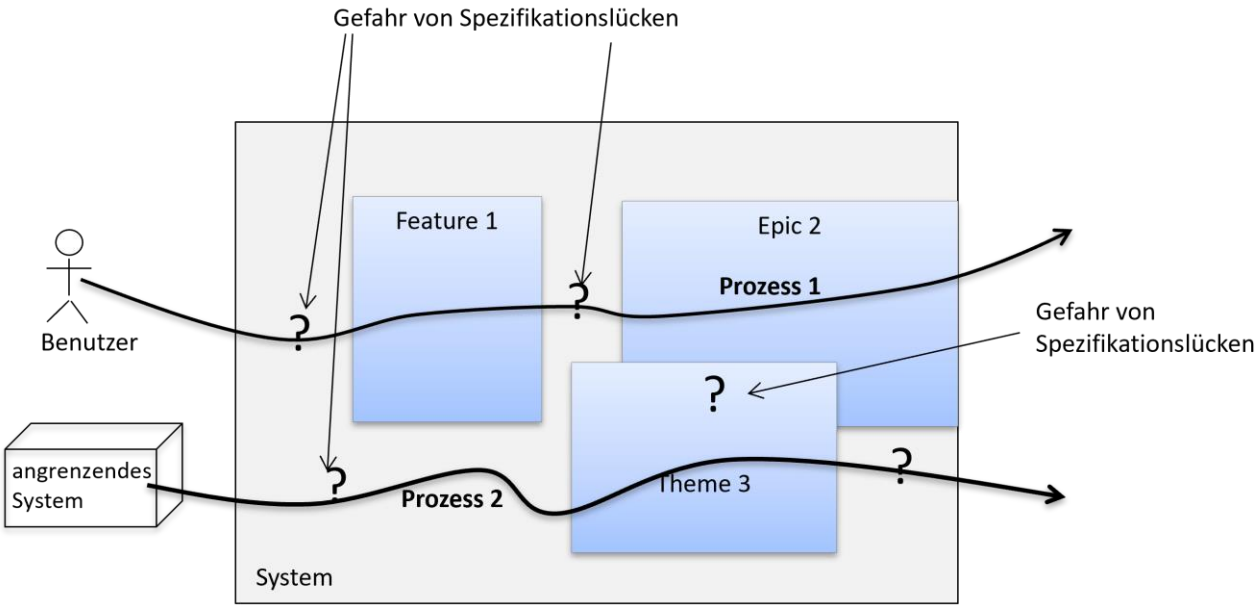

Abbildung 8: Lücken und Überschneidungen in Spezifikationen

Um zu einer guten wertorientierten Prozesszerlegung zu gelangen, empfiehlt es sich, nicht in Bezug auf Benutzer oder Akteure des Systems zu denken, sondern Ereignisse zu identifizieren, die in dem Kontext geschehen und auf die das System reagieren muss. [McPa1988] identifizierte zwei grundlegende Arten von Ereignissen:

- Externe Ereignisse: ausgelöst durch Benutzer oder angrenzende Systeme;
- Zeitliche Ereignisse: ausgelöst durch einen Zeitablauf oder die Beobachtung systeminterner Ressourcen.

[4] Eine andere Interpretation des Buchstabens „I" ist „Immediately actionable" [S@S Guide]

Als Product Owner fehlt Ihnen vielleicht die zweite Kategorie, da es keinen expliziten Akteur oder Benutzer gibt. Das System führt ohne externe Auslöserdaten einen vordefinierten Prozess aus, der lediglich durch einen Zeitablauf oder die Beobachtung systeminterner Ressourcen ausgelöst wird.

Beispiele für beide Arten aus unserer Fallstudie:

- Externes Ereignis: „Als Student möchte ich mein Wissen anhand von Testfragen beurteilen."
- Zeitliches Ereignis: „Zwei Wochen vor Ablauf des *Einschreibungszeitraumes* müssen die Studenten an eine mögliche Verlängerung erinnert werden."

Wir haben jetzt mehrere Herangehensweisen kennengelernt, mit deren Hilfe wir funktionale Anforderungen zur Erfüllung unserer Visionen und Ziele ermitteln können. Der Vorschlag besteht darin, eine prozessorientierte Zerlegungsstrategie anzuwenden, da dies die Identifizierung von Funktionsblöcken erleichtert, die **I**ndependent (unabhängig), **N**egotiable (verhandelbar) und **V**aluable (wertvoll) sind. Jede andere Zerlegungsstrategie, die zu solchen INV-Blöcken führt, ist ebenfalls in Ordnung.

Als Product Owner möchten Sie einen Überblick über die Funktionalitäten des Systems erhalten. Sie können Ihrem Backlog selbstverständlich jederzeit weitere Funktionalitäten hinzufügen; der Überblick wird Ihnen jedoch für Entscheidungen zur Projekt-Roadmap, grobe Schätzungen oder für die Diskussion von Minimum Viable Products (MVP) oder Minimum Marketable Products (MMP) nützlich sein. Das Backlog ist eine gute Basis, auf der Sie entscheiden können, was man sich frühzeitig detaillierter ansehen muss.

Nun, da wir verschiedene Möglichkeiten besprochen haben, um eine grobe Zerlegung zu erreichen, konzentrieren wir uns auf das Kommunizieren und Dokumentieren der funktionalen Anforderungen.

Die grundlegende Entscheidung liegt zwischen Zeichnen und Schreiben. Sie können eine Zerlegung Ihrer Ziele oder Visionen auf Stufe 1 visualisieren, indem Sie entweder ein Use-Case-Diagramm zeichnen oder indem Sie größere User-Storys auf separaten Karten erfassen. Abbildung 7 zeigte Auszüge aus unserer Fallstudie in beiden Varianten nebeneinander. Im folgenden Kapitel werden User-Storys detaillierter behandelt.

Beachten Sie, dass beide Notationen im Prinzip denselben Informationsumfang aufweisen und gleich detailliert oder abstrakt sind. Es ist mehr oder weniger eine Frage des persönlichen Geschmacks, ob Sie Überblicksbilder oder Storys in Textform bevorzugen.

3.3 Arbeiten mit User-Storys

Für einen Product Owner sind User-Storys eine hervorragende Möglichkeit, Anforderungen an alle Stakeholder und auch an die Entwickler zu kommunizieren. User-Storys werden gewöhnlich auf Story-Cards erfasst. Es gibt aber auch eine Vielzahl von Werkzeugen für die elektronische Erfassung. In diesem Kapitel konzentrieren wir uns auf die Idee der User-Storys.

3.3.1 Das 3-C-Modell

Wie zuvor erwähnt, werden Storys häufig auf Karteikarten oder Haftnotizzetteln erfasst und zur Erleichterung von Planungen und Diskussionen an Wänden oder in Tabellen angeordnet. Dies führt zu einem erheblichen Fokuswechsel vom Schreiben über Features zur Diskussion darüber. Diese Diskussionen können tatsächlich wichtiger sein, als der auf einer Karte oder einem Haftnotizzettel erfasste, eigentliche Text.

Ron Jeffries [Jeffries2001] fasste diesen Aspekt in seinem 3-C-Modell (Card, Conversation, Confirmation – Karte, Diskussion, Bestätigung) zusammen, um den eher sozialen Charakter von Storys von dem eher dokumentarischen Charakter anderer Anforderungsnotationen abzugrenzen. In den folgenden Kapiteln werden seine Ideen erläutert:

Die „**Karte**“ (Karteikarte oder Haftnotizzettel) ist ein physischer Gegenstand, der den Inhalten, die sonst nur eine Abstraktion wären, eine greifbare und beständige Form verleiht. Die Karte enthält nicht alle Informationen, aus der eine Anforderung besteht. Stattdessen ist darauf gerade ausreichend Platz für Text, um die Anforderung zu erkennen und jeden daran zu erinnern, was die Story ist. Die Karte ist somit nur ein Hinweis auf eine bestehende Anforderung. Sie wird bei der Planung verwendet. Darauf werden Notizen erfasst, um beispielsweise die Priorität und Kosten darzustellen. Sie wird häufig an die Programmierer übergeben, wenn die Story zur Implementierung eingeplant wird, und sie wird an den Kunden zurückgegeben, wenn die Story abgeschlossen ist.

Die „**Diskussion**“ findet zu verschiedenen Zeitpunkten und an unterschiedlichen Orten während eines Projekts statt, insbesondere, wenn die Story geschätzt wird (in der Regel während der Release-Planung) und noch einmal beim Planungsmeeting für die Iteration, wenn die Story für die Implementierung eingeplant wird. Dabei werden verschiedene Personen einbezogen, die sich mit einem bestimmten Feature eines Softwareprodukts befassen: Kunden, Benutzer, Entwickler, Tester – und es handelt sich größtenteils um einen verbalen Austausch von Gedanken, Meinungen und Gefühlen.

Bei der Diskussion können andere Anforderungen, Artefakte und Dokumentation ergänzend hinzugezogen werden. Eine gute Ergänzung bilden Beispiele; die besten Beispiele sind ausführbare Testfälle.

Die „**Bestätigung“:** Ganz gleich, wie viel wir diskutieren oder wie viel Dokumentation wir produzieren, wir können nicht sicher sein, ob es von allen richtig verstanden wurde. Das dritte C der zentralen Aspekte einer User-Story liefert die Bestätigung, die wir brauchen: den Akzeptanztest/Abnahmetest.

Die Bestätigung durch den Abnahmetest erlaubt es uns, den einfachen Ansatz mit Karten und Diskussionen zu nutzen. Wenn in der Diskussion über eine Karte die Details des Abnahmetests zur Sprache kommen, stimmen der Product Owner und die Entwickler die noch offenen abschließenden Details ab. Indem das Implementierungsteam am Ende der Iteration zeigt, dass die Abnahmetests laufen, erfährt der Product Owner, dass das Team die benötigten Anforderungen liefern kann und wird.

3.3.2 Eine Vorlage für User-Storys:

Mike Cohn definiert User-Storys auf folgende Weise:
(https://www.mountaingoatsoftware.com/agile/user-stories):

„User-Storys“ sind kurze, einfache Beschreibungen eines Features. Diese werden aus der Perspektive der Person dargestellt, die die neue Funktion wünscht, also gewöhnlich ein Benutzer oder Kunde des Systems. Die Beschreibung erfolgt in der Regel nach einer einfachen Schablone:

Als <Art des Benutzers> möchte ich <ein Ziel>, um <ein Grund/Nutzen>.“

Beachten Sie die drei Komponenten dieser Schablone. Sie stellen sicher, dass:

1. wir **jemanden** haben, der diese Funktionalität haben möchte („Als Anwender …“),
2. wir wissen, **was** der Anwender möchte („… möchte ich …“) und
3. wir verstehen **warum**, das heißt, den Grund oder die Motivation dahinter („… um …“).

Diese Formel hilft uns, darüber nachzudenken, wer was und weshalb haben möchte. Es ist jedoch weniger der Formalismus, der User-Storys erfolgreich macht, sondern das Stellen und Beantworten dieser drei Fragen.

In der Abbildung 7 haben Sie einige Beispiele für Storys aus unserer Fallstudie iLearnRE gesehen. Hier sind einige zusätzliche Beispiele:

- Als Student möchte ich Fragen in einem Forum stellen können, um Antworten oder Meinungen von anderen zu erhalten.

- Als Verfasser von Fragen möchte ich dem Pool neue Fragen und Antworten hinzufügen, damit die Studenten ihr Wissen testen können.
- Als Manager des Portals möchte ich neue Versionen von offiziellen IREB Fragen hochladen, sodass unser Portal immer auf dem aktuellen Stand von IREB ist.

Mike Cohn erklärte in seiner Definition, dass User-Storys aus der Perspektive der Person dargestellt werden, die die neue Funktion wünscht. Beachten Sie, dass der Begriff „Benutzer" manchmal etwas irreführend sein kann, da die Person, die ein Feature will, nicht notwendigerweise dieselbe Person ist, die mit dem System als Benutzer arbeitet.

Betrachten wir dazu das letzte Beispiel: Der Manager, der neue Versionen von offiziellen IREB Fragen hochladen muss, ist nicht zwingend derjenige, der diese Aufgabe erledigt haben möchte. Der Business Owner möchte, dass dies erledigt wird.

Das gilt vor allem auch für zeitgesteuerte Prozesse, wenn also ein Prozess zu einem bestimmten Zeitpunkt oder bei Erfüllung einer Bedingung automatisch vom System ausgeführt wird. Ein „Benutzer" ist nicht notwendig, aber es muss jemanden geben, der von dem Prozess profitiert – andernfalls ergibt die Ausführung des Prozesses keinen Sinn. Als Product Owner oder Requirements Engineer sollten Sie immer nach diesem Empfänger der Leistung suchen. Stellen Sie sich die Frage: Wer möchte dieses Feature wirklich und sieht für sich einen Mehrwert darin?

Aus fachlicher Sicht ist es häufig sinnvoll, weniger von „User-Storys" zu sprechen, sondern sie einfach als „Storys" zu bezeichnen – so vermeidet man die explizite Referenz auf Benutzer. Sie müssen jedoch immer herausfinden, wer eine bestimmte Story wirklich möchte. Im folgenden Text verwenden wir, wann immer dies möglich ist, die Kurzform „Storys" als Alias für „User-Storys".

Wenn Sie diese Diskussion vermeiden möchten, beziehen Sie sich bei allen Aspekten einfach auf Backlog Items gemäß Scrum Guide [S@S Guide].

Hier ist ein Beispiel aus unserer Fallstudie für einen Prozess, der durch ein zeitliches Ereignis ausgelöst wird:

- Zwei Wochen vor Ablauf des Einschreibungszeitraumes sollen die Studenten an eine mögliche Verlängerung erinnert werden.

Wenn Sie dieses Feature als Story nach der Vorlage von Mike Cohn verfassen möchten, müssen Sie den Owner der Plattform als Leistungsempfänger benennen.

- Als Owner der Plattform möchte ich, dass den Studenten zwei Wochen vor Ablauf des Einschreibungszeitraumes eine Erinnerung zugesandt wird, damit sie die Gelegenheit haben, ihren Kontozugang zu verlängern.

3.3.3 INVEST: Kriterien für „gute" Storys

2003 veröffentlichte Bill Wake einen Artikel [Wake2003], in dem er für die INVESTititon in gute Storys plädierte. Wir haben die ersten drei Buchstaben dieses Akronyms bereits in Kapitel 3.1 besprochen: Storys sollten **I**ndependent, also unabhängig voneinander sein, sie sind **N**egotiable, das heißt verhandelbar, und sie müssen **V**aluable, also werthaltig für jemanden sein.

Damit sie gut genug für die Implementierung durch die Entwickler sind, müssen sie außerdem drei weitere Kriterien erfüllen: **E**stimated, das heißt geschätzt, **S**mall, also klein genug für die nächste Iteration und **T**estable, das heißt testbar.

Schätzmethoden behandeln wir in Kapitel 5.

Wenn die Schätzung ergibt, dass die Story immer noch zu groß für die Implementierung in einer Iteration ist, dann muss sie in mehrere Storys aufgeteilt werden. Techniken für das Aufteilen von Storys werden in Kapitel 3.4 besprochen.

Schließlich müssen Storys, wie in Kapitel über Bestätigung angesprochen, ausreichende Detailinformationen zu Testfällen oder Akzeptanzkriterien enthalten (gewöhnlich werden diese auf der Rückseite der Karte erfasst). Dies stellt eine Vereinbarung dar, die Entwickler dem Product Owner am Ende einer Iteration demonstrieren müssen. Siehe Kapitel 3.5.

3.3.4 Ergänzende Storys mit anderen Anforderungsartefakten

Wie zuvor erwähnt, enthält die Story auf einer Karte nicht alle Informationen zu einer Anforderung. Sie dient lediglich als ein physischer Hinweis, um die Kommunikation aller Stakeholder und Teammitglieder untereinander zu fördern. Manchmal ist es sehr nützlich, andere Anforderungsnotationen und Artefakte zu verwenden, um die Story auf einer Karte zu ergänzen.

Sie können Aktivitätsdiagramme, BPMN, Flussdiagramme oder Datenflussdiagramme verwenden – kurz gesagt, alles, was Sie auch bislang schon zur visuellen Darstellung eines Geschäftsprozesses oder einer Abfolge von Schritten verwendet haben.

Beispiel:

Um die Story „Als Student möchte ich ein Konto für die Schulungsplattform erstellen, damit ich überall Requirements Engineering-Wissen erwerben kann“ besser zu verstehen, könnten Sie das folgende Aktivitätsdiagramm hinzufügen:

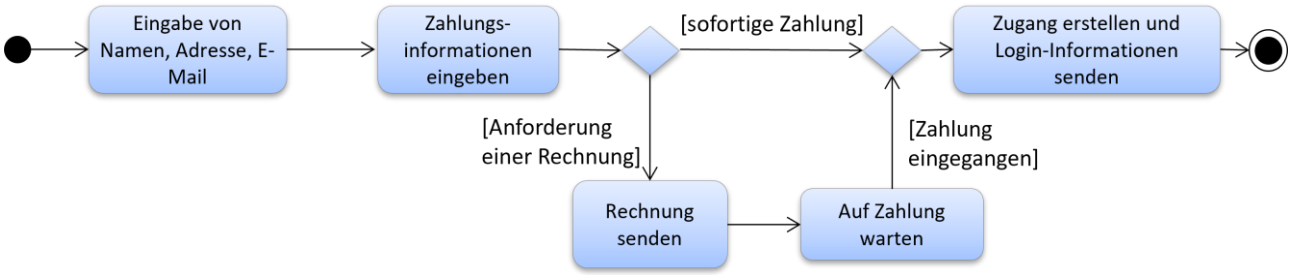

Abbildung 9: Aktivitätsdiagramm zur Erläuterung der Details einer Story

3.4 Aufteilungs- und Gruppierungstechniken

Um User-Storys zu erzeugen, die klein genug für eine einzige Iteration sind, können Sie größere Storys in detailliertere Storys aufteilen. Eine Reihe von Autoren haben Muster vorgestellt, die von der Verringerung der Featureliste bis zur Eingrenzung der geschäftlichen Variationen oder Informationskanäle reichen und für diesem Zweck verwendet werden können [Leffingwell2010].

Einer der umfassendsten Vorschläge stammt von Lawrence [Lawrence1], der eine einfach zu erlernende Methode in Form eines Cheat-Sheets empfiehlt. Lawrence zufolge sollten Sie sich die folgenden Fragen stellen, um zu kleineren Storys zu gelangen:

1. WORKFLOW: Beschreibt die Story einen Workflow? Wenn ja, können Sie sie so unterteilen, dass Sie sich zunächst um Anfang und Ende des Workflows kümmern und diese dann durch Storys aus dem mittleren Teil des Workflows erweitern? Oder können Sie zuerst einen schlanken Durchgang des Workflows erstellen und diesen später durch zusätzliche Storys erweitern?
2. MEHRERE VORGÄNGE: Beinhaltet die Story mehrere Vorgänge? (Geht es beispielsweise darum, etwas zu „managen“ oder zu „konfigurieren“? Können Sie die Vorgänge in separate Storys unterteilen?)
3. VARIATIONEN VON GESCHÄFTSREGELN: Enthält die Story verschiedene Geschäftsregeln? (Enthält die Story z. B. einen Bereichsbegriff wie „flexible Daten“, der auf mehrere Variationen schließen lässt?) Können Sie die Story so aufteilen, dass Sie zuerst eine Teilmenge der Regeln umsetzen und diese später durch zusätzliche Regeln erweitern?

4. VARIATIONEN BEI DATEN: Umfasst die Story dieselben Schritte für verschiedene Arten von Daten? Können Sie die Story so aufteilen, dass sie zunächst eine Art von Daten bearbeiten und sie später durch die anderen Arten von Daten erweitern?
5. VARIATIONEN BEI SCHNITTSTELLEN: Beinhaltet die Story eine komplexe Schnittstelle? Gibt es eine einfache Version, die Sie zuerst umsetzen könnten? Bezieht die Story dieselbe Art von Daten über mehrere Schnittstellen? Können Sie die Story so aufteilen, dass zunächst die Daten einer Schnittstelle behandelt werden und später die Daten der anderen Schnittstellen?
6. HAUPTAUFWAND: Wenn Sie die offensichtliche Aufteilung vornehmen, ist dabei die erste Story, die Sie bearbeiten, die schwierigste? Könnten Sie die späteren Storys gruppieren und die Entscheidung darüber, welche Story zuerst umgesetzt wird, verschieben?
7. EINFACH/KOMPLEX: Enthält die Story einen einfachen Kern, der den Großteil des Geschäftswerts oder Verbesserungspotentials umfasst? Könnten Sie die Story so aufteilen, dass zuerst dieser einfache Kern behandelt und dann in späteren Storys erweitert wird?
8. PERFORMANCE ZURÜCKSTELLEN: Ist die Komplexität der Story weitgehend darauf zurückzuführen, dass dadurch Qualitätsanforderungen wie die Performance erfüllt werden? Könnten Sie die Story so aufteilen, dass sie zunächst einfach funktioniert und Sie sie später erweitern, damit sie die Qualitätsanforderungen erfüllt?
9. Letztes Mittel: FÜHREN SIE EIN SPIKE DURCH: Haben Sie immer noch keine Idee, wie Sie die Story aufteilen können? Können Sie einen kleinen Teil ausmachen, den Sie gut genug verstehen, um damit zu beginnen? Wenn ja, schreiben Sie zunächst diese Story, entwickeln Sie sie und beginnen Sie dann erneut ganz oben bei den Vorschlägen. Falls nicht, können Sie eine bis drei Fragen definieren, die Sie am meisten bremsen? Führen Sie einen Spike mit diesen Fragen durch, tun Sie das Nötigste, um diese Fragen zu beantworten und beginnen Sie dann erneut ganz oben bei den Vorschlägen.

Beachten Sie, dass selbst detaillierte User-Storys so definiert werden sollten, dass sie für mindestens einen Stakeholder einen gewissen Wert liefern. Daher ist das Aufteilen eines Workflows in die einzelnen Schritte häufig kontraproduktiv, da das Implementieren des einen oder anderen Schrittes unter Umständen keinen Nutzen bietet. Daher schlägt [Hruschka2017] vor, einen Use Case (oder einen großen Prozess) besser in Scheiben, sogenannte Slices, zu zerlegen, die vom Anfang bis zum Ende reichen. Dies basiert auf Ivar Jacobsons Idee der Use Case Slices [Jacobson2011]. Abbildung 10 zeigt diese Idee in grafischer Form.

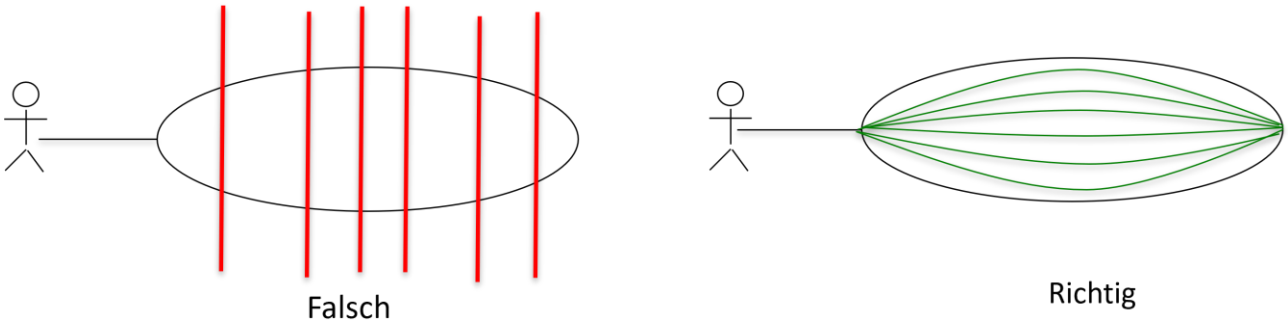

Abbildung 10: Use Case Slices anstelle von Prozessschritten

Das Erstellen von Slices kann nach verschiedenen Geschäftsobjekten oder nach Technologien erfolgen. Anschließend können Sie einen Slice für die frühzeitige Implementierung aussuchen und später noch weitere hinzufügen. Zudem können Sie ein Slice verkleinern, indem Sie:

1. Alternativen weglassen (nehmen Sie sich beispielsweise zuerst den normalen Ablauf vor und fügen Sie die Ausnahmefälle später hinzu),

2. Optionen weglassen (lassen Sie beispielsweise Dinge weg, die nicht unbedingt in einem frühen Release implementiert werden müssen) oder
3. Schritte weglassen, die in frühen Releases immer noch manuell erledigt werden können.

Wenn Sie ursprünglich Ideen für Storys hatten, die für die Schaffung eines Geschäftswertes zu klein waren (vor allem, wenn sie nicht unabhängig und nicht werthaltig sind – und dadurch gegen Kriterien des INVEST-Prinzips verstoßen), sollten Sie einige davon kombinieren oder anderweitig umformulieren, um gute, wenn auch große, Anfangsstorys zu bekommen.

Sehen Sie sich die folgenden Storys aus unserer Fallstudie an:

- Als Student möchte ich meinen Namen und meine Adresse eingeben, damit ich ein Konto anlegen kann.
- Als Student möchte ich meine E-Mail-Adresse zu meinem Konto hinzufügen, damit ich einen Link zum Kurs erhalte.

Dies ist zu detailliert für werthaltige Storys, da für die Geschäftsregel alle diese Daten zum Erstellen eines Kontos erforderlich sind. Besser wäre folgende Umformulierung:

- Als Student möchte ich ein Konto erstellen, damit ich Zugang zur Videoschulungsplattform erhalte.

Die Zerlegung und Gruppierung von Storys resultiert in Anforderungshierarchien, wie in Kapitel 3.1 erläutert. Wie Sie in Abbildung 11 sehen, lässt sich diese Hierarchie als zweidimensionale Story-Map [Patton2015] darstellen. Oberhalb der Trennlinie werden größere Gruppierungen (wie große Storys, Epics und Features) so ausgerichtet, dass die gesamte Bandbreite der Funktionalitä des Produkts abgedeckt ist. Das hilft dabei, den Überblick über die Anforderungen zu behalten. Unterhalb der Trennlinie können alle Informationen zu den größeren Gruppen detaillierter hinzugefügt und wie in einem linearen Backlog für die Zuordnung zu Sprints und Releases geordnet werden. Mit anderen Worten: In der Story-Map werden Backlogs pro Feature oder Epic dargestellt, wobei die Anforderungsstruktur auf höherer Ebene beibehalten wird.

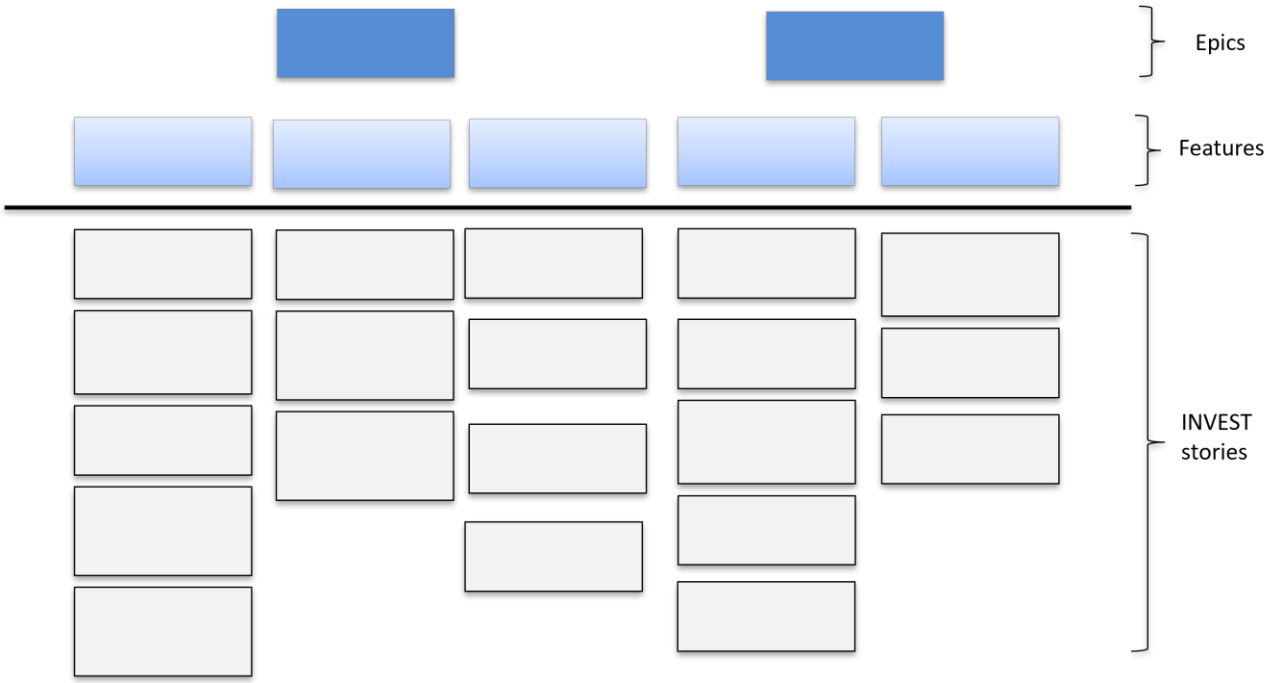

Abbildung 11: Die Struktur einer Story-Map

3.5 Man sollte wissen, wann man aufhören sollte

Der Product Owner ist für die Fortführung der Diskussionen mit Entwicklern verantwortlich, bis beide Seiten ein gemeinsames Verständnis von den Anforderungen erreicht haben [Meyer2014] Meyer. Um zu beurteilen, wann dieser Punkt erreicht ist, kann man auf das Pareto-Prinzip zurückgreifen: Anforderungen müssen nicht zu 100 % perfekt definiert sein, aber gut genug, um die zentralen Fragen des Teams aufzugreifen, und klar genug, damit der Implementierungsaufwand geschätzt werden kann. Beginnt man mit der Implementierung, wenn noch zu viele Fragen offen sind, kann das die Entwicklungsgeschwindigkeit erheblich verringern und gemessen an den Forecasts zu Verzögerungen führen.
Für diese Stufe des gemeinsamen Verständnisses wurde für Agile die Definition of Ready (DoR) [AgileAlliance] definiert.
Eine Story ist „ready“, also bereit, wenn sie die INVEST-Kriterien [Wake2003] erfüllt; dies gilt vor allem für die letzten drei Buchstaben des Akronyms:

- Die Entwickler konnten die Story schätzen.
- Die Schätzung ist klein genug, damit die Story in einer Iteration umgesetzt werden kann. Lawrence zufolge sollte die Story nicht nur in eine Iteration passen, sondern außerdem so klein sein, dass der nächsten Iteration 6 bis 10 Storys zugewiesen werden können [Lawrence1]. Um das zu erreichen, muss der Product Owner die Arbeitsgeschwindigkeit des Teams kennen. (Weitere Details zur Geschwindigkeit finden Sie in Kapitel 5.) Wenn das Team beispielsweise 28 Story-Points pro Sprint bearbeiten kann, dann sollten die User-Storys so klein sein, dass die Summe von 6 bis 10 Storys diesen Wert nicht überschreitet. Das Sprint Backlog sollte beispielsweise aus 8 Storys bestehen, mit 1, 1, 2, 3, 5, 8 und 8 Story-Punkten und es sollte ein klares Sprint-Ziel formuliert werden, für den Fall, dass das Team nicht alle Storys fertigstellen kann.
- Der Product Owner hat Akzeptanzkriterien für die Story definiert. Auf Basis des CCC-Prinzips stimmen alle darüber überein, dass genügend *Diskussionen* stattgefunden haben (*Conversation*) und die Kriterien zur *Bestätigung* (*Confirmation*) einer erfolgreichen Umsetzung in Form von Abnahmetests definiert wurden. Bei Verwendung von Karten (*Cards*) für die Erfassung der Storys, werden die Abnahmetests normalerweise auf der Rückseite der Karte notiert.

Falls eine Story bereits klein genug für einen Sprint ist, haben Product Owner die Wahl: Sie können diese Story beibehalten und weitere Abnahmetests auf der Karte hinzufügen. Alternativ können sie die Story in mehrere Storys aufteilen, für die dann gewöhnlich weniger und einfachere Abnahmetests gebraucht werden.

Die Formulierung von Akzeptanzkriterien kann auf unterschiedliche Art und Weise erfolgen [Beck2002]. Die Akzeptanzkriterien können in informeller, natürlicher Sprache formuliert sein und nach der Implementierung geprüft werden.

Mit Hilfe der Gherkin-Syntax [WyHT2017] können die Akzeptanzkriterien etwas formaler beschrieben werden. Gherkin ist eine für Menschen lesbare, domänenspezifische Sprache, die speziell für die Beschreibung von erwarteten Verhaltensweisen von Produkten entwickelt wurde. Sie bietet die Möglichkeit, logische Details aus dem Test der Verhaltensweise zu entfernen.

Gherkin schlägt die folgende Struktur für das Verfassen von Testszenarios vor:

- **Scenario:** <<kurzer deskriptiver Name>
- **Given** <<eine Vorbedingung>>
- **And** <<eine andere Vorbedingung>>

- **When** <<eine Benutzeraktion>>
- **And** <<eine andere Aktion>>
- **Then** <<ein testbares Ergebnis wird erreicht>>
- **And** <<es tritt auch etwas anderes ein, das wir prüfen können>>

Einige Methoden befürworten sogar die Verwendung von TDD (Test Driven Development). Statt der Verwendung einer domänenspezifischen Sprache (DSL) wie Gherkin, können Sie die Testfälle formal codieren, damit sie nach der Implementierung automatisch ausgeführt werden können [Meyer2014] Meyer. Dieser formale Ansatz ist zwar sehr präzise, kann jedoch für Product Owner und betriebswirtschaftlich orientierte Stakeholder schwer durchzuführen und zu verstehen sein.

Die Definition of Ready (DoR) ist für den Product Owner das Äquivalent zur Definition of Done (DoD) der Entwickler. Nach den in der DoD festgelegten Kriterien wird bestimmt, ob eine Story erfolgreich implementiert wurde. Die DoR hingegen definiert, dass die Entwickler genügend Informationen über eine User-Story haben, um sie innerhalb einer Iteration zu erledigen („done“).

Die Diskussion von Anforderungen mit Entwicklern benötigt Zeit und findet am besten vor der Planung einer Iteration statt. Bei der Planung kann dann der Fokus auf die Auswahl der richtigen User-Storys gelegt werden und deren Zuordnung zu den verantwortlichen Entwicklern. Im Idealfall haben die Entwickler die Entstehung der Anforderungen mitverfolgt und den Product Owner mit ihren Fragen und Schätzungen unterstützt.

Es sind auch unterschiedliche Formen möglich ein Produkt Backlog Refinement durchzuführen. Zum Beispiel sind Refinement Meetings eine effizientere Möglichkeit, um Verfeinerungen durchzuführen, als einzelne Entwickler wiederholt zu stören. Die Zeit für Refinement Meetings und sämtliche dazugehörigen Aktivitäten geht von der für die gesamte Iteration verfügbaren Zeit ab. Im Scrum Guide [S@S Guide] wird empfohlen, maximal 10 % der Kapazität der Entwickler für die Verfeinerung zu verwenden: Sollte dafür mehr Zeit erforderlich sein, dann ist das ein Warnzeichen für qualitativ schlechte Anforderungen. Ein Product Owner sollte das Verhältnis zwischen Länge der Iteration, Risiko und Verwaltungsaufwand für die Iteration kennen. Weiter sollte er wissen, dass es kürzere Feedbackschleifen als die Iteration an sich gibt.

3.6 Projekt- und Produktdokumentation von Anforderungen

Agile Projekte, insbesondere Scrum-Projekte, verwenden einen Product Backlog, der eine priorisierte Liste der in einem Produkt oder Service zu entwickelnden Funktionalitäten darstellt. Obwohl Product Backlog Items alles sein können, was das Team wünscht, haben sich Epics, Features und User Storys als die beliebtesten Formen von Product Backlog Items etabliert.

Ein Product Backlog kann man sich als Ersatz für das Anforderungsdokument eines herkömmlichen Projekts vorstellen. Es ist jedoch wichtig, in Erinnerung zu behalten, dass der schriftliche Teil einer agilen User-Story („Als Benutzer möchte ich…“) unvollständig ist, bis die Diskussionen über diese Story stattgefunden haben.

Es ist oft am besten, sich den schriftlichen Teil als Hinweis auf eine präzisere Darstellung dieser Anforderung vorzustellen. User-Storys könnten auf ein Diagramm hinweisen, das einen Workflow darstellt, auf eine Tabellenkalkulation, in der die Durchführung einer Berechnung veranschaulicht wird, oder auf ein beliebiges anderes Artefakt, das der Product Owner oder das Team wünscht.

Im RE@Agile Primer [Primer2017] haben wir vier verschiedene Zwecke für die Anforderungsdokumentation dargelegt.

Betrachten wir einmal die ersten beiden Zwecke:

a) **Dokumentation für Kommunikationszwecke**: Effektive und effiziente Kommunikation ist bei agilen Methoden ein wichtiges Werkzeug, da sie interaktiv ist und kurze Feedbackzyklen ermöglicht. In der Praxis gibt es verschiedene Konstellationen, die eine unmittelbare verbale Kommunikation verhindern: geografisch verteilte Teams, Sprachbarrieren oder zeitliche Einschränkungen bei den Beteiligten. Darüber hinaus sind Informationen manchmal so komplex, dass direkte Kommunikation ineffizient oder irreführend sein kann. Ein Prototyp in Papierform oder ein Diagramm eines komplizierten Algorithmus können später beispielsweise noch einmal gelesen werden. Manchmal ziehen Stakeholder einfach schriftliche Kommunikation dem Lesen von Quellcode oder dem Prüfen von Software vor. In diesen Fällen vereinfacht Dokumentation den Kommunikationsprozess zwischen allen Beteiligten und die Ergebnisse des Prozesses werden aufbewahrt.

Für das Erstellen einer Dokumentation für Kommunikationszwecke gilt folgendes Prinzip: Ein Dokument wird als zusätzliches Mittel der Kommunikation erstellt, wenn Stakeholder oder die Entwickler in dem Dokument einen Wert sehen. Verlief die Kommunikation erfolgreich, sollte das Dokument archiviert werden.

b) **Dokumentation zum Zwecke von Überlegungen**: Es gibt einen Aspekt beim Schreiben eines Dokuments, der häufig vergessen wird: Der Schreibvorgang ist immer ein gutes Mittel zur Verbesserung und Unterstützung der Erkenntnisprozesse des Schreibenden. Selbst wenn das Dokument zu einem späteren Zeitpunkt im Prozess verworfen wird, bleibt der Vorteil der Verbesserung und Unterstützung des Denkvorgangs bestehen. Das Schreiben eines Anwendungsfalls beispielsweise zwingt den Schreibenden, über konkrete Interaktionen zwischen dem System und den Akteuren nachzudenken, einschließlich z. B. Ausnahmen und alternativen Szenarios. Aus diesem Grund kann das Schreiben eines Anwendungsfalls als Werkzeug zum Testen des eigenen Wissens und Verständnisses eines Systems betrachtet werden.

Für das Erstellen von Dokumentation als Teil des Erkentnisprozesses gilt folgendes Prinzip: Die nachdenkende Person entscheidet über die Dokumentform, die ihre Denkvorgänge am besten unterstützt. Die nachdenkende Person muss diese Entscheidung nicht rechtfertigen. Das Dokument kann verworfen werden, wenn der Überlegungsprozess abgeschlossen ist.

Für die ersten beiden Zwecke ist ein Product Backlog mit Epics und Storys (in welcher Form auch immer (Karten an der Wand oder in Werkzeugen erfasste Storys) und vielleicht ergänzt durch Skizzen, Diagramme und Prototypen) als Dokumentation ausreichend, um den Fortschritt der Produktentwicklung zu unterstützen.

Für die beiden anderen Zwecke ist eine formalere Anforderungsdokumentation zu berücksichtigen.

c) **Dokumentation für gesetzliche Zwecke**: Bestimmte Domänen oder Projektkontexte (z. B. Software im Gesundheitssektor oder in der Avionik) erfordern die Dokumentation bestimmter Informationen (z. B. Anforderungen und Testfälle für ein System) für eine bestimmte Zielgruppe, damit eine gesetzliche Genehmigung erteilt wird.

Für die Erstellung einer Dokumentation für gesetzliche Zwecke gilt folgendes Prinzip: In den geltenden Gesetzen und Normen ist dargelegt, welche gesetzlich erforderliche Dokumentation erstellt werden muss. Diese Dokumentation ist ein untrennbarer Bestandteil des Produkts.

d) Dokumentation zum Zwecke der Bewahrung: Bestimmte Informationen über ein System haben einen über den initialen Entwicklungsaufwand hinaus anhaltenden Wert. Beispiele dafür sind die Ziele des Systems, die zentralen Anwendungsfälle, die es unterstützt oder Entscheidungen, die während der Entwicklung getroffen wurden, um z.B. bestimmte Funktionalitäten auszuschließen. Die Dokumentation zu Bewahrungszwecken kann zum gemeinsamen Archiv des Teams, eines Produkts oder einer Organisation werden. Sie verringert die Abhängigkeit von der Erinnerung einzelner Teammitglieder, und sie kann Diskussionen über vorherige Entscheidungen unterstützen (z. B. „Warum haben wir uns gegen diese Implementierung entschlossen?“).

Das Prinzip der Dokumentation zu Bewahrungszwecken ist Folgendes: Das Team entscheidet, was zum Zwecke der Bewahrung dokumentiert wird.

Für diese beiden Zwecke ist das Product Backlog – das ein Werkzeug für die Interaktion von Product Ownern mit Entwicklern ist – nicht ausreichend. Die gute Nachricht ist, dass Dokumentation für gesetzliche Zwecke oder zum Zweck der Bewahrung des Know-hows über Produktanforderungen nicht vorab erstellt werden muss.

Sie kann jedes Mal aktualisiert und gepflegt werden, wenn eine neue Version des Produkts veröffentlicht wird, beispielsweise nach der erfolgreichen Implementierung von Features. Daher umfasst sie nur die Dokumentation von Funktionalitäten, Rahmenbedingungen und Qualitätsanforderungen, die das Produkt tatsächlich enthält. So werden zeitaufwendige Aktivitäten zum Versions- und Konfigurationsmanagement für Dokumente vermieden, während Stakeholder noch in den Verhandlungen sind und vielleicht ihre Meinungen ändern.

Das Definieren eines adäquaten Maßes an Dokumentation hängt von zahlreichen Faktoren ab: etwa vom Umfang der Projekte, von der Anzahl der beteiligten Stakeholder, von rechtlichen Randbedingungen und/oder von den sicherheitskritischen Aspekten des Projekts. Basierend auf diesen Faktoren versuchen agile Teams, ein Übermaß an Dokumentation zu vermeiden und einen minimalen Nenner an nützlicher Dokumentation zu finden.

Das Arbeiten mit einem „lebenden" Product Backlog ist zwar eine effiziente Weise des Umgangs mit Dokumentation, doch sie ist nicht immer ausreichend. Eine strukturierte Dokumentation aller in einem Produkt implementierten Anforderungen, die auf dem neuesten Stand ist, ist in manchen Projekten nicht nur rechtlich verpflichtend, sondern sie dient auch als idealer Ausgangspunkt für die schnellere Identifizierung von Änderungsanträgen basierend auf der vorhandenen Dokumentation.

3.7 Zusammenfassung

Was immer Sie von Ihren Stakeholdern über benötigte Funktionalitäten erfahren, ist der richtige Ausgangspunkt für Anforderungen. Es ist aber lediglich der Ausgangspunkt. Ihre Aufgabe als Product Owner ist es, diese funktionalen Anforderungen zu strukturieren.

Epics, Themen, Features oder große Storys (die potenziell komplexe Geschäftsprozesse darstellen) sind eine gute Möglichkeit, ein Gesamtbild zu erhalten, d. h. einen Überblick über all die Dinge, die Ihre Stakeholder von einem System oder einem Produkt erwarten. Sie haben jedoch gelernt, dass diese – per Definition – nicht präzise genug sind, um auf dieser Präzisionsstufe aufhören zu können.

Ihr Ziel für eine gute Anforderungsarbeit ist es, User-Storys zu erstellen, die die Definition of Ready oder die INVEST-Kriterien erfüllen: Sie sollten unabhängig und werthaltig sein, klein genug, damit sie in einer Iteration umgesetzt werden können, schätzbar und mit testbaren Abnahmekriterien. Die Schablone von Mike Cohn „Als <Benutzer> möchte ich <eine Funktionalität>, um <bestimmte Ziele> zu erreichen“ ist ein guter Ausgangspunkt, aber Sie sollten nicht in jedem Fall darauf bestehen, diese Formel zu verwenden.

Ist eine Anforderung immer noch zu groß für eine Iteration, dann können Sie diese mithilfe einer der verschiedenen Vorgehensweisen, die Sie gelernt haben, aufteilen. Versuchen Sie dabei aber, so viel Unabhängigkeit und Werthaltigkeit wie möglich beizubehalten.

4. Umgang mit Qualitätsanforderungen und Randbedingungen

In Kapitel 3 lag der Schwerpunkt auf dem Umgang mit funktionalen Anforderungen. Der Umgang mit funktionalen Anforderungen, das heißt, herauszufinden, welche Funktionalitäten die verschiedenen Stakeholder benötigen, wird die zeitaufwändigste Aktivität in der Systementwicklung sein und die meisten Diskussionen zwischen Product Owner, Stakeholdern und den Entwicklern prägen.

Bestimmte Qualitätsmerkmale (der Funktionen) des Systems, etwa Leistung, Benutzerfreundlichkeit, Zuverlässigkeit und Erweiterbarkeit, werden häufig vorausgesetzt. Benutzer und/oder andere Stakeholder gehen häufig davon aus, dass diese nicht explizit angegeben werden müssen, da die Entwickler sie bereits kennen.

Das gleiche gilt für organisatorische und technische Randbedingungen. Es weiß doch jeder, dass wir ein Standardprozessmodell haben, für das bestimmte Artefakte erstellt werden müssen, oder? Ist nicht jedem bekannt, dass wir unsere Datenbanksysteme immer bei Unternehmen X einkaufen und unseren Code selbstverständlich in der Sprache Y entwickeln?

Experten im Requirements-Engineering verteidigen die Bedeutung dieser „nicht-funktionalen“ Anforderungen schon seit Jahrzehnten. Auch wenn der Begriff „nicht-funktionale Anforderungen“ in der Praxis immer noch häufig als Überbegriff für Qualitätsanforderungen und Randbedingungen zu finden ist, verwendet das IREB die konkreteren und präziseren Kategorien „Qualitätsanforderungen“ und „Randbedingungen“ nach [Glinz2014].

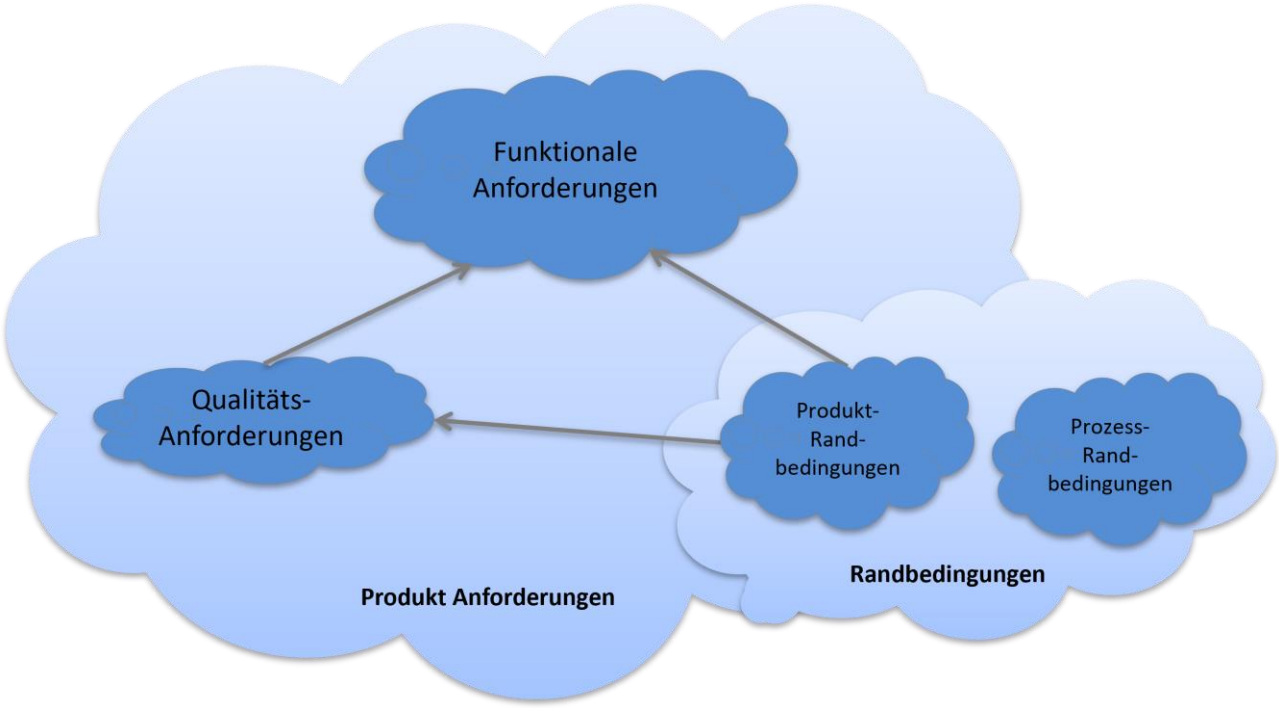

Abbildung 12: Kategorisierung von Anforderungen

Abbildung 12 zeigt die drei Kategorien von Anforderungen und einige ihrer wichtigen Beziehungen. Eine Qualitätsanforderung steht niemals für sich alleine, das heißt, sie bezieht sich immer auf eine oder mehrere – oder sogar alle – funktionalen Anforderungen. Randbedingungen sind entweder Produktrandbedingungen, die das Design einer Funktion oder einer Qualitätseigenschaft einschränken, oder Prozessrandbedingungen, die die Arbeit der Entwickler auf eine Weise einschränken, die nicht direkt mit dem Produkt in Verbindung steht, beispielsweise, wenn bestimmte Prozessschritte ausgeführt oder bestimmte Artefakte erstellt werden müssen.

Zu Beginn sind Qualitätsanforderungen und Randbedingungen oftmals bewusst vage formuliert. In den nächsten Kapiteln beschreiben wir, wie solche vagen Qualitätsanforderungen und Randbedingungen erfasst werden. Sie erfahren außerdem, wie Sie vage Qualitätsanforderungen und Randbedingungen in präzisere Anforderungen (bis hin zur Spezifizierung präziser Akzeptanzkriterien) transformieren und wie Sie diese in Verbindung mit funktionalen Anforderungen handhaben können.

4.1 Die Bedeutung von Qualitätsanforderungen und Randbedingungen verstehen

[Meyer2014] Meyer äußert die Sorge, dass „viele agile Methoden ausschließlich auf funktionale Anforderungen ausgerichtet sind und Qualitätsanforderungen und Randbedingungen nicht genügend Gewicht einräumt werden“. Bertrand Meyer führt weiter aus: „Die für ein System vorgesehenen zentralen Randbedingungen und Qualitätsanforderungen sollten im Lebenszyklus eines Produkts frühzeitig explizit benannt werden, da sie für wichtige Entscheidungen hinsichtlich der Architektur ausschlaggebend sind (Infrastruktur, Softwarearchitektur und Softwaredesign). Werden sie in einem Projekt ignoriert oder zu spät erkannt, gefährdet dies unter Umständen die gesamte Entwicklungstätigkeit. Andere Qualitäten können iterativ erfasst werden, just in time, so wie es auch bei funktionalen Anforderungen der Fall ist.

Es sind zwar viele Kategorien von Qualitätsanforderungen zu berücksichtigen, doch eine Reihe von veröffentlichten Kategorisierungsschemata – oder Checklisten – erleichtern Product Ownern die Arbeit etwas, wie die beiden folgenden Beispiele verdeutlichen. Als Product Owner sollten Sie einfach einen dieser „Spickzettel“ verwenden, um explizite Fragen zu den Qualitätsaspekten zu stellen. Oder noch besser: Erstellen Sie auf Basis der verfügbaren Checklisten Ihre eigene Checkliste, um die Qualitätsaspekte zu betonen, die in Ihrer Domäne am wichtigsten sind.

2011 veröffentlichte die ISO eine neue Reihe von Qualitätsnormen als Ersatz für das bekannte Qualitätsmodell ISO/IEC 9126 aus dem Jahr 2001. Die wichtigste Norm für das Requirements Engineering ist [ISO25010], in der Qualitätsanforderungen definiert werden. Sie wurde zuletzt 2017 aktualisiert. Abbildung 13 zeigt die acht Qualitätsmerkmale von Systemen der obersten Ebene und deren Zerlegung in Untermerkmale. Beachten Sie, dass in der Norm nicht die Rede von Anforderungen ist, sondern von Qualitätsmerkmalen des Systems. Durch das Hinzufügen des Begriffs „Anforderung“ zu den einzelnen Kategorien können Sie Ihre Bedürfnisse zu diesem Bereich diskutieren; „Kapazität“ wird beispielsweise zu „Kapazitätsanforderungen“.

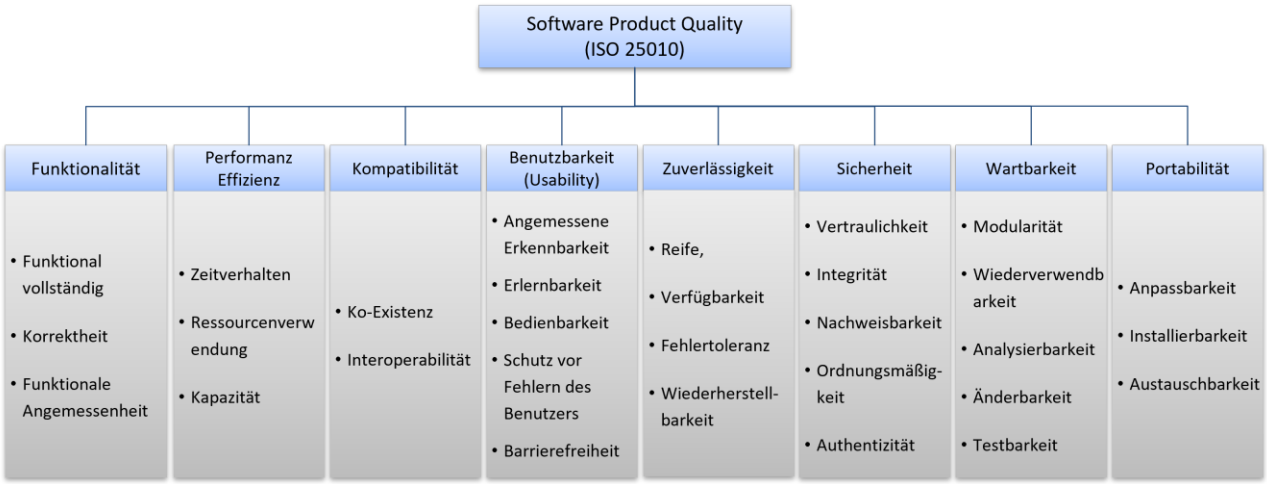

Abbildung 13: Kategorien von Qualitäten nach ISO25010

In der Norm finden Sie detaillierte Definitionen all dieser Kategorien. Zusätzlich zu dem allgemeinen Qualitätsmodell enthält die Norm ISO/IEC 25012 ein ergänzendes Modell für Datenqualität.

Ein ähnliches Kategorisierungsschema für Qualitätsanforderungen finden Sie im VOLERE Template [RoRo2017]. In den Kapiteln 10 bis 17 werden dort Kategorien von Qualitätsanforderungen beschrieben. Die Kategorisierung beruht auf jahrzehntelanger Erfahrung im Bereich der Systemspezifikation. In der Originalvorlage wird jeder Kategorie der Begriff „Anforderungen" hinzugefügt, das heißt „Langlebigkeit" lautet dann „Langlebigkeitsanforderungen". In Abbildung 14 haben wir diesen Zusatz ausgelassen, damit die Kategorien besser lesbar bleiben.

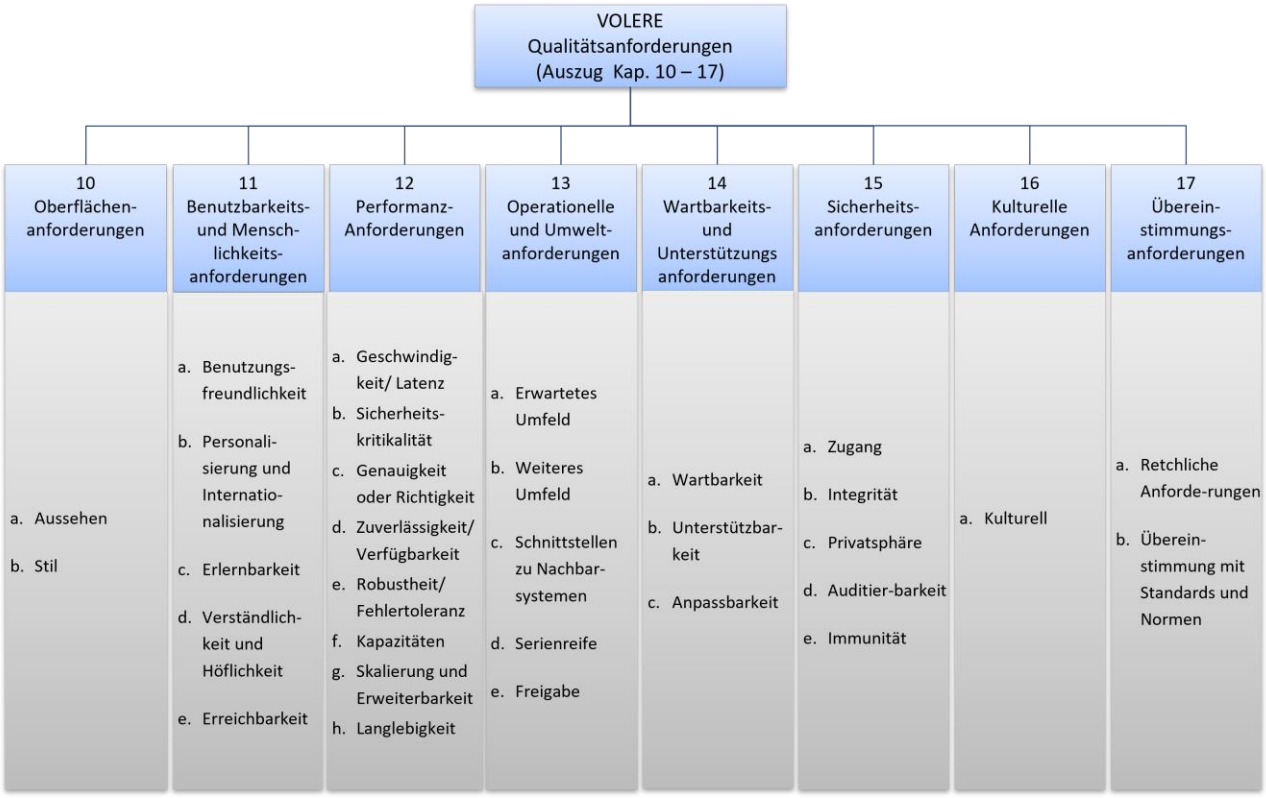

Abbildung 14: Qualitätskategorien von VOLERE

In [RoRo2013] finden Sie nicht nur Definitionen für all diese Kategorien, sondern auch Gründe dafür, weshalb sie so wichtig sind. Sie finden auch Beispiele dafür, wie Sie die Kategorien formulieren, einschließlich der Akzeptanzkriterien.

Das folgende Beispiel stammt von http://volere.co.uk/template.htm [RoRo2017]. Beachten Sie, dass Akzeptanzkriterien („Acceptance Criteria") in dieser Publikation als Abnahmekriterien („Fit Criteria") bezeichnet werden.

11 c. Lernanforderungen

Inhalt

Anforderungen, die spezifizieren, wie einfach das Erlernen der Produktverwendung sein sollte. Diese Lernkurve reicht von keinem Zeitaufwand für Produkte, die zur Platzierung im öffentlichen Bereich vorgesehen sind (z. B. eine Parkuhr oder eine Website) bis hin zu einem beträchtlichen Zeitaufwand für komplexe, hochgradig technische Produkte.

Motivation

Zur Quantifizierung des für Ihren Kunden akzeptablen Zeitaufwands, nach dem ein Benutzer das Produkt erfolgreich verwenden kann. Diese Anforderung dient Designern als Richtlinie beim Verstehen der Art und Weise, wie Benutzer das Produkt kennenlernen. Designer können beispielsweise ausgefeilte interaktive Hilfsmittel oder ein Tutorial in das Produkt einbauen. Alternativ muss das Produkt gegebenenfalls so gestaltet werden, dass sämtliche Funktionalitäten bei der ersten Verwendung offensichtlich sind.

Beispiele

Das Produkt sollte für einen Techniker einfach zu erlernen sein.

Ein Mitarbeiter sollte damit innerhalb kurzer Zeit produktiv arbeiten können.

Das Produkt sollte von Personen verwendet werden können, die vor der Verwendung keine Schulung erhalten.

Das Produkt sollte von Technikern verwendet werden können, die vor der Nutzung eine fünfwöchige Schulung erhalten.

Abnahmekriterium

Ein Techniker sollte ein [festgelegtes Ergebnis] innerhalb einer [bestimmten Zeit] erreichen können, wenn er mit der Verwendung des Produkts beginnt, ohne dass er dazu das Handbuch verwenden muss.

Nachdem ein Mitarbeiter [Anzahl von Stunden] Schulung durchlaufen hat, sollte er [Menge eines festgelegten Ergebnisses] pro [Zeiteinheit] herstellen können.

[Vereinbarter Prozentsatz] einer Testruppe soll/en [eine bestimmte Aufgabe] innerhalb [eines festgelegten Zeitlimits] erfolgreich abschließen.

Die Techniker sollen bei der abschließenden Prüfung am Ende der Schulung eine Erfolgsquote von [vereinbarter Prozentsatz] erreichen.

Übungsvorschlag:

Diskutieren Sie für einige der in Abbildung 13 oder Abbildung 14 dargestellten Kategorien darüber, ob die Entwickler frühzeitig von diesen Anforderungen erfahren sollten oder ob sie später im Entwicklungsprozess berücksichtigt werden können.

4.2 Qualitätsanforderungen präzisieren

Qualitätsanforderungen müssen an die Entwickler sowohl eindeutig als auch testbar kommuniziert werden. Wie zuvor erwähnt, sind Qualitätsanforderungen zu Beginn häufig sehr vage. Zum Beispiel: *Die neue Generation von Mobiltelefonen soll für Teenager attraktiv sein.*

Diese Qualitätsanforderung ist weder eindeutig noch testbar (in der Art und Weise, wie sie ausgedrückt ist), doch sie kann der Ausgangspunkt für Diskussionen über detailliertere Qualitätsmerkmale sein, die für die nächste Generation von Mobiltelefonen erforderlich sind.

Deren Präzision (oder eher deren fehlende Präzision) lässt sich mit einem funktionalen Epic wie „Als Mobiltelefonbenutzer hätte ich gerne intelligente Wählfunktionen“ vergleichen. In Kapitel 3 haben wir behandelt, wie wir ein solches Epic auf eine Präzisionsstufe bringen, auf der es von den Entwicklern implementiert werden kann.

In diesem Kapitel tun wir dasselbe für Qualitätsanforderungen. Wir erklären zunächst, wie Qualitätsanforderungen konkretisiert werden können, und zwar bis auf die Stufe von Akzeptanzkriterien. Anschließend – in Kapitel 4.3 – beschreiben wir, wie und wo man diese (physisch) dokumentiert oder speichert.

Es gibt zwei Möglichkeiten, vage Qualitätsanforderungen präziser und klarer zu formulieren. Sie können entweder mehr Details hinzufügen oder sie zerlegen, oder Sie können präzisere (funktionale) Anforderungen aus der ursprünglichen Anforderung ableiten. In Abbildung 15 sind diese Alternativen grafisch dargestellt.

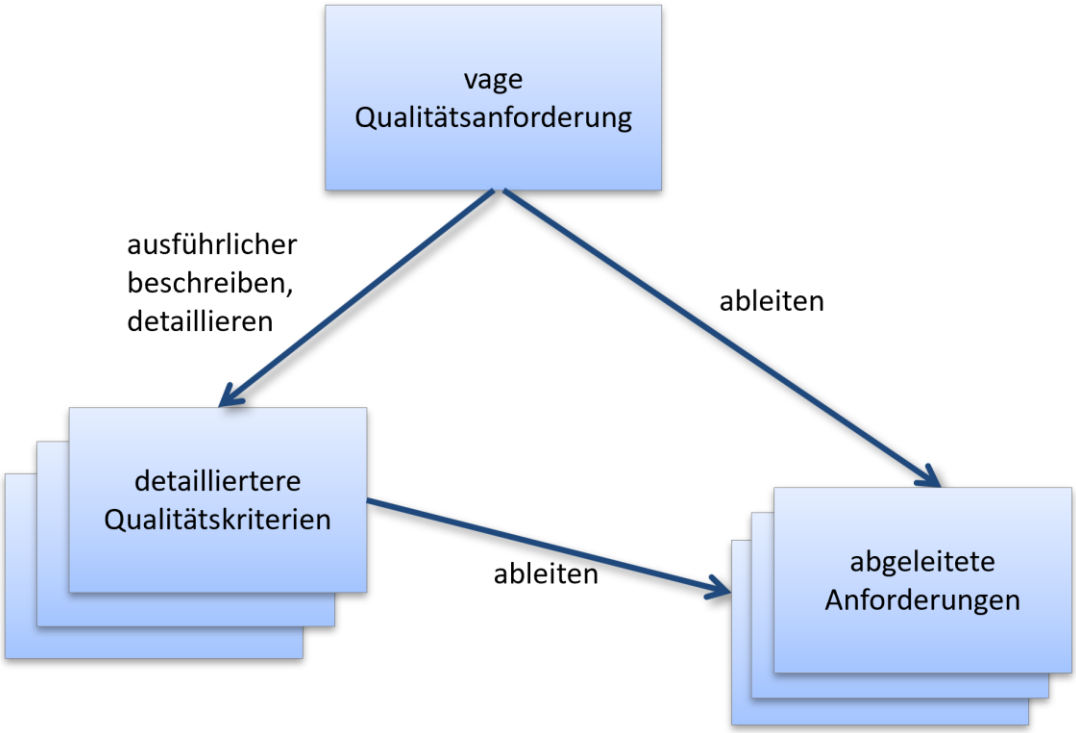

Abbildung 15: Qualitätsanforderungen detaillierter beschreiben und zerlegen

Bei der detaillierteren Beschreibung oder Zerlegung wird die ursprüngliche, vage Qualitätsanforderung hergenommen und durch zwei oder mehrere detaillierte Qualitätsanforderungen ersetzt.

Beispiel: Wenn wir das Kategorisierungsschema in Abbildung 14 betrachten, könnte man die Anforderung für Benutzerfreundlichkeit (VOLERE-Kategorie 11) „*Das System sollte benutzerfreundlich sein*“ durch die folgenden beiden Anforderungen detaillierter darstellen:

- Als Benutzer möchte ich, dass das System einfach zu erlernen ist (VOLERE-Kategorie 11 c) und
- als Benutzer möchte ich, dass das System leicht zu bedienen ist (VOLERE-Kategorie 11 a).

Diese beiden Anforderungen sind immer noch vage, aber bereits präziser als die ursprüngliche Anforderung.

Die zweite Alternative „ableiten“ bedeutet, dass die ursprüngliche Qualitätsanforderung in eine oder mehrere (funktionale) Anforderungen transformiert wird.

Nehmen wir beispielsweise die ursprüngliche Anforderung: „Als Verantwortlicher für Sicherheit möchte ich Zugriff auf die folgenden Funktionen, die nur für autorisiertes Personal verfügbar sind.“

Die Ableitung präziserer Anforderungen bedeutet beispielsweise, zu entscheiden, dass ein Login-Mechanismus mit Benutzername und Passwort für die Zugriffsbeschränkung verwendet wird.

Beachten Sie, dass die ursprüngliche Absicht der Qualitätsanforderung lediglich darin bestand, den Zugriff für bestimmte Funktionen zu schützen. Es ist eine Design-Entscheidung, dies durch die Einführung von Rollen und Passwörtern zu erreichen. Es sind auch andere Ideen denkbar, beispielsweise das Wegsperren des Computers in einem Raum, zu dem nur autorisierte Personen Zugang haben. Alternativ könnten Sie Fingerprints verwenden, um berechtigte Benutzer zu identifizieren.

Wenn Sie neue funktionale Anforderungen von ursprünglichen Qualitätsanforderungen ableiten, möchten Sie die ursprüngliche Anforderung vielleicht aufbewahren, beispielsweise, um sich an deren Ursprung zu erinnern, falls Sie in zukünftigen Versionen des Produkts geschicktere Möglichkeiten finden, um die Originalqualität zu erreichen.

Das Ableiten neuer funktionaler Anforderungen von erforderlichen Qualitätsanforderungen bringt Sie einer Lösung oder einer Erfüllung dieser Anforderung näher.

Übungsvorschlag:
Wählen Sie eines Ihrer Produkte aus und verfeinern Sie einige Qualitätsanforderungen.

Qualitätsbäume [Clements et al.2001] sind ebenfalls ein bewährtes Verfahren, um Qualitätsanforderungen zu strukturieren. In einem Qualitätsbaum werden die beiden oben genannten Techniken kombiniert. Abbildung 16 zeigt die allgemeine Form eines Qualitätsbaums. Er beginnt mit einer Wurzel mit der Bezeichnung „*Specific Quality*“ (eine spezifische Qualitätseigenschaft). Die nächsten Zweige des Baumes sind Qualitätskategorien, gefolgt von Unterkategorien. Die Blätter des Baumes enthalten konkrete Szenarios für eine Kategorie oder Unterkategorie, beispielsweise funktionale Anforderungen oder testbare Qualitätsaussagen.

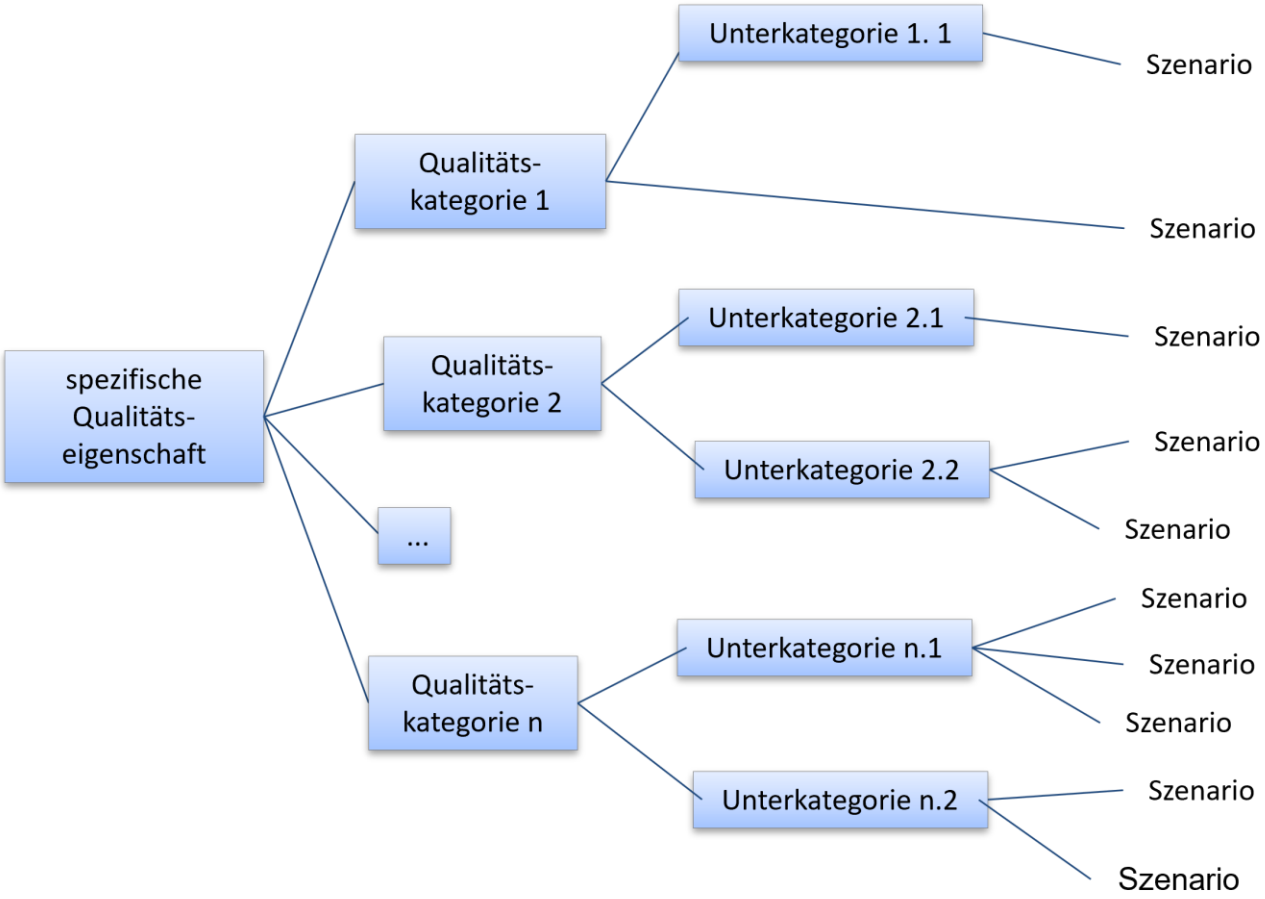

Abbildung 16: Allgemeines Schema für einen Qualitätsbaum

Für unsere Fallstudie iLearnRE werden in Abbildung 17 Auszüge aus einem Qualitätsbaum dargestellt. Beachten Sie die folgenden Punkte:

- Die Blätter sind unter Umständen immer noch nicht genau genug für einen Test, beispielsweise „usable without training of students“ (ohne Schulung der Studenten nutzbar). Deshalb werden für Qualitätsanforderungen Akzeptanzkriterien benötigt, um die Entwickler über die Erwartungen des Product Owners zu informieren.

- In der Anforderung für „other languages“ (andere Sprachen) ist eine sehr klare geschäftliche Entscheidung enthalten. Der Product Owner hat gemeinsam mit allen Stakeholdern entschieden, dass Untertitel für die Vermarktung des Produkts in anderen Ländern genügen, anstatt die Videos beispielsweise zu synchronisieren.
- In der Anforderung zur „Adaptability“ (Anpassbarkeit) ist sogar ein Design-Vorschlag enthalten: Anstatt lediglich anzugeben, dass das System auf verschiedenen Arten von Geräten mit unterschiedlichen Auflösungen funktionieren soll, fordert der Product Owner die Verwendung der Standard-Unternehmenstechnologie an: responsives Design.

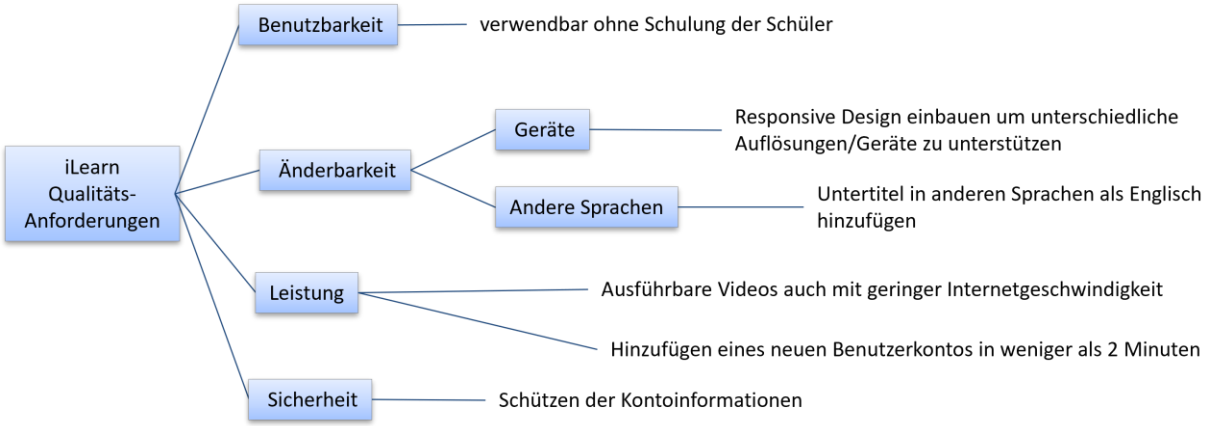

Abbildung 17: Teile eines Qualitätsbaumes für iLearnRE

Übungsvorschlag:

Führen Sie ein Brainstorming zu einem Teilqualitätsbaum für eines Ihrer Produkte durch. Stellen Sie sicher, dass Sie als Blätter sehr konkrete Szenarios haben!

Wie zuvor erwähnt, sind für Qualitätsanforderungen auch Akzeptanzkriterien erforderlich, um mehr Präzision zu erreichen. Die Art der verwendeten Akzeptanzkriterien hängt von der Qualitätskategorie ab. Die folgende Tabelle enthält systematische Ratschläge zur Formulierung von Akzeptanzkriterien für unterschiedliche VOLERE-Qualitätskategorien.

Art der Anforderung	Empfohlene Skalierung
10 Erscheinungsbild	Normenkonformität – geben Sie an, wer dies testet bzw. wie dies getestet wird
11 Benutzbarkeit	Zeitdauer zum Erlernen Schulungsdauer Testgruppe kann Funktionen in Zielzeit ausführen
12 Leistung	Zeit zur Vervollständigung einer Aktion
13 Operativ	Quantifizierung der Zeit/einfache Benutzbarkeit in der Umgebung
14 Änderbarkeit	Quantifizierung des Aufwands für die Übertragbarkeit Spezifikation der zulässigen Zeit für Änderungen
15 Sicherheit	Spezifikation, wer das Produkt wann verwenden kann
16 Kulturell und politisch	Wer akzeptiert die Quantifizierung spezieller Gepflogenheiten
17 Rechtlich	Meinung eines Rechtsanwalts/Prozess

Die folgenden Kapitel enthalten Beispiele für Akzeptanzkriterien zu Qualitätsanforderungen. Weitere Informationen dazu finden Sie in [RoRo2013].

Benutzbarkeitsanforderung: Das Produkt muss von einer Person verwendet werden können, die gegebenenfalls nicht Englisch spricht.

Akzeptanzkriterium: 45 von 50 zufällig ausgewählten nicht Englisch sprechenden Personen müssen in der Lage sein, das Produkt innerhalb des Leistungskriteriums plus 25 % zu verwenden.

Leistungsanforderung: Das Produkt muss eine akzeptable Geschwindigkeit aufweisen.

Akzeptanzkriterium: Die einzelnen Vorgänge des Verkaufsautomats dürfen nicht länger als 15 Sekunden dauern.

Operative Anforderung: Als Arbeiter muss ich das Produkt auch im Freien, bei Kälte und Regen verwenden können.

Akzeptanzkriterium: 90 % der Arbeiter müssen das Produkt im ersten Nutzungsmonat innerhalb der angestrebten Zeit erfolgreich verwenden.

Sicherheitsanforderungen: Nur direkte Vorgesetzte dürfen die Personaldaten ihrer Mitarbeiter einsehen. Personaldatensätze von Mitarbeitern dürfen für niemand anderen einsehbar sein.

Akzeptanzkriterium: Aufzeichnung von Zugriffen und Tests, um zu ermitteln, ob jemand Zugriff hatte, der kein Vorgesetzter ist. Alternativ könnten Sie sagen, dass die Konformität des Produkts nach der Sicherheitsnorm xyz zertifiziert sein muss.

Gesetzliche Anforderung: Personenbezogene Kundendaten dürfen nur in Übereinstimmung mit dem Datenschutzgesetz verwendet werden.

Akzeptanzkriterium: Die Rechtsabteilung muss dem zustimmen, dass das Produkt der Datenschutzregistrierung der Organisation entspricht.

Übungsvorschlag:

Nehmen Sie zwei Beispiele für Qualitätsanforderungen und fügen Sie diesen Akzeptanzkriterien hinzu.

4.3 Qualitätsanforderungen und Backlog

Wir haben besprochen, wie Qualitätsanforderungen erkannt und ermittelt werden und wie vage Qualitätsanforderungen präzisiert werden können. Nun betrachten wir, wie diese in agilen Umgebungen in Verbindung mit einem Product Backlog, das überwiegend funktionale Anforderungen enthält, dokumentiert werden. Abhängig von der Art der Qualitätsanforderung, ist der eine oder andere der folgenden Ansätze eine gute Lösung für Sie.

Am einfachsten lässt sich eine Qualitätsanforderung dokumentieren, indem man sie direkt einem Backlog Item hinzufügt. Dieser Ansatz funktioniert nur dann, wenn die Qualität einzigartig ist, also nur bei diesem Feature oder der User-Story vorkommt.

Ein zweiter Ansatz ist die Dokumentation von Qualitätsanforderungen außerhalb des Backlogs, entweder:

- Auf separaten Karten;
- Als Qualitätsbaum.

Sie müssen die Qualitätsanforderungen in beiden Fällen mit allen relevanten funktionalen Anforderungen verknüpfen. In Abhängigkeit von den verwendeten Werkzeugen, können Sie dies entweder mit Hyperlinks tun oder Sie müssen die funktionalen Anforderungen und die einzelnen, darauf abzielenden Qualitätsanforderungen nummerieren.

Als dritte Alternative können Qualitätsanforderungen in der Definition of Done erfasst werden. Da die Regeln in der Definition of Done für SÄMTLICHE Iterationen gelten, geben Sie damit an, dass diese Anforderungen immer erfüllt werden müssen, unabhängig davon, welche funktionalen Anforderungen Sie der nächsten Iteration hinzufügen.

4.4 Randbedingungen verdeutlichen

Randbedingungen sind eine wichtige Art von Anforderungen. Glinz definiert Randbedingungen als Anforderungen, die den Lösungsraum darüber hinaus eingrenzen, was für die Erfüllung der gegebenen funktionalen Anforderungen und Qualitätsanforderungen erforderlich ist [Glinz2014]. Das Produkt muss innerhalb der Randbedingungen erstellt werden. Randbedingungen beschränken den Spielraum für Entscheidungen und beeinflussen und formen dadurch das Produkt.

Sie werden entweder von Ihren Vorgesetzten oder von Stakeholdern außerhalb Ihres Kontrollbereichs vorgegeben, wie etwa Aufsichtsbehörden, Ihrer Muttergesellschaft oder einem Enterprise Architect.

Beachten Sie, dass viele Randbedingungen zwar legitim sind, es aber häufig für Product Owner oder die Entwickler lohnend ist, deren Validität zu prüfen und mit den Personen oder Organisationen, die Ihrer Entwicklung solche Randbedingungen setzen, zu verhandeln und deren Beweggründen und Motivation auf den Grund zu gehen.

Gelegentlich werden Sie feststellen, dass einige der Randbedingungen reine Überlieferungen sind, und dass Sie diese – sobald Sie sie infrage stellen und Alternativen vorschlagen – mit den zuständigen Stakeholdern verhandeln und lockern können, was Ihnen eine flexiblere Implementierung ermöglicht. In agiler Terminologie ausgedrückt: Randbedingungen können ebenso wie Funktionalitäten verhandelbar sein. Wenn die anderen Parteien jedoch auf den Randbedingungen bestehen, müssen die Entwickler sie akzeptieren.

In diesem Handbuch haben wir gesetzliche Bestimmungen oder (etwas allgemeiner) jede Art von Compliance-Anforderungen als Kategorie von Qualitätsanforderungen aufgenommen (siehe Kapitel 4.1). Sie könnten ebenso in dieses Kapitel über Randbedingungen eingefügt werden, da jede Lösung diese Qualitätsanforderungen erfüllen muss. Verglichen mit den anderen Kategorien von Randbedingungen sind solche Compliance-Anforderungen häufig nicht verhandelbar.

Abbildung 12 zeigt eine Möglichkeit der Kategorisierung von Randbedingungen: Sie können entweder als Produkt- oder als Prozessrandbedingungen eingestuft werden. Lediglich Produktrandbedingungen beziehen sich auf funktionale Anforderungen oder Qualitätsanforderungen des Produkts und grenzen daher deren Implementierung ein. Prozessrandbedingungen weisen keine direkte Beziehung zum Produkt auf. Sie setzen der Organisation, die das Produkt entwickelt oder dem Entwicklungsprozess für das Produkt, Grenzen. Dadurch wirken sie sich nur indirekt auf das eigentliche Produkt aus.

In Abbildung 18 werden einige Unterkategorien für diese beiden Kategorien vorgeschlagen. Nachfolgend werden einige Beispiele besprochen. Weitere Details zur Formulierung solcher Randbedingungen und weitere Beispiele finden Sie in [RoRo2013].

In den Produktrandbedingungen wird möglicherweise eine bestimmte Infrastruktur gefordert, das heißt, eine technologische und/oder physische genau definierte Umgebung, in der das Produkt installiert werden soll. Andere Beispiele umfassen die obligatorische Verwendung von Standardsoftware (das heißt, die Entscheidung zum Kauf gegenüber der Entwicklung von Subsystemen innerhalb des Projekts).

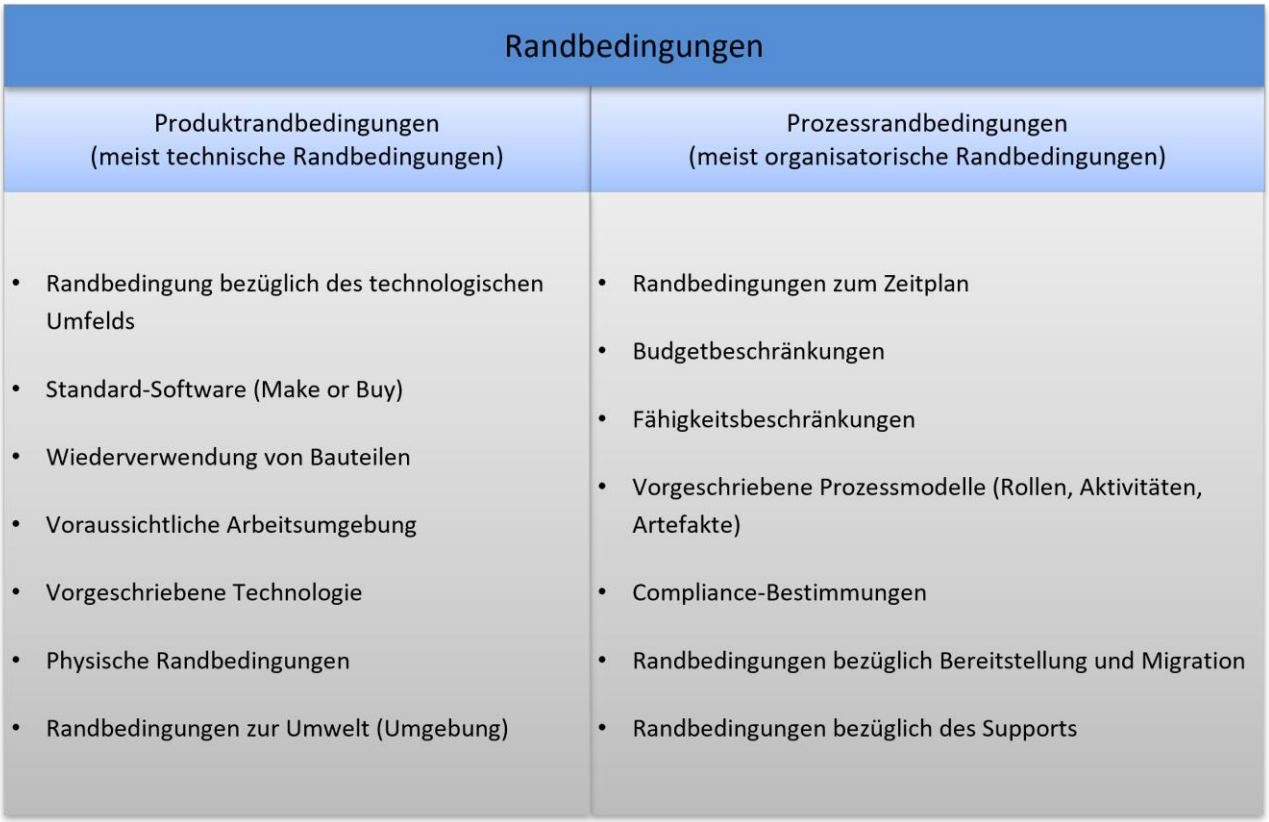

Abbildung 18: Kategorisierung von Randbedingungen

Eine häufig gestellte Randbedingung ist, dass vorhandene Komponenten oder Subsysteme von Vorgängerprodukten oder anderen Produkten, die das Unternehmen entwickelt hat, wiederverwendet werden sollen. Der Grund für die Wiederverwendung ist offensichtlich: Sie möchten kein Geld ausgeben, wenn Sie bereits akzeptable (Teil-) Lösungen zur Verfügung haben.

Randbedingungen, die die erwartete operative Umgebung des Produkts betreffen, beschreiben alle Features des Arbeitsplatzes, die sich auf das Design auswirken könnten.

Produktdesigner sollten beispielsweise wissen, dass der Arbeitsplatz laut ist und Audiosignale daher vielleicht nicht funktionieren.

Umgekehrt sollte der Geräuschpegel eines Produkts, das in ruhigeren Umgebungen eingesetzt werden soll, einen bestimmten Dezibelwert nicht überschreiten. Befindet sich der Arbeitsplatz unter freiem Himmel, wo es nass und kalt sein kann, dann sollten die Nutzer das Produkt mit Handschuhen verwenden können.

Ähnliches gilt für Systeme mit Hardwarebestandteilen: Hier können physische Randbedingungen in Bezug auf die Größe oder das Gewicht des Geräts – denken Sie nur an Mobiltelefone oder Handheldgeräte – ebenfalls von hoher Relevanz sein (d. h. relevant im Hinblick auf das Hardwaredesign und die Software, die es unterstützt).

Am häufigsten schränken Produktrandbedingungen allerdings die Technologie ein, die die Entwickler verwenden dürfen.

Beispiele sind:

- Als Enterprise Architect möchte ich, dass Sie das Produkt in C# entwickeln, damit unsere Mitarbeiter das Produkt warten können.
- Als Datenbankadministrator möchte ich, dass das Produktteam ORACLE verwendet, da wir für dieses Produkt eine ausgezeichnete Support-Hotline haben.

Beachten Sie, dass Sie Randbedingungen nicht als Storys verfassen müssen. Es ist gegebenenfalls ausreichend, das Team darüber zu informieren, dass C# und ORACLE nicht verhandelbare Randbedingungen sind.

Prozessrandbedingungen bezeichnet man häufig als organisatorische Randbedingungen, da sie entweder Managementaspekte wie das Budget, die Zeitplanung oder die Kompetenzen von Teammitgliedern einschränken, die für das Projekt verfügbar sind („Sie müssen mit diesem Team arbeiten. Wir haben kein Budget, um zusätzliche oder externe Mitarbeiter zu beschäftigen. "), oder da sie bestimmte Richtlinien oder Vorschriften durchsetzen. Sie müssen unter Umständen einem Entwicklungsprozess folgen, der bestimmte Rollen, während der Entwicklung auszuführende obligatorische Aktivitäten und Dokumente oder andere Artefakte vorgibt, die erstellt und gepflegt werden müssen.

Für Randbedingungen gibt es, ebenso wie für andere Arten von Anforderungen, eine Beschreibung: Sie kann einen Beweggrund oder eine Motivation enthalten, die beschreibt, weshalb es die Randbedingung gibt. Und sie sollten, genau wie funktionale Anforderungen oder Qualitätsanforderungen, Akzeptanzkriterien aufweisen.

Wenn Sie eine Zeit lang in einer Organisation gearbeitet haben, haben Sie vermutlich die technologischen Präferenzen der Organisation kennengelernt und werden die organisatorischen Regelungen und Randbedingungen kennen. Dennoch ist es wichtig, diese Randbedingungen explizit zu machen, damit sich alle Teammitglieder dieser bewusst sind. Randbedingungen, die für die größten Einschränkungen sorgen, sollten im Projektverlauf frühzeitig bekannt sein. Sonstige Randbedingungen sollten erfasst werden, sobald sie erkannt werden.

Derartige Randbedingungen gelten normalerweise für eine Vielzahl von Projekten. Grundlegende Technologie-Stacks ebenso wie Prozessmodelle werden in einem Unternehmen normalerweise für einen längeren Zeitraum festgelegt. Wenn die Randbedingungen also einmal erfasst wurden, können sie leicht in anderen Produktentwicklungen wiederverwendet werden.

4.5 Zusammenfassung

Qualitätsanforderungen und Randbedingungen sind genauso wichtig für den Projekterfolg wie funktionale Anforderungen. Für einen Product Owner ist es nicht schwierig, relevante Anforderungen in diesen Kategorien zu finden, da es viele öffentlich zugängliche Checklisten mit Vorschlägen für Kategorien von Qualitäten und Randbedingungen gibt.

Qualitätsanforderungen können anfangs vage formuliert sein. Um für die Entwicklung bereit zu sein, müssen sie auf das Niveau von Abnahmetests präzisiert werden – genau wie funktionale Anforderungen.

Die Präzisierung von Qualitätsanforderungen wird häufig dadurch erreicht, dass man entsprechende funktionale Anforderungen, welche die ursprünglich erforderlichen Qualitätsanforderungen erfüllen, ableitet. Stellen Sie sicher, dass solche Entscheidungen dokumentiert und die ursprünglichen Qualitätsanforderungen nicht verworfen werden, da Sie mit der Zeit eventuell bessere Möglichkeiten finden, um die Qualitätsanforderungen zu erfüllen.

Einige Qualitätsanforderungen können einfach zu bereits ermittelten User-Storys hinzugefügt werden, beispielsweise indem man einzelne Funktionen durch Leistungs- oder spezielle Sicherheitsaspekte ergänzt. Viele Qualitätsanforderungen betreffen bereichsübergreifende Aspekte, das heißt, sie sind für viele funktionale Anforderungen relevant.

Wir schlagen vor, dass Sie für diese übergreifenden Qualitätsanforderungen eine separate Liste pflegen, welche für die Entwickler immer sichtbar sein muss, da sie stets erfüllt werden muss. Alternativ können Sie die Qualitätsanforderungen in die Definition of Done aufnehmen, da die DoD alle Aspekte festlegt, welche immer erfüllt sein müssen.

Für technische, organisatorische und rechtliche Randbedingungen kann ein ähnlicher Ansatz gewählt werden. Stellen Sie sicher, dass diese den Entwicklern explizit bekannt sind. Wenn es sich nicht um projektspezifische, sondern eher um allgemeine unternehmensspezifische Bestimmungen handelt, können Sie diese an einer zentralen Stelle für alle Projekte pflegen und so für viele Entwicklungsprojekte wiederverwenden.

5. Priorisieren und Schätzen von Anforderungen

Agile Herangehensweisen zielen darauf ab, im Laufe der Zeit den gesamten Geschäftswert zu maximieren und den Erstellungsprozess des Geschäftswertes dauerhaft zu optimieren [Leffingwell2010]. Dieser konstante, wertsteigernde Prozess ist in Abbildung 19 dargestellt. Jede Iteration sollte zu einer Wertsteigerung führen – manchmal mehr, manchmal weniger.

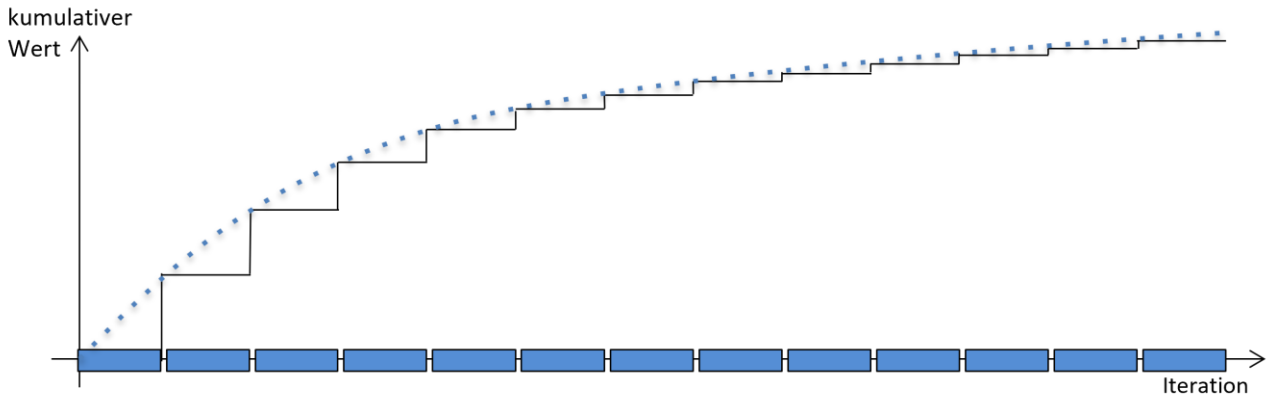

Abbildung 19: Agile Entwicklung = konstante Wertschaffung

Jede Iteration soll ein potenziell releasefähiges Produktinkrement liefern, das den Wert des Produkts insgesamt steigert. (Hinweis: In einigen Versionen von Scrum und anderen agilen Ansätzen wird dies als „potenziell auslieferbares Produkt“ oder „potenziell verwendbares Produktinkrement“ bezeichnet.)

[LeSS] erklärt dieses Ziel wie folgt: „Potenziell auslieferbar ist eine Aussage über die Qualität der Software und nicht über den Wert oder die Vermarktbarkeit der Software. Wenn ein Produkt potenziell auslieferbar ist, bedeutet das, dass die gesamte Arbeit, die für die aktuell implementierten Features aufgewendet werden muss, erledigt wurde und das Produkt aus technischer Perspektive auslieferbar ist. Es heißt jedoch nicht, dass die implementierten Features für den Kunden ausreichend werthaltig sind, so dass er ein neues Release haben möchte. Letzteres bestimmt der Product Owner.“

Zum Planen und Erreichen konstanter Wertschöpfung, sollten sämtliche Anforderungen (ob grob- oder feingranular) in erster Linie auf Basis des Mehrwerts sortiert werden, den sie dem Geschäft liefern können. Geschäftswert kann für verschiedene Organisationen allerdings viele verschiedene Dinge bedeuten. Die Klärung des Begriffs „Geschäftswert“ ist eines der zentralen Themen dieses Kapitels, das in den Unterkapiteln 5.1 bis 5.3 behandelt wird.

Natürlich muss die Wertschöpfung mit dem Aufwand für ihre Schaffung und dem Zeitpunkt, zu dem der Wert geliefert wird, ausgewogen sein. Deshalb müssen die Entwickler den Product Owner mit Schätzungen zu den Aufwänden, die zur Schaffung des Geschäftswertes erforderlich sind, unterstützen. Die Schätzung von Backlog Items ist das zweite zentrale Thema dieses Kapitels, das im Unterkapitel 5.4 besprochen wird. Basierend auf dem Wert-/Aufwand-Verhältnis kann der Product Owner die Storys auswählen, die die Entwickler in der nächsten Iteration übernehmen.

5.1 Ermittlung des Geschäftswerts

Wie zuvor erwähnt, kann „Wert" in verschiedenen Kontexten viele verschiedene Dinge bedeuten. Hier sind einige Aspekte, die der Festlegung des Geschäftswertes und beim Sortieren der Backlog Items nach diesem Wert berücksichtigt werden sollten.

- Wert für den Kunden oder andere Stakeholder

  Wenn Sie ein Produkt für einen bestimmten Kunden oder Auftraggeber entwickeln, dann beeinflusst dessen Meinung darüber, was mehr und was weniger wichtig ist, definitiv Ihre Entwicklungsreihenfolge der Backlog Items. Nicht jeder Stakeholder betrachtet Geld als ein Kriterium für Wert. Für Greenpeace könnte Wert beispielsweise alles Gute sein, das Sie zum Schutz der Umwelt tun. Das heißt, alles, was Ihr Kunde oder wichtiger Stakeholder am meisten wertschätzt, wird berücksichtigt.

- Wert für das Unternehmen

  Auch wenn Sie bestimmte Auftraggeber haben, die das Produkt verwenden oder kaufen, kann (oder sollte) das Unternehmen selbst strategische Ziele haben, die es erreichen möchte, beispielsweise das Erstellen einer wiederverwendbaren Plattform für einen bestimmten Bereich, damit zukünftige individuelle Projekte schneller und kostengünstiger durchgeführt werden können. Tatsächlich kann jede Art von Optimierung und Automatisierung interner Geschäftsprozesse eine treibende Kraft für die Wertschöpfung des Unternehmens sein. Wenn die Backlog Items eng an solche strategischen Ziele geknüpft sind, wird deren Geschäftswert als sehr hoch betrachtet.

- Existenzbedrohung

  Ein bestimmtes Feature oder eine Funktionalität nicht zu haben, kann für das Produkt oder die Organisation insgesamt eine Bedrohung darstellen. Typische Beispiele für solche Bedrohungen sind gesetzliche Bestimmungen (z. B. für den Datenschutz). Ein solches Feature bietet vielleicht keinen Mehrwert im wirtschaftlichen Sinn, aber es muss implementiert werden, um die weitere Existenz des Produkts oder des Unternehmens sicherzustellen.

- Erwarteter finanzieller Wert eines Features (Umsatz, Gesamterlös, Investitionsrendite)

  Ziel der meisten kommerziellen Organisationen ist es, Geld zu verdienen (Gewinne zu machen). Features und Storys, die mehr Umsatz oder eine schnelle Investitionsrendite versprechen, werden also naturgemäß höher eingestuft.

- Kurzfristige Projekt- oder Release-Ziele (vs. mittelfristige Produktziele)

  Manchmal ist es wichtig, Features oder zumindest Mock-ups von Features auf einer anstehenden Fachmesse oder bei einer wichtigen Präsentation demonstrieren zu können. Daher wertschätzen Product Owner solche Ergebnisse unter Umständen mehr als diejenigen, die zur längerfristigen Produktstrategie beitragen. Andererseits möchte ein Unternehmen möglicherweise in ein Entwicklungs-Framework investieren, das nicht unmittelbar einen Geschäftswert erzeugt, aber langfristige Entwicklungskosten senkt und das Mehrwertverhältnis für anstehende Produktinkremente verbessert.

- Kosten für Verzögerungen (Costs of Delay)

  Dieses Kriterium ist für die Bestimmung des Geschäftswerts sehr interessant. Die entscheidende Frage lautet: Wie hoch sind die Kosten einer verzögerten Auslieferung einer Story? Ein neues Feature für ein Online-Shoppingportal soll den Umsatz um 500.000 US-Dollar pro Monat steigern. Wenn dieses Feature also mit einem Monat Verspätung zur Verfügung steht, bedeutet das einen Verlust in Höhe von 500.000 US-Dollar für das Unternehmen. Reinertsen [Reinertsen2008] betrachtet die Kosten für Verzögerungen (Costs of Delay – CoD) als einen Gesichtspunkt, unter dem alle anderen in diesem Kapitel erwähnten Aspekte zusammengefasst werden können.

- Markteinführungszeiten

  Für bestimmte Features kann es bestimmte, günstige Gelegenheiten geben. Ein Beispiel: Wenn dieses Feature innerhalb dieses Zeitraums verfügbar ist, sorgt dies für einen deutlichen Anstieg des Geschäftsvolumens. Wird es zu spät fertig, kann der Wert deutlich geringer ausfallen. Fachmessen sind beispielsweise eine gute Gelegenheit, um neue Produkte zu verkaufen. Wenn das Produkt bei Eröffnung der Messe nicht fertig ist, kaufen die Kunden vielleicht ein anderes Produkt und brauchen das Produkt in der näheren Zukunft nicht, selbst wenn es mehr und bessere Funktionalitäten aufweist. Manche Methoden empfehlen daher, jedem Backlog Item eine Eigenschaft mit „Haltbarkeitsdatum“ zuzuweisen. Auf diese Weise ist jeder Stakeholder über den Zeitraum einer Gelegenheit informiert.

- Häufigkeit von Anforderungen

  Wenn Sie ein Produkt für einen Massenmarkt entwickeln, kann es bei der Bestimmung des Geschäftswerts wichtig sein, einen Überblick über die Nachfrage zu bekommen. Wurde es von vielen Kunden nachgefragt? Oder lediglich von einer kleinen Gruppe? Wie viel Umsatz erwarten Sie, basierend auf der Anzahl der Kunden, die das Feature angefragt haben?

- Geschäftliche und technische Abhängigkeiten

  Manchmal müssen Sie ein Backlog Item priorisieren, da es eine Voraussetzung für ein oder mehrere andere Backlog Items sein kann, das heißt, dass die anderen Items nicht entwickelt werden können, solange dieses Item nicht verfügbar ist. Ein Beispiel in der Fallstudie iLearnRE: Die Entwicklung eines Benutzerkontos schafft an sich keinen Geschäftswert. Sie können allerdings keine personalisierten Features entwickeln, wenn Sie das Benutzerkonto noch nicht entwickelt haben. Diese Abhängigkeiten könnten auch technischer Natur sein: Die Entwicklung eines Features erfordert beispielsweise den Aufbau einer bestimmten Infrastruktur oder es müssen gewisse Werkzeuge angeschafft und geprüft werden, bevor Sie das Feature ausliefern können. Diese Voraussetzungen (Features) schaffen zwar keinen Geschäftswert, aber ohne Erledigung dieser Voraussetzungen können Sie die wirklich werthaltigen Backlog Items nicht entwickeln.

Einige Qualitäten könnten auch als werthaltig betrachtet werden. Sie könnten Backlog Items priorisieren, die beispielsweise:

- Die Benutzbarkeit verbessern;
- Die Zuverlässigkeit verbessern;
- Die Wartungskosten verringern;
- Die Auswirkungen auf das aktuelle System minimieren.

Die Arbeit an solchen Qualitätsverbesserungen erzeugt nicht sehr oft neue marktfähige Features und damit keinen direkten Umsatz. Sie können jedoch von bestimmten Stakeholder-Gruppen als sehr wichtig erachtet werden und daher unter den Backlog Items einen hohen Rang einnehmen.

Der gelieferte Wert kann nur aufseiten des Endbenutzers gemessen werden, weil der Endbenutzer des Produkts entscheidet, ob er das Produkt verwenden (und kaufen) möchte, und ob er es anderen potenziellen Kunden empfiehlt. In der Folge könnte der Umsatz des Herstellerunternehmens steigen.

Handelt es sich um einen internen Kunden, kann kein Umsatz gemessen werden. In diesem Fall wird der Wert des ausgelieferten Produktinkrements typischerweise durch eine sprintweise Bewertung des ausgelieferten Produktinkrements und der resultierenden Produktversion sowie durch den Vergleich mit der Produkt-Roadmap basierend auf den geplanten und ausgelieferten Features und Produktfunktionen ermittelt.

5.2 Geschäftswert, Risiko

Ein wichtiges Kriterium für die Priorisierung von Backlog Items ist, dass einige risikoreicher sind als andere.

[DeMaLi2003] gibt eine zyklische Definition von Risiken und Problemen:

- Ein Risiko ist ein potenzielles Problem.
- Ein Problem ist ein manifestiertes Risiko.

In der Produktentwicklung gibt es zahlreiche Kategorien von Risiken. Das Feature an sich kann ein Risiko bergen, da es beispielsweise möglich ist, dass es vom Zielpublikum nicht akzeptiert wird. Das Risiko könnte in der Implementierung eines Features liegen, etwa wenn das Team eine bestimmte Technologie verwenden möchte, aber nicht alle Teammitglieder in dieser Technologie erfahren sind.

Das Risiko könnte auch in der Technologie selbst liegen, wenn sie z. B. zu neu (und daher riskant in der Anwendung) oder zu alt bzw. überholt ist. Einen umfassenden Überblick über Risiken, vor allem die fünf Hauptrisiken, die sich auf jedes IT-Projekt auswirken, bietet [DeMaLi2003].

Die risikoreichen Backlog Items liefern möglicherweise keinen hohen Geschäftswert, wenn man die im letzten Kapitel definierten Kriterien zugrunde legt. Wenn Sie jedoch mit den Risiken umgehen möchten, um spätere Überraschungen zu vermeiden, dann sollten Sie Backlog Items, die mit einem Risiko verbunden sind, frühzeitig im Entwicklungsprozess angehen. Sobald Sie sich um diese Backlog Items gekümmert haben, bedeutet die übrige Arbeit ein geringeres Risiko.

Wenn Sie risikoreiche Backlog Items haben, stehen Ihnen vier Alternativen zur Auswahl:

1. Das Risiko vermeiden: Das heißt, risikoreiche Backlog Items nicht behandeln. Das Vermeiden solcher Items impliziert allerdings, dass Ihnen auch die damit verbundenen Chancen entgehen. Vermeiden sollte also im Umgang mit risikoreichen Items nicht Ihre Wahl sein.
2. Risiken mindern: Als Manager können Sie Geld zur Seite legen und/oder Zeit freihalten, um sich um Risiken zu kümmern, sobald sich diese zu Problemen auswachsen. Als Product Owner (verantwortlich für das Requirements Engineering) können Sie daher die detaillierte Untersuchung solcher Items verschieben, bis sie für das Geschäft an Bedeutung gewinnen.
3. Risiken reduzieren: Neben der Minderung ist das die zweite offensichtliche Wahl, um mit risikoreichen Items umzugehen. Das bedeutet jedoch, dass Sie jetzt Maßnahmen ergreifen müssen, um das Risiko zu verringern. In der Regel unterteilen Sie ein risikoreiches Item in kleinere Items (beispielsweise Spikes), damit Sie mehr über deren risikoreiche Bestandteile erfahren können. Sie entwickeln zum Beispiel einen UI-Prototyp, um sicherzustellen, dass das Zielpublikum diese akzeptiert, oder Sie entwickeln einen Prototyp, um Erfahrungen mit einem neuen Framework zu sammeln.
4. Hoffen, dass sich das Risiko nicht in ein Problem verwandelt. Dies ist, ähnlich wie die erste Alternative, keine gangbare Wahl. Stellen Sie sich vor, dass Sie zwölf Risiken haben, die eine Wahrscheinlichkeit von jeweils nur zehn Prozent aufweisen. Nach den Regeln der Mathematik liegt die Möglichkeit, dass eines davon eintritt, bereits bei 75 Prozent.

Als Product Owner entscheiden Sie sich nur für die Alternativen zwei und drei. Aus der Anforderungsperspektive ist Alternative drei die wichtigste. Sie müssen Wege finden, eine Anforderung so zu zerlegen, dass das Risiko reduziert wird. Manchmal untersuchen Sie ein Spike oder entwickeln einen Prototyp, um das Risiko zu verringern, bevor Sie mit der tatsächlichen Entwicklung eines Features beginnen.

[DeMaLi2003] schlussfolgert: „Der wahre Grund, weshalb wir Risikomanagement betreiben müssen, ist nicht die Vermeidung von Risiken, sondern die Möglichkeit offensiv an Risiken heranzugehen."

Übungsvorschlag:

Diskutieren Sie, welche (Kombinationen von) Kriterien in Ihrer Organisation verwendet werden, um (geschäftlichen) Wert zu ermitteln.

5.3 Äußern von Prioritäten und Sortieren des Backlogs

Sobald Sie festgelegt haben, was Wert für Sie bedeutet, müssen Sie diese Prioritäten zum Ausdruck bringen und das Backlog nach den Prioritäten der Backlog Items sortieren. Es gibt viele verschiedene Methoden, um Backlog Items einen Wert zuzuweisen. Einige davon sind sehr einfach, andere hochkomplex. Im folgenden Kapitel behandeln wir einige gängige Ansätze.

Eine Methode ist der Einsatz von MoSCoW. Diese Priorisierungsmethode wurde von [ClBa1994] entwickelt, um mit Stakeholdern ein gemeinsames Verständnis darüber zu erreichen, welche Bedeutung sie der Auslieferung der einzelnen Anforderungen beimessen. Der Begriff MoSCoW ist ein Akronym, das aus den ersten Buchstaben der vier Priorisierungskategorien besteht (Must have, Should have, Could have und Won't have – muss, sollte, kann und wird nicht vorhanden sein). Dazwischen wurden zwei o eingefügt, damit man das Wort aussprechen kann.

Die Kategorien werden in der Regel so verstanden:

- **Must have:** Anforderungen, die als *Must have* (muss vorhanden sein) gekennzeichnet sind, sind für den Erfolg der aktuellen Auslieferungs-Timebox kritisch. Selbst, wenn nur eine *Must-have*-Anforderung nicht enthalten ist, sollte die Projektauslieferung als gescheitert betrachtet werden (beachten Sie: Anforderungen können bei Zustimmung aller relevanten Stakeholder von einem *Must have* herabgestuft werden, beispielsweise, wenn neue Anforderungen als wichtiger erachtet werden).
- **Should have:** Anforderungen, die als *Should have* (sollte vorhanden sein) gekennzeichnet sind, sind wichtig, aber für die aktuelle Auslieferungs-Timebox nicht notwendig. *Should-have*-Anforderungen können zwar ebenso wichtig wie *Must-have*-Anforderungen sein, sind aber häufig nicht so zeitkritisch, oder es gibt eine andere Möglichkeit, die Anforderung zu erfüllen, damit sie bis zu einer zukünftigen Auslieferungs-Timebox zurückgehalten werden kann.
- **Could have:** Anforderung mit *Could have* (kann vorhanden sein) gekennzeichnet sind wünschenswert, aber nicht notwendig; sie könnten zu niedrigen Entwicklungskosten die User-Experience oder die Zufriedenheit von Benutzern verbessern. Diese werden üblicherweise berücksichtigt, wenn Zeit und Ressourcen es zulassen.
- **Won't have** (dieses Mal): Anforderungen, die als *Won't have* (wird nicht vorhanden sein) gekennzeichnet sind, wurden im Einverständnis der Stakeholder als am wenigsten kritisch eingestuft, als Items mit der geringsten Rendite oder als zu diesem Zeitpunkt nicht geeignet. *Won't-have*-Anforderungen werden dementsprechend nicht in den Zeitplan für die nächste Auslieferungs-Timebox aufgenommen. *Won't-have*-Anforderungen werden entweder fallen gelassen oder in einer späteren Auslieferungs-Timebox erneut für die Aufnahme berücksichtigt.

Als einfacheres Schema, um Prioritäten auszudrücken, können drei Kategorien dienen (anstatt der vier von MoSCoW), und zwar H(igh), M(edium) und L(ow) (hoch, mittel, niedrig) oder alternativ A, B und C.

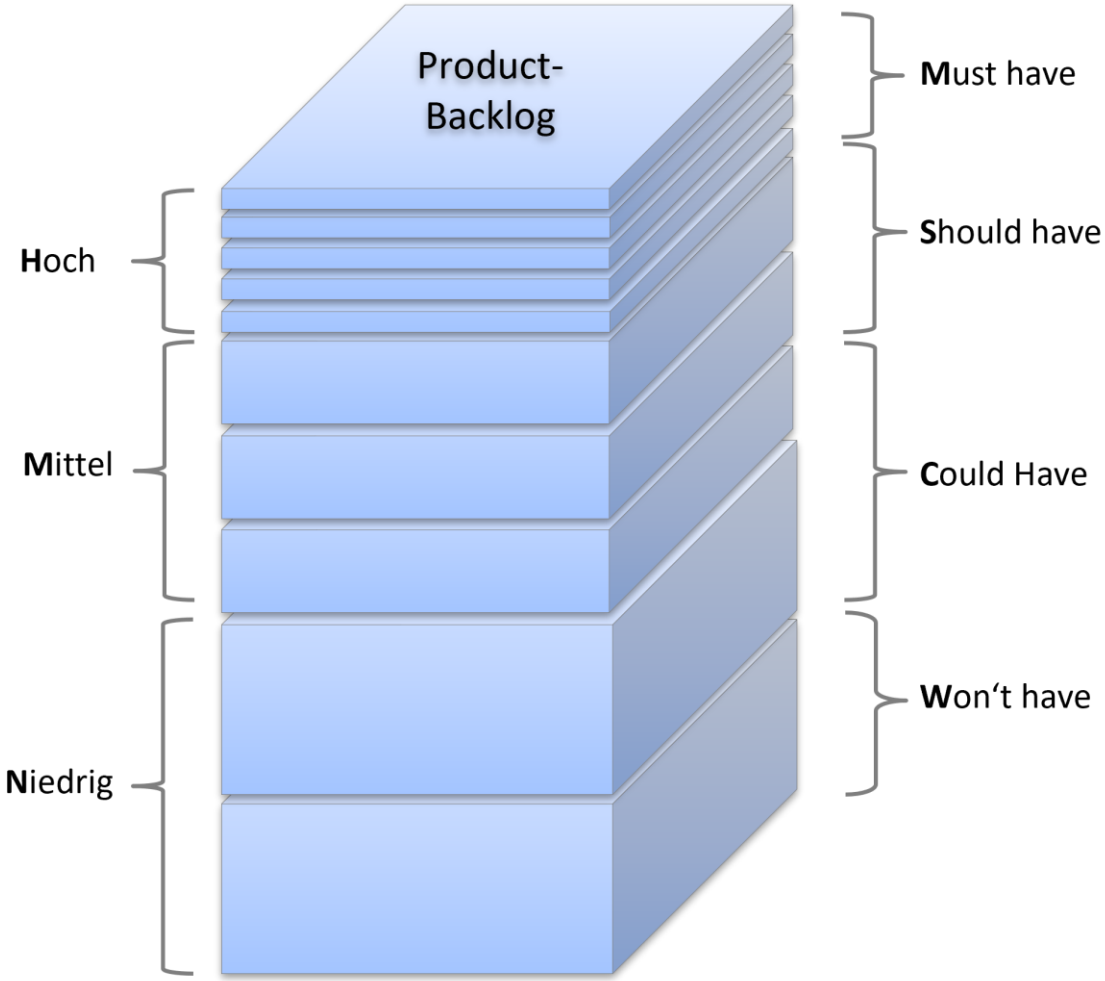

Abbildung 20: Die Prioritäten MoSCoW oder Hoch/Mittel/Niedrig

Abbildung 20 zeigt ein Backlog, bei dem die Items die Annotationen Hoch, Mittel und Niedrig oder MoSCoW aufweisen. Beachten Sie Folgendes: Je höher der Wert ist, der einer Anforderung zugewiesen wird, umso detaillierter sollte diese bereits beschrieben sein, da es sich um einen potenziellen Kandidaten für die nächste (oder eine der nächsten) Iteration(en) handelt.

Einige Unternehmen verwenden eine Bandbreite von Zahlen zwischen 1 und 100 und interpretieren diese so, dass eine höhere Zahl einen größeren Geschäftswert bedeutet. Auf diese Weise können Sie größere Differenzen ausdrücken, etwa indem Sie einem Backlog Item die Priorität 87 zuweisen und einem anderen 38, was eindeutig angibt, wie viel wichtiger das Item mit der Priorität 87 ist.

Abbildung 21 zeigt eine Bandbreite von Zahlen, die kleineren oder größeren Backlog Items zugewiesen wurden. Beachten Sie Folgendes: Wenn ein mittelgroßes Item den Wert 95 hat oder ein größeres Epic den Wert 76, wie in der Abbildung unten, dann ist das eine klare Botschaft für den Product Owner, mit der Arbeit an diesem Item zu beginnen, um es zur Definition of Ready zu bringen, damit solche wichtigen Items in einer baldigen Iteration behandelt werden können.

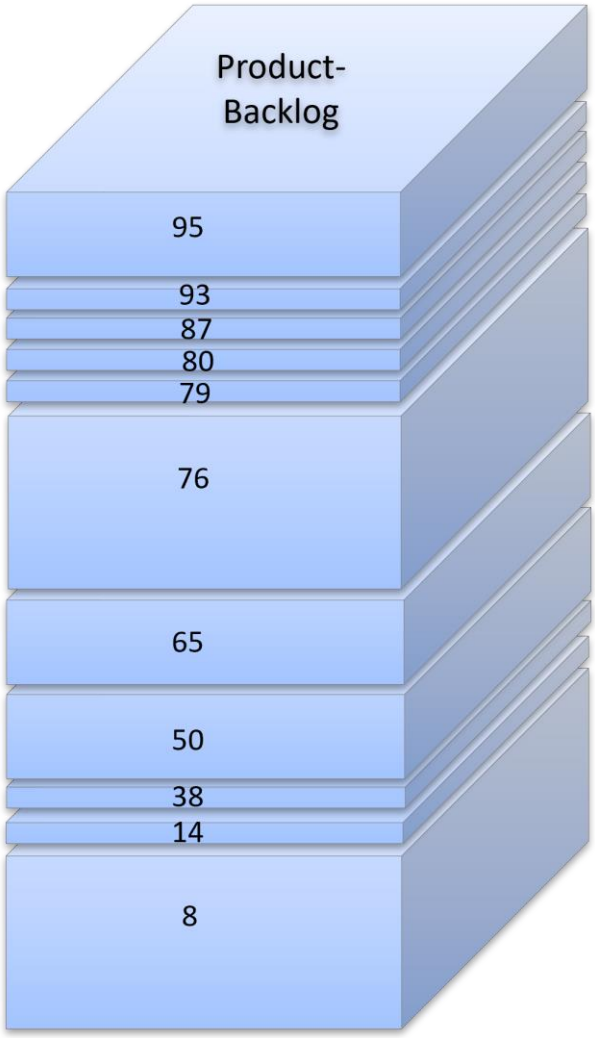

Abbildung 21: Verwendung einer Bandbreite von Zahlen zur Anzeige des Geschäftswertes

Die einfachste Möglichkeit wäre, alle Backlog Items in einer linearen Abfolge zu sortieren (das heißt, Story-Cards von links nach rechts in einer Reihe zu sortieren). Je weiter links sich ein Backlog Item befindet, umso wichtiger wird es eingeschätzt. Je weiter rechts sie es positionieren, umso weniger wichtig wird es eingeschätzt. Dies wird in Abbildung 22 verdeutlicht.

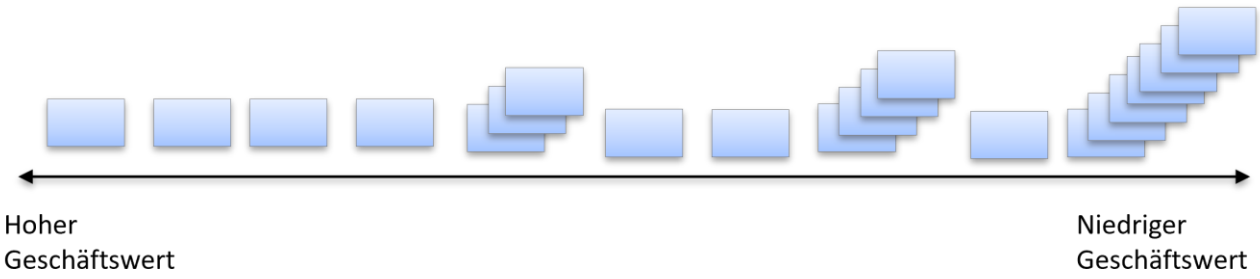

Abbildung 22: Lineare Sortierung nach Clustern des Geschäftswerts

Beachten Sie, dass nur die Items ganz links eine klare lineare Reihenfolge aufweisen, da die Entwickler sie für die nächste Iteration aussuchen. Je weiter rechts ein Item positioniert wird, umso weniger wichtig ist seine exakte Position. Sie können Item-Cluster also stapeln, ohne eine explizite Entscheidung zu ihrem genauen Wert zu treffen.

Der Product Owner hat Zeit für Verfeinerungen, bevor die Items für die Implementierung ausgewählt werden. Nehmen Sie die Sortierung von links nach rechts schnell vor und konzentrieren Sie sich dabei nur auf die Items, die einen hohen Geschäftswert versprechen.

Natürlich können Sie auch viel komplexere Algorithmen zur Wertermittlung verwenden. Sie können sich beispielsweise einige der in Kapitel 5.1 erwähnten Kriterien heraussuchen und diesen jeweils eine Gewichtung zuweisen, um deren Werte zueinander ins Verhältnis zu setzen. Anschließend können Sie die Product Backlog Items innerhalb der einzelnen Kriterien einstufen und den resultierenden Wert errechnen. Abbildung 23 zeigt dies anhand von drei Kriterien und einem Ranking von 0 bis 5 innerhalb der einzelnen Kriterien. Wie Sie sehen, erhalten wir mit diesem kombinierten Ansatz aus Umsatz, Risiko und Benutzbarkeit für Story 3 das werthaltigste Ergebnis.

Backlog Items	Umsatz im 2. Quartal		Minimierung des technischen Risikos		Erhöhung der Benutzbarkeit		Gesamt-ergebnis
	Gewicht = 5	Umsatz Wert	Gewicht = 4	Risiko Wert	Gewicht = 2	Benutzbarkeit Wert	
Anforderung 1	3	15	0	0	0	0	**15**
Anforderung 2	0	0	3	12	1	2	**14**
Anforderung 3	4	20	2	8	2	4	**32**
Anforderung 4	2	10	2	8	3	6	**24**
Anforderung 5	0	0	2	8	5	10	**18**
…							
…							

Abbildung 23: Berechneter Geschäftswert auf Basis mehrerer Kriterien

5.4 Schätzen von User-Storys und anderen Backlog Items

Für Product Owner dient dieses Kapitel nur zu Informationszwecken. Sie sind, wie im letzten Kapitel besprochen, nur für die Ermittlung der Reihenfolge der Backlog Items auf Basis von Wert und Risiko verantwortlich. Es ist die Aufgabe der Entwickler, Schätzungen für die einzelnen Backlog Items zu erarbeiten. Product Owner sollten den Schätzungsprozess nicht beeinflussen, sie sollten nur die Ergebnisse kennen.

Selbst in einer perfekten agilen Welt sind Prognosen nützlich und wertvoll (bei richtiger Anwendung), um zu ermitteln, wie viel Arbeit innerhalb einer zuvor festgelegten Iteration (Timebox) erledigt („done“) werden kann. Aus zwei Gründen dürfen keine Elemente ohne Schätzung in einen Sprint in Scrum aufgenommen werden [Cohn2006]:

1. Es ist unklar, ob das Element innerhalb des Sprints abgeschlossen werden kann. Folglich funktioniert die Software am Ende des Sprints möglicherweise nicht.
2. Ohne Diskussionen und Schätzungen hat ein Team keinen Referenzpunkt (Planung vs. tatsächliche Ausführung), um daraus für die folgenden Sprints zu lernen.

Die meisten Menschen haben eine Abneigung gegen Schätzungen. Viele nicht-agile Organisationen haben unpräzise Schätzungen später in der Regel gegen ihre Mitarbeiter verwendet. War eine Schätzung zu hoch, gilt man möglicherweise als zu vorsichtig oder ängstlich. War die Schätzung dagegen zu gering, musste man sich später womöglich rechtfertigen, dass man den realen Aufwand für die zu erledigende Arbeit nicht erkannt hat.

Agile Organisationen versuchen, diese Abneigung zu überwinden, indem sie eine andere Art von Schätzkultur etablieren. Eine Kultur, die dazu beiträgt, Schuldzuweisungen zu vermeiden. Die Prinzipien dieser Kultur werden in diesem Kapitel behandelt.

Der Grund für bessere Schätzungen sind vor allen Dingen kurze Iterationen in der agilen Entwicklung. Es ist wesentlich einfacher, präzisere Schätzungen für die nächsten zwei bis vier Wochen abzugeben, verglichen mit Schätzungen für Quartale oder Jahre.

Natürlich sind Entwicklungsorganisationen, die mit mehreren Teams an großen Projekten arbeiten, auch auf Prognosen angewiesen, um die Arbeit richtig zu priorisieren und zu planen. Umfangreiche Schätzungen und Planungen werden detaillierter in Kapitel 6 behandelt. In diesem Kapitel konzentrieren wir uns auf die kurzfristige Schätzung, beispielsweise für die nächsten Iterationen.

Agile Methoden empfehlen einige bewährte Verfahren, die bessere und präzisere Schätzungen ermöglichen:

1. Alle am Schätzprozess Beteiligten müssen dasselbe Verständnis der zu erledigenden Arbeit („done“) haben. Dies wird erreicht, indem man die Entwickler in die Backlog-Verfeinerung (z.B. mittels Refinement Meetings) einbezieht. Die Entwickler unterstützen den Product Owner bei der Verfeinerung noch unklarer Epic-Features und Storys oder jeder Art von Anforderung auf diesen Granularitätsstufen. Dabei gewinnen sie einen tieferen Einblick in die zu erledigende Arbeit. Durch die Schaffung eines gemeinsamen Verständnisses davon, was zu erledigen („done“) in diesem Kontext wirklich bedeutet, werden typische Stolpersteine bei der Schätzung vermieden (das Vergessen von Aufwänden für die Dokumentation, Tests oder die Vorbereitung der Auslieferung).
2. Schätzungen werden von denjenigen erledigt, die die Arbeit ausführen, also von cross-funktionalen Entwicklern. Durch den Austausch von Wissen und die Weitergabe von Annahmen über die zu erledigende Arbeit werden alle beteiligten Personen auf denselben Wissensstand gebracht. Sie müssen natürlich jeweils einen gewissen Kompromiss eingehen, wenn Sie entweder alle Teammitglieder in den Schätzprozess einbeziehen oder nur einige. Beziehen Sie alle mit ein, dann sind alle Teil des Prozesses und fühlen sich dem Ergebnis verpflichtet. Dies kann jedoch viel Zeit in Anspruch nehmen, die andernfalls auf die Entwicklung von Features verwendet werden könnte. Wenn Sie nur einige Entwickler in den Schätzungsprozess einbeziehen, fühlen sich die anderen möglicherweise nicht verpflichtet. Es empfiehlt sich, das gesamte Team einzuladen und dann das Team entscheiden zu lassen, wer wirklich für die Schätzung gebraucht wird. In jedem Fall sollte die Schätzung von Gruppen und nicht von Einzelpersonen vorgenommen werden. Später in diesem Kapitel stellen wir Techniken zur Beschleunigung der Schätzung vor.
3. Die Schätzung sollte im Verhältnis zur bereits erledigten Arbeit vorgenommen werden, oder zu Beginn eines Projekts mit kleineren Aufgaben verglichen werden, auf die sich alle Beteiligten einigen können. Die Schätzung nach Analogie oder Neigung ist wahrscheinlich präziser als eine absolute Schätzung. Bei Betrachtung von Abbildung 24 lässt sich leicht feststellen, dass der rechte Stein mehr als doppelt so groß ist wie der linke Stein. Es wäre wesentlich schwieriger, die exakte Größe oder das Gewicht der beiden zu schätzen. Für die Planung sind relative Schätzungen ausreichend präzise.

Abbildung 24: Relative Schätzungen

4. Schätzungen sollten mithilfe einer künstlichen Einheit erfolgen (meist als Story-Points bezeichnet), die Aufwand, Komplexität und Risiko vereint. Die Verwendung einer künstlichen Einheit wie Story-Points ist notwendig, damit jeder mit der neuen Art und Weise der Schätzung und der damit verbundenen Kultur vertraut wird und vom herkömmlichen Verhalten Abstand nimmt.

Für relative Schätzungen stehen mehrere Techniken zur Verfügung. Die bekanntesten Techniken sind das T-Shirt Sizing oder das sogenannte Planning Poker [Cohn2006].

Bei all diesen Techniken ist es hilfreich, zunächst ein Referenz-Item (oder eine Referenz-Story) zu vereinbaren. Nehmen wir einmal an, dass der Apfel in Abbildung 25 die gewählte Referenz ist. Nun können Sie die Größe aller anderen Früchte im Vergleich zu diesem Apfel schätzen. Haben sie annähernd dieselbe Größe? Sind sie viel kleiner? Oder viel größer?

Relative Schätzungen nehmen den Entwicklern die Angst, dass sie genau sein müssen.

Die Größenkennzeichnungen für T-Shirts reichen von „extra small“ bis „extra large“ (XXS, XS, S, M, L, XL, XXL). Einige Methoden empfehlen, dass man bei der Schätzung nicht alle Größenkennzeichnungen verwenden soll, da sie bereits zu präzise sein könnten. Stellen Sie sich die Teilmenge XS, L und XXL vor, wie in Abbildung 25 dargestellt. Selbstverständlich ist eine Kirsche größer als eine Heidelbeere, aber beide sind zweifellos kleiner als Äpfel oder Orangen. Und Melonen sind definitiv größer als Orangen, die eine ähnliche Größe wie Äpfel haben.

Abbildung 25. Reduziertes T-Shirt Sizing

Beim Planning Poker schätzen die Entwickler die Backlog Items basierend auf einer Reihe von Karten mit Zahlen, die an die Fibonacci-Folge angelehnt sind und die ein relatives Größenverhältnis darstellen (vgl. Abbildung 26).

Wenn Sie sich auf eine mittelgroße Referenz-User-Story geeinigt haben, beispielsweise fünf Story-Points, dann entscheidet das Team, welche Größe andere Backlog Items in Bezug auf die Referenz-Story haben. Nachdem alle verdeckt eine Pokerkarte gezogen haben, sieht sich jeder die Werte an: Liegen drei Mal eine „5“ und zwei Mal eine „3“ auf dem Tisch, dann wird das Item als „5“ gekennzeichnet usw. Weichen die Zahlen voneinander ab, dann diskutieren die Teammitglieder mit der niedrigsten und der höchsten Karte die Beweggründe für ihre Schätzungen und versuchen, die anderen Teammitglieder zu überzeugen. Anschließend wird die nächste Schätzungsrunde gestartet. Kann sich das Team nicht innerhalb von drei Runden auf einen gemeinsamen Wert einigen, wird die Anforderung zur Klärung an den Product Owner zurückgegeben.

Für die anstehenden Iterationen möchten Sie sich vielleicht in dem Bereich zwischen 1 und 13 bewegen. Eine „20“, „40“ oder „100“ ist ein Indikator dafür, dass der Product Owner das Item verfeinern muss. Diese Zahlen bedeuten nicht buchstäblich „20“, „40“ oder „100“, sondern „zu groß“, „viel zu groß“ und „enorm“ – aber sie zeigen im Vergleich zu den Items zwischen 1 und 13 zumindest an, um „wie viel zu groß“ die Schätzung ist. Hat das Team kein Verständnis von einem Wert, dann sollte es „?“ auswählen, anstatt seine Angst durch die Wahl von „100“ auszudrücken.

Einige Sets enthalten die „0“, was gewöhnlich bedeutet: „Stoppt die Diskussion, dies ist kein relevanter Aufwand und er ist es nicht wert, in den Plan aufgenommen zu werden.“

Abbildung 26: Planning-Poker-Karten

Der Vorteil der Planning-Poker-Technik ist, dass sie sich sehr gut für die Schätzung durch neue und unerfahrene Teams eignet, da ein zu starker Einfluss (das sog. „Anchoring“) durch einzelne Teammitglieder verhindert wird. Nachteilig ist, dass das Planning Poker sehr zeitaufwendig ist. *[Hinweis: Das Buch „Schnelles Denken, langsames Denken“ von D. Kahneman* [Kahneman2016] *bietet eine sehr gute Einführung in die unbewusste Beeinflussung („Anchoring“) und andere psychologische Effekte in Bezug auf Denken und Urteilsvermögen.]*

Das T-Shirt Sizing oder Planning Poker benötigt in der Regel einige Zeit, da jedes Backlog Item einzeln diskutiert und geschätzt wird. Um diesen Nachteil auszugleichen, können erfahrenere Teams verbesserte Techniken anwenden.

Eine Vereinfachung der Planning-Poker-Technik beruht auf denselben Prinzipien wie das Planning Poker, dabei kommt jedoch eine andere Ermittlungsweise der richtigen Schätzung zum Einsatz.

Anstatt dass jedes Teammitglied eine eigene Schätzung erstellt, wird auf einem Tisch ein Poker-Kartenset verteilt, und die Referenzanforderungen werden in dem entsprechenden „Container“ platziert, der durch die Pokerkarte repräsentiert wird. Anschließend werden die Anforderungen von den Teammitgliedern nach einem Round-Robin-Ansatz ausgewählt, wobei jedes Teammitglied die Möglichkeit hat, eine neue Anforderung in dem entsprechenden „Container“ zu platzieren oder eine bereits platzierte Anforderung einem anderen Container zuzuordnen. Wird eine Anforderung einige Male neu zugewiesen, so wird sie entfernt und an den Product Owner zurückgesendet.

Dieser Ansatz ist wesentlich zeitsparender, aber die Voraussetzung dafür ist ein Team, das erfahren genug ist, um Zuordnungen von anderen Teammitgliedern abzulehnen, anstatt diesen einfach zuzustimmen („Anchoring“).

Der nächste Entwicklungsschritt wird gewöhnlich als „Affinitätsschätzung“ (Affinity Estimation) oder „Wall Estimation“ bezeichnet. Er wird zur Schätzung einer größeren Anzahl von Anforderungen verwendet, etwa für Grobschätzungen als Vorbereitung für Release-Planungen. Anders als beim vorherigen Ansatz werden hierbei die Anforderungen nicht per Round-Robin-Ansatz zugeordnet, sondern jedes Teammitglied erhält einige Anforderungen und ordnet diese stillschweigend den über das Poker-Kartenset dargestellten „Containern“ zu (vgl. Abbildung 27). Nach der stillschweigenden Zuordnung dürfen alle Beteiligten die zugeordneten Anforderungen prüfen und fragliche Zuordnungen kennzeichnen. Dies führt in der Regel zu einer Quote von 20 bis 30 % von Anforderungen, für die Diskussionsbedarf besteht, und 70 bis 80 % von Anforderungen, die von allen Teammitgliedern akzeptiert werden.

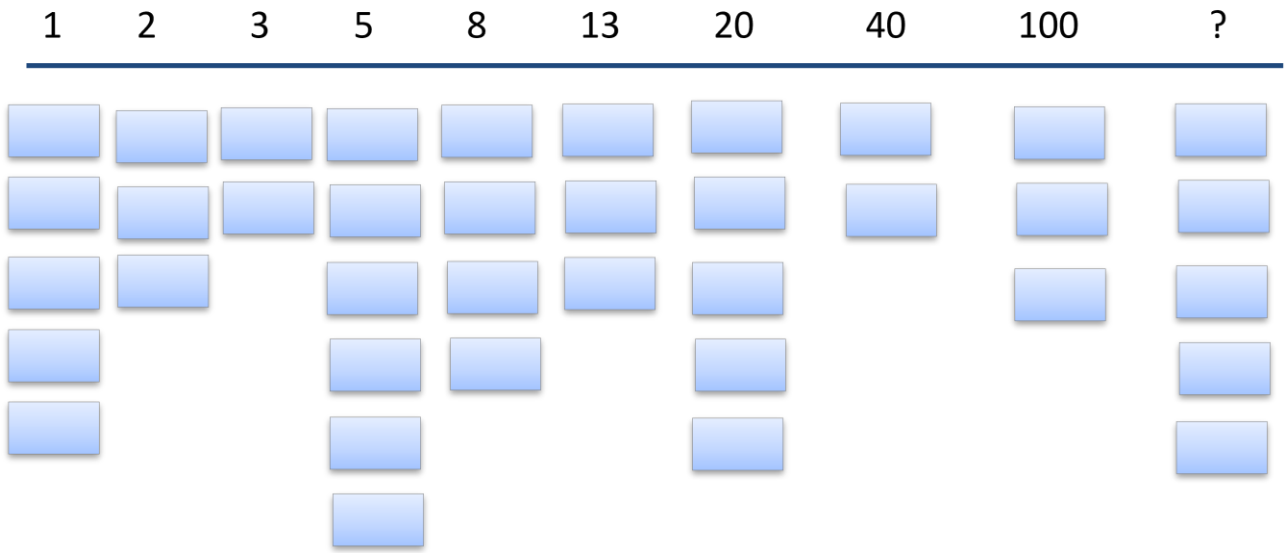

Abbildung 27: „Wall Estimation“ oder Affinitätsschätzung

Nun noch ein paar abschließende Anmerkungen zu Schätzungen:

Der Schätzungsprozess innerhalb eines Teams ändert sich im Laufe der Zeit. Das Team lernt aufgrund der Ergebnisse abgeschlossener Iterationen hinzu und wird präzisere Regeln in seine Definition of Done aufnehmen.

Relative Schätzungen haben zahlreiche Vorteile und funktionieren gut innerhalb eines Teams (wie zuvor angesprochen). In Bezug auf teamübergreifende Schätzungen haben sie jedoch auch einige Nachteile. Dies wird in Kapitel 6 (Skalierung) detaillierter behandelt.

Übungsvorschlag:

Suchen Sie sich eine Fallstudie heraus und wenden Sie eine schnelle Möglichkeit zur Schätzung des Umfangs der Backlog Items an. Besprechen Sie Ihre Feststellungen und gehen Sie vor allem durch, welche Aspekte beim Schätzen funktioniert haben und welche nicht.

5.5 Auswählen einer Entwicklungsstrategie

Beim Auswählen der Elemente für frühe Releases können basierend auf bekannten Werten, Risiken und dem für die Entwicklung eines Backlog Items benötigten Aufwand verschiedene Strategien zum Einsatz kommen. Es gibt bei agilen Entwicklungsprozessen zwei typische Konzepte: Die Entwicklung eines Minimum Viable Product (MVP) und die Entwicklung eines Minimum Marketable Product (MMP).

Minimum Viable Product

Mit Minimum Viable Product ist sinngemäß die Version eines neuen Produkts gemeint, die einem Team das Erfassen des maximal möglichen validierten Lernens über Kunden bei geringstem Aufwand ermöglicht. Der Begriff wurde ursprünglich im Jahr 2001 von Frank Robinson geprägt und von Steve Blank und Eric Ries [Ries2011] popularisiert.

Erkenntnisse durch ein MVP zu sammeln, ist meist weniger kostspielig, als ein Produkt mit mehr Features zu entwickeln. Die Entwicklung eines Produkts mit mehr Features verursacht höhere Kosten und beinhaltet mehr Risiken, wenn das Produkt etwa aufgrund von falschen Annahmen misslingt.

Das MVP ist eine zentrale Idee der von Eric Ries entwickelten Lean-Startup-Methodologie, die auf dem „Build-Measure-Learn"-Zyklus (bauen, messen, lernen) beruht (siehe Abbildung 28).

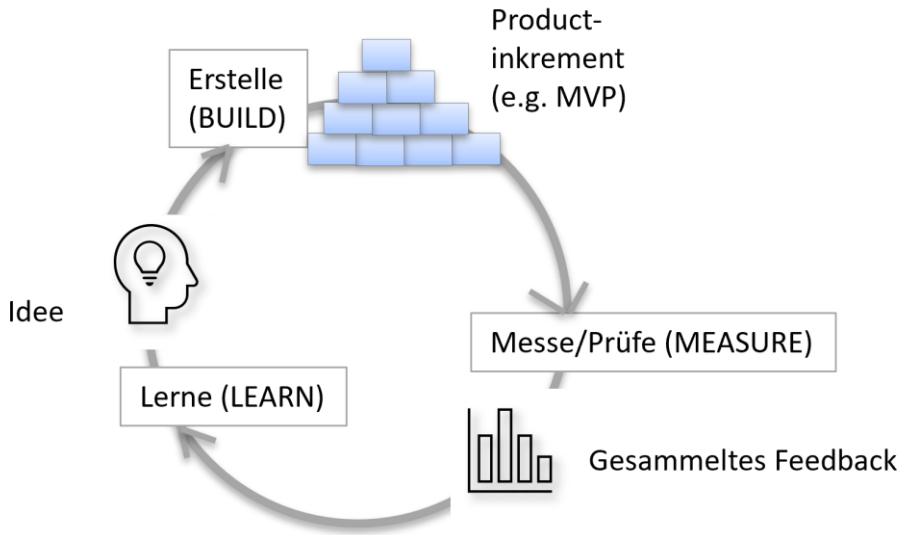

Abbildung 28: Der „Build-Measure-Learn"-Zyklus des Lean Development

Ein MVP ist also ein Vehikel zum Lernen, das Ihnen das Testen einer Idee ermöglicht. Sie können den gewünschten Stakeholdern damit schnell etwas Greifbares anbieten und haben die Möglichkeit, Daten zu sammeln und daraus Erkenntnisse über Ihren Zielmarkt abzuleiten.

Roman Pichler [Pichler2016] merkt Folgendes an: „Das MVP wird als minimum bezeichnet, da Sie möglichst wenig Zeit und Aufwand für dessen Erstellung investieren sollten. Das heißt jedoch nicht, dass es „quick and dirty" sein muss. Wie lange die Erstellung eines MVP dauert und wie viele Features es enthalten soll, hängt von Ihrem Produkt und dem Markt ab. Sie sollten aber versuchen, das Feature-Set möglichst klein zu halten, um den Lernprozess zu beschleunigen und keine Zeit und kein Geld zu verschwenden, da sich Ihre Idee möglicherweise als falsch herausstellen kann!"

Das MVP ist nicht notwendigerweise ein einsatzbares Softwareprodukt. Mitunter können Prototypen aus Papier und anklickbare Mock-ups verwendet werden, um Erkenntnisse abzuleiten, sofern diese dazu beitragen, die Idee zu testen und relevante Kenntnisse zu gewinnen.

Ein MVP für das System iLearnRE könnte einfach aus der Veröffentlichung eines Einführungs- und Zusammenfassungsvideos für die einzelnen Lernziele bestehen, um Erkenntnisse zum Benutzerverhalten und zur UI-Akzeptanz zu gewinnen.

Minimum Marketable Product

Der nächste Schritt sollte die Erstellung eines Minimum Marketable Product (MMP) sein. Es beruht auf der Idee „weniger ist mehr“: Das MMP beschreibt das Produkt mit dem kleinstmöglichen Feature-Set, das auf die Bedürfnisse der ersten Benutzer (Innovatoren und frühe Anwender) eingeht und somit vermarktet werden kann. Studien zufolge enthalten die meisten Softwareprodukte viele Features, die niemals oder nur sehr selten genutzt werden.

Daher erscheint es vernünftig, dass Sie sich auf die Features konzentrieren, die von der Mehrheit Ihrer Stakeholder geschätzt werden, und weniger gängige Features zurückstellen. Die Ermittlung dieser Features ist nicht gerade einfach, aber MVPs sind eine ausgezeichnete Möglichkeit, um dieses Ziel zu erreichen. Einige Ihrer MVPs sind möglicherweise Wergwerf-Prototypen, die lediglich für Lernzwecke erstellt wurden. Wenn Sie es jedoch richtig angehen, entwickeln Sie diese so, dass sie wiederverwendet oder in das erste MMP verwandelt werden können.

Durch die Kombination dieser beiden Konzepte haben Sie eine Strategie, wie sie in Abbildung 29 dargestellt ist. Entwickeln Sie ein paar MVPs, um den Markt zu testen und reale Daten als Feedback zu erhalten. Entscheiden Sie dann, welche Mindestanzahl von Features ein Produkt aufweisen muss, damit es zumindest für eine Kerngruppe Ihrer Stakeholder nützlich ist. Fügen Sie anschließend kontinuierlich Features hinzu, die einen größeren Geschäftswert versprechen.

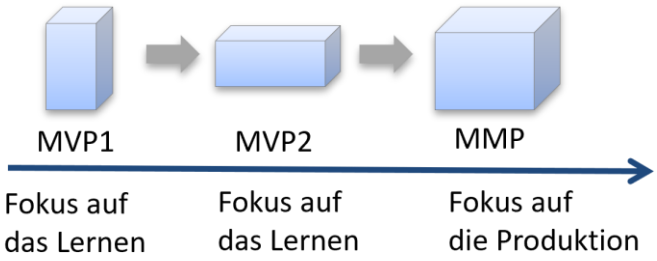

Abbildung 29: Kombinieren von MVP und MMP

Risikoreduktion

Die Entwicklung von MVPs hängt sehr eng mit der Idee einer Strategie zur Risikominderung zusammen. MVPs werden meist entwickelt, um das Risiko falscher Features für die Stakeholder zu verringern. Sie können aber auch MVPs (oder Spikes) erstellen, um technische Risiken zu reduzieren. Es ist besser, schnell zu scheitern (entweder mit der Funktionalität oder mit der Technologie), als ein vollwertiges Produkt zu entwickeln und dann festzustellen, dass es am Markt nicht erfolgreich ist.

In unserer Fallstudie iLearnRE kann eine realisierbare frühe Version beispielsweise ein Leistungstest der geplanten Videoplattform bei hoher Auslastung sein.

„Low Hanging Fruit" oder „Quick Wins"

Das Gegenteil einer risikoorientierten Strategie ist es, wenn man zuerst nach niedrig hängenden Früchten ("low hanging fruits") greift. Beginnen Sie mit der Veröffentlichung von Features, die einfach und schnell zu implementieren sind. So können Sie frühzeitig Umsatz generieren und Geld verdienen, was es Ihnen erlaubt, in komplexere Features zu investieren. Nehmen Sie sich aber davor in Acht, risikoreiche Teile zu verschieben, da diese die Architektur eines Produkts, das auf niedrig hängenden Früchten („low hanging fruits") beruht, ruinieren können.

Die Warnung von Professor Kano

Professor Kano führte Studien über Kundenzufriedenheit in Bezug auf ausgelieferte Features durch. Im CPRE Foundation Level Lehrplan haben wir drei Kategorien von Anforderungen vorgestellt, die Sie nun unterscheiden können: Basisfaktoren, Leistungsfaktoren und Begeisterungsfaktoren.

Kano macht darauf aufmerksam, dass jedes erfolgreiche Release eines Produkt-Features aller drei Kategorien enthalten sollte. Wenn Sie konstant nur Basisfaktoren bereitstellen, werden Ihre Kunden nicht sehr glücklich sein. Sie müssen einige Leistungsfaktoren einbeziehen, beispielsweise Features, um die Kunden ausdrücklich gebeten haben, selbst wenn diese nicht absolut notwendig sind.
Sie sollten versuchen innovativ zu sein, indem Sie Features in ihr Produkt aufnehmen, die ihre Kunden bei Erhalt begeistern, auch wenn sie diese Features nicht angefragt haben.

Es ist schwierig, für jedes Release eine solche Mischung aus Features zusammenzustellen. Aus diesem Grund sollten Sie, wie oben erwähnt, kontinuierlich Ihre Märkte anhand von MVPs testen und reale Daten sammeln, bevor Sie zur zeitaufwendigen und teuren Feature-Entwicklung übergehen.

Weighted Shortest Job First (WSJF)

Eine andere interessante Entwicklungsstrategie ist „Weighted Shortest Job First“ (WSJF, „Wert/Aufwand-Verhältnis“). Sie beruht auf dem Verhältnis aus den Kosten für Verzögerungen und dem für die Entwicklung geschätzten Aufwand [Reinertsen2008].

$$WSJF = \frac{Cost\ of\ Delay}{Duration}$$

Die Kosten für Verzögerungen sind wesentlich höher als der Nutzen (Geschäftswert), wenn die entsprechende Anforderung entwickelt wird. Darin eingerechnet ist auch der Fall, dass die entsprechende Anforderung nicht entwickelt wird (beispielsweise Verlust des Marktanteils, Vertragsstrafen), wenn die Entwicklung dieser Anforderung das Risiko für die gesamte Implementierung verringert (Proof of Concept) oder wenn sie eine neue Chance eröffnet (z. B. die Verwendung von Frameworks, die den zukünftigen Entwicklungsaufwand senken).

Der WSJF-Ansatz kann dabei helfen, die Anforderungen (oder Teile davon) zu bestimmen, welche zuerst, ohne genaue Kenntnis aller Details, entwickelt werden sollten, und zwar einfach durch Nutzen der Beziehung zwischen Anforderungen der Cost of Delay und Dauer (Entwicklungsaufwand).

Backlog Anforderung	Geschäfts-wert		Zeitliche Kritikalität		RR / OE		CoD		Dauer		WSJF
Anforderung 1	8	+	5	+	13	=	26	/	2	=	**13**
Anforderung 2	1	+	13	+	1	=	15	/	3	=	**5,0**
Anforderung 3	8	+	1	+	8	=	17	/	2	=	**8,5**
Anforderung 4	13	+	8	+	5	=	26	/	1	=	**26**
Anforderung 5	5	+	2	+	2	=	9	/	3	=	**3,0**

Abbildung 30: Beispiel für WSJF

Die Tabelle ist folgendermaßen aufgebaut:

- Füllen Sie die Spalte mit den Items/Anforderungen aus, die eingestuft werden sollen (in unserem Beispiel die Items 1 bis 5).

- Füllen Sie die Spalten (außer CoD) nacheinander von links nach rechts aus:
  - Geschäftswert – welcher Wert entsteht, wenn das Item entwickelt wird?
  - Zeitliche Kritikalität – welcher Wert geht verloren, wenn das Item nicht entwickelt wird?
  - RR (Risk Reduction, Risikoreduktion) /OE (Opportunity Enablement, Ermöglichen von Chancen) – um wie viel kann das Risiko verringert werden oder wie viele Chancen können ergriffen werden, wenn das Item entwickelt wird?
- Suchen Sie in jeder Spalte das Element, das den GERINGSTEN Wert pro Spalte hat und weisen sie ihm den Wert „1“ zu.
- Stufen Sie alle anderen Items in der Spalte als einen Faktor in Beziehung zu „1“ ein (die Fibonacci-Folge hat sich zwar bewährt, aber Sie können auch beliebige andere Zahlen verwenden).
- Berechnen Sie den Wert der CoD als Summe der vorherigen Spalten.
- Berechnen Sie den Wert von WSJF als Verhältnis aus CoD/Dauer.

Das Item mit dem größten WSJF-Verhältnis sollte zuerst entwickelt werden, gefolgt von dem Item mit dem zweitgrößten Verhältnis usw.

Bei Verwendung dieses Ansatzes werden große Blöcke in der Regel später entwickelt, da diese normalerweise ein geringeres Verhältnis haben. Die Empfehlung für den Product Owner lautet also, große Blöcke aufzuteilen und die Teile zu identifizieren, die einen hohen Wert liefern bzw. einen geringen Aufwand bedeuten, und die weniger werthaltigen Bestandteile aufzuschieben.

5.6 Zusammenfassung

Das Sortieren des Backlogs ist ein iterativer zweistufiger Prozess. Als Product Owner sortieren Sie während des ersten Schritts vorab das Backlog auf Basis des Geschäftswertes. Sie haben verschiedene Wege kennengelernt, um zu definieren, was Geschäftswert in Ihrer Organisation bedeutet. Als Product Owner sollten Sie Risiken nicht unterschätzen. Manchmal müssen Sie Wert gegen das Risiko abgleichen, um Ihre Produktentwicklung nicht zu gefährden. Wert lässt sich mithilfe verschiedener Skalen ausdrücken, beispielsweise MoSCoW oder Hoch, Mittel und Niedrig. Alternativ setzen Sie alle Items basierend auf ihrem Wert in eine lineare Folge. Dann müssen Sie keine Zahlen verwenden.

Im zweiten Schritt müssen die Entwickler Schätzungen zu den einzelnen Backlog Items abgeben. Agilität hat viel dafür getan, den Prozess der Schätzung weniger bedrohlich erscheinen zu lassen:

- Die richtigen Mitarbeiter (diejenigen, die die Arbeit ausführen) geben Schätzungen ab.
- Schätzungen werden in einer Gruppe vorgenommen, nicht von einer einzelnen Person.
- Schätzungen sollten relativ erfolgen; es sollten Größe und Aufwand von Items verglichen werden, anstatt diesen einen absoluten Wert zuzuweisen.

Für Schätzungen können verschiedene Methoden genutzt werden, beispielsweise das T-Shirt Sizing oder die Nutzung von Fibonacci-Karten im Planning Poker. Zur Beschleunigung der Schätzmethode können die Wall Estimation oder die Affinitätsschätzung verwendet werden.

Wenn Backlog Items klein genug sind und gut verstanden wurden, sind die Schätzungen für die Iterationsplanung ausreichend präzise. Sind die Items noch immer zu groß oder wurden sie nicht vollständig verstanden, dann zeigt das Team dies durch einen höheren Wert an. So erfährt der Product Owner, dass für diese Items Klärungs- und/oder Verfeinerungsbedarf besteht.

Sobald die Items geschätzt wurden, kann der Product Owner die Reihenfolge des Backlogs noch einmal ändern, beispielsweise durch den Austausch einer Gruppe von günstigeren Items durch ein teureres Item.

Auf Basis des ermittelten Wertes und der Schätzungen kann eine Reihe von verschiedenen Strategien angewendet werden, um die Reihenfolge zu bestimmen, in der die Items zu Iterationen zugewiesen werden sollen. Strategien, wie das Erstellen einer Reihe von Minimum Viable Products (MVPs), gefolgt von einem Minimum Marketable Product (MMP), bevor weitere Features hinzugefügt werden, unterstützen das Agilitätsprinzip von früh liefern und oft liefern. Gangbare Alternativen sind aber auch das Ernten niedrig hängender Früchte („low hanging fruits") oder das frühzeitige Reduzieren von Risiken.

Besteht beispielsweise das primäre Ziel einer Organisation darin, das Produkt frühzeitig auszuliefern und Marktanteile zu gewinnen, könnte sie eine Strategie wählen, mit der sie rasch einen Geschäftswert erzielt. Wenn ein Lieferant Produktrücksendungen, etwa aufgrund von Leistungs- oder Sicherheitsmängeln, um jeden Preis vermeiden möchte, bevorzugt er womöglich eine Strategie der frühen Risikoreduktion.

6. Skalierung von RE@Agile

Requirements Engineering ist einfacher für Produkte, die so klein sind, dass sie von einem einzigen Team an einem Standort bearbeitet werden können. Alle bisherigen Kapitel gingen implizit von dieser Annahme aus: Wir haben gezeigt, wie die wichtigsten Anforderungen (d. h. diejenigen, die den höchsten Geschäftswert liefern) von diesem Team umgesetzt werden können, ohne dass die Anforderungen auf mehrere (Entwicklungs-)Teams verteilt werden müssen. Wenn diese Annahme nicht mehr zutrifft, d. h. wenn wir mehr als ein Team brauchen, um unsere Geschäftsziele und Visionen zu erreichen, müssen wir über eine Skalierung unserer Entwicklung nachdenken.

In diesem Kapitel wird erörtert, warum die Produktentwicklung manchmal skaliert werden muss und warum Produkte von mehr als einem Team entwickelt werden müssen, sei es am gleichen Standort oder geografisch verteilt. Wenn skaliert wird, ist der Product Owner des Gesamtprodukts (als die für das Anforderungsmanagement verantwortliche Rolle) wahrscheinlich mehr mit Managementaspekten als mit Anforderungsaspekten konfrontiert. Wir werden erörtern, dass die beiden Faktoren *Markteinführungszeit* und *Komplexität* (entweder funktionale Komplexität oder anspruchsvolle Qualitätsanforderungen) den Skalierungsprozess rechtfertigen und vorantreiben. Aber auch organisatorische und technische Randbedingungen werden die Art und Weise der Skalierung beeinflussen.

In diesem Kapitel behandeln wir die folgenden Aspekte:

- Was bedeutet Skalierung und wie wirkt sie sich auf Anforderungen und Teams aus (Kapitel 6.1)?
- Wie (re)organisieren wir die Anforderungen und die Teams *im Großen* (Kapitel 6.2)?
- Wie werden Releases und Roadmaps definiert und in der langfristigen Planung verwendet (Kapitel 6.3)?
- Wie werden Anforderungen in skalierten Umgebungen validiert (Kapitel 6.4)?

6.1 Skalierung von Anforderungen und Teams

Wir verwenden den Begriff *Skalierung*, um eine Veränderung der Größe zu beschreiben, entweder des Systems oder des Produkts oder der Anzahl der beteiligten Personen.

Seit etwa 2010 wurde eine Reihe verschiedener agiler Skalierungs-Frameworks entwickelt, um diese Probleme zu lösen. Dazu gehören Nexus [Nexus Guide], SAFe [SAFe1] [SAFe2], LeSS [LeSS], Scrum@Scale [S@S Guide], BOSSA Nova [BOSSANOVA], Scrum of Scrums [SofS], und Spotify [Spotify2012], aber es gibt noch mehr. Skalierungs-Frameworks variieren hinsichtlich ihres Reifegrades, der Anzahl bewährter Verfahren, Richtlinien und Regeln sowie dem Grad der Anpassbarkeit des Frameworks an bestimmte Bedürfnisse einer Organisation. Wir werden die einzelnen Frameworks nicht im Detail besprechen, sondern sie vielmehr als Beispiele heranziehen, insbesondere dann, wenn sie alternative Ansätze für den Umgang mit Anforderungen im Großen darstellen.

Unter Abbildung 31 werden die treibenden Kräfte für die Skalierung sowie die Beschränkungen, die auf dem Weg dorthin auftreten können, aufgezeigt.

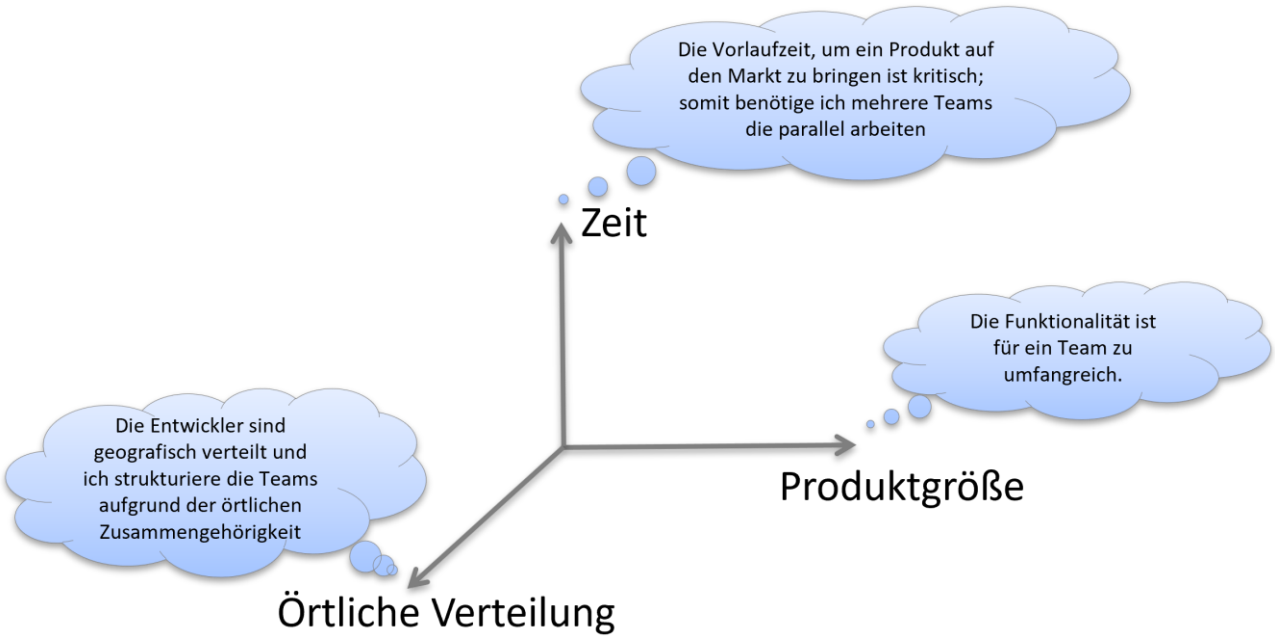

Abbildung 31: Drei Dimensionen, die eine Skalierung auslösen können

Die ersten beiden Dimensionen in der obigen Abbildung sind:

- Markteinführungszeit: Ein einziges Team würde zu lange brauchen, um alle Anforderungen für ein zufriedenstellendes Produkt umzusetzen. Um das Release zu beschleunigen, setzen Sie mehrere Teams ein.
- Komplexität des Produkts: Die Produktdomäne, oder die für die Umsetzung verwendeten Technologien sind so komplex, dass ein Team nicht alle Aspekte behandeln kann. Sie beschließen daher, mit mehreren Teams zu arbeiten, die sich jeweils auf unterschiedliche Aspekte des Produkts konzentrieren.

In beiden Fällen sind Sie sofort mit der Tatsache konfrontiert, dass Sie die Arbeit von mehr als einem Team koordinieren müssen. Dies erschwert die Entwicklung im Vergleich zur Arbeit mit einem einzigen, an einem Ort angesiedelten Team.

In der obigen Abbildung ist eine dritte Dimension dargestellt:

- Möglicherweise müssen Sie aus organisatorischen oder politischen Gründen mit mehreren Teams zusammenarbeiten: Sie haben vielleicht Mitarbeiter an verschiedenen geografischen Standorten oder arbeiten in mehreren Unternehmen, oder die Teams sind nach bestimmten Fachkenntnissen organisiert. Wir betrachten alle diese Aspekte als Randbedingungen, die sich manchmal nicht vermeiden lassen, obwohl wir nicht unbedingt empfehlen würden, diese Organisationsstrukturen zu wählen, wenn sie nicht bereits vorhanden sind. Mehr über gute und schlechte Kriterien für die Teamstrukturierung in Kapitel 6.2.

Seien Sie jedoch vorsichtig mit der Skalierung, wenn sie nicht unbedingt notwendig ist: Die Arbeit mit mehr als einem Team führt immer zu einem erhöhten Kommunikations- und Koordinationsaufwand. Wenn also die oben genannten Gründe für eine Skalierung nicht zutreffen, sollten Sie wahrscheinlich überhaupt nicht skalieren!

Wenn Sie jedoch skalieren, werden zwei Dinge immer zutreffen: Sie werden gezwungen sein, die Anforderungen zu hierarchisieren und die Organisation zu hierarchisieren. Grobgranulare Anforderungen werden benötigt, wenn es um das Produkt als Ganzes geht; feingranulare Anforderungen werden in den Teams benötigt, die bestimmte Aspekte des Produkts umsetzen.

Und die Teams selbst müssen ihre Zusammenarbeit so organisieren, dass sie innerhalb eines größeren Teams erfolgreich funktionieren.

In den folgenden Kapiteln wird erörtert, wie die verschiedenen Skalierungsframeworks diese beiden Aspekte behandeln und welche Terminologie sie für Anforderungs- und Teamhierarchien vorschlagen.

6.1.1 Organisation umfangreicher Anforderungen

In Kapitel 3 haben wir das Thema der Anforderungsgranularität erörtert und die Begriffe *grobgranulare Anforderungen*, *Anforderungen mittlerer Granularität* und *feingranulare Anforderungen* eingeführt. Wir haben uns bewusst für diese allgemeinere Terminologie entschieden, da sich die Skalierungsframeworks (und agilen Anforderungswerkzeuge) in ihren spezifischen Begriffen deutlich unterscheiden.

Die hierarchische Darstellung von Anforderungen spiegelt eine der Schlüsselideen des Product Backlogs wider: Grobgranulare Anforderungen können immer noch vage oder ungenau sein, bis sie (oder Teile davon) für eine kommende Iteration relevant werden und daher mehr Details und Präzision benötigen. Auf diese Weise werden feinere Anforderungen ausgearbeitet, und es wird eine Beziehung zu ihren größeren Eltern beibehalten. Die sich daraus ergebende Hierarchie erfüllt zwei Zwecke:

- Sie bietet einen Überblick über alle bekannten Anforderungen.
- Sie ermöglicht die selektive Ausarbeitung derjenigen Elemente, die am ehesten in Kürze entwickelt werden können.

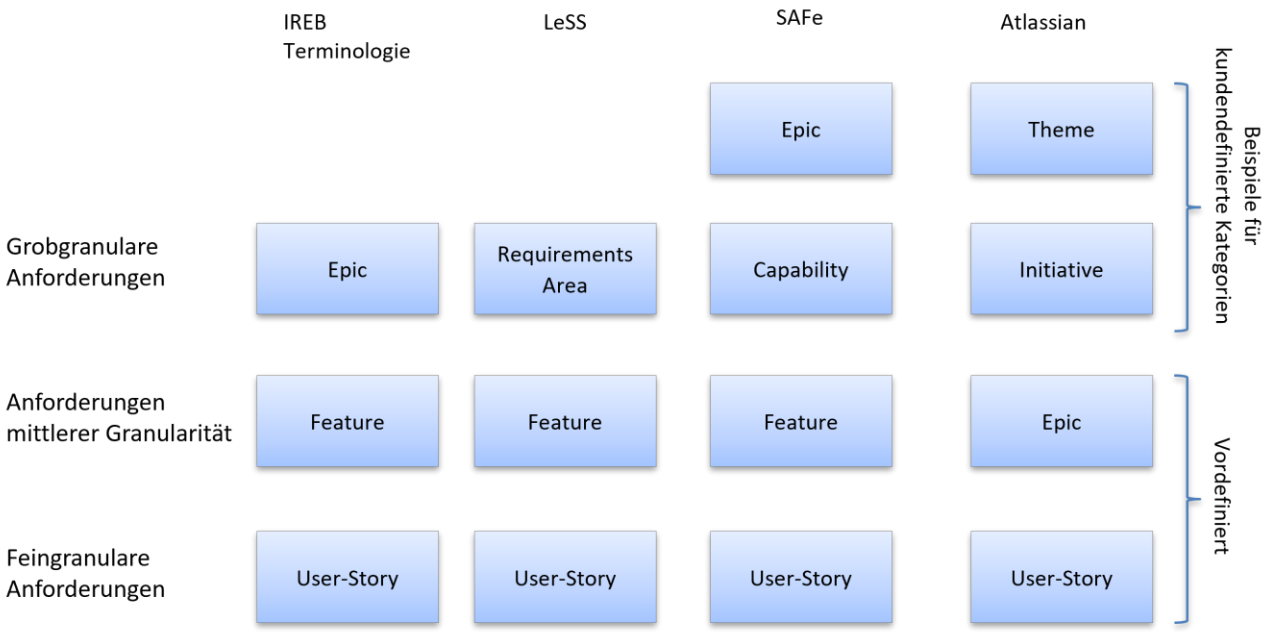

Abbildung 32: Terminologie für Anforderungen auf verschiedenen Granularitätsebenen in ausgewählten Methoden und Werkzeugen

Für die Zwecke dieses Handbuchs hat das IREB eine der populäreren Begriffsgruppen für Anforderungen auf verschiedenen Granularitätsstufen gewählt, die drei Begriffe enthält: Epics (für grobgranulare Anforderungen), Features (mittlere Granularität) und User Storys (feingranular).

Einige Skalierungs-Frameworks und -Werkzeuge geben den einzelnen Anforderungsebenen keine expliziten Namen, sondern bezeichnen sie einfach als *Backlog Items* und erlauben ihre Verfeinerung, bis sie klein genug sind, um in einer einzigen Iteration implementiert zu werden.

Andere Tools beginnen mit einem zweistufigen Ansatz, erlauben dann aber die Erweiterung der Anzahl der Stufen. Jira von Atlassian beispielsweise verwendet standardmäßig Epics und Storys, lässt aber eine Erweiterung dieser Hierarchie zu (in neueren Versionen wird vorgeschlagen, die größten Anforderungen als *Themen* und die nächste Ebene als *Initiativen* zu bezeichnen). LeSS bezeichnet Anforderungen auf der Ebene oberhalb der User Storys als *Features* und auf der größten Ebene als *Requirements Areas*.

Das SAFe-Framework bietet ein umfangreiches Anforderungs-Metamodell [SAFeMDM] mit vier Anforderungsebenen und einem strengen Namensschema: Epics, Capabilities, Features und User Storys. Abbildung 33 zeigt eine vereinfachte Version dieses Metamodells. Die Unterscheidung zwischen den Ebenen erfolgt nicht so sehr nach dem Inhalt, sondern eher nach der Größe.

Eine Story muss so klein sein, dass sie in eine Iteration (oder einen Sprint) passt; ein Feature muss so klein sein, dass es in ein Release passt. Capabilities und Epics sind so umfangreich, dass sie sich über mehr als ein Release erstrecken (mehr zur Release-Planung in Kapitel 6.3).

Beachten Sie, dass SAFe auf jeder Ebene zwischen *Business* Features - diejenigen, die den Geschäftswert schaffen - und *Enabler* Features - den notwendigen architektonischen Voraussetzungen, ohne die der Geschäftswert nicht erreicht werden kann - unterscheidet. Wir werden diese Unterscheidung in Kapitel 6.2.3 näher erläutern.

SAFe verwendet auch spezifische Begriffe für die Akzeptanzkriterien auf verschiedenen Granularitätsebenen, wie unten dargestellt.

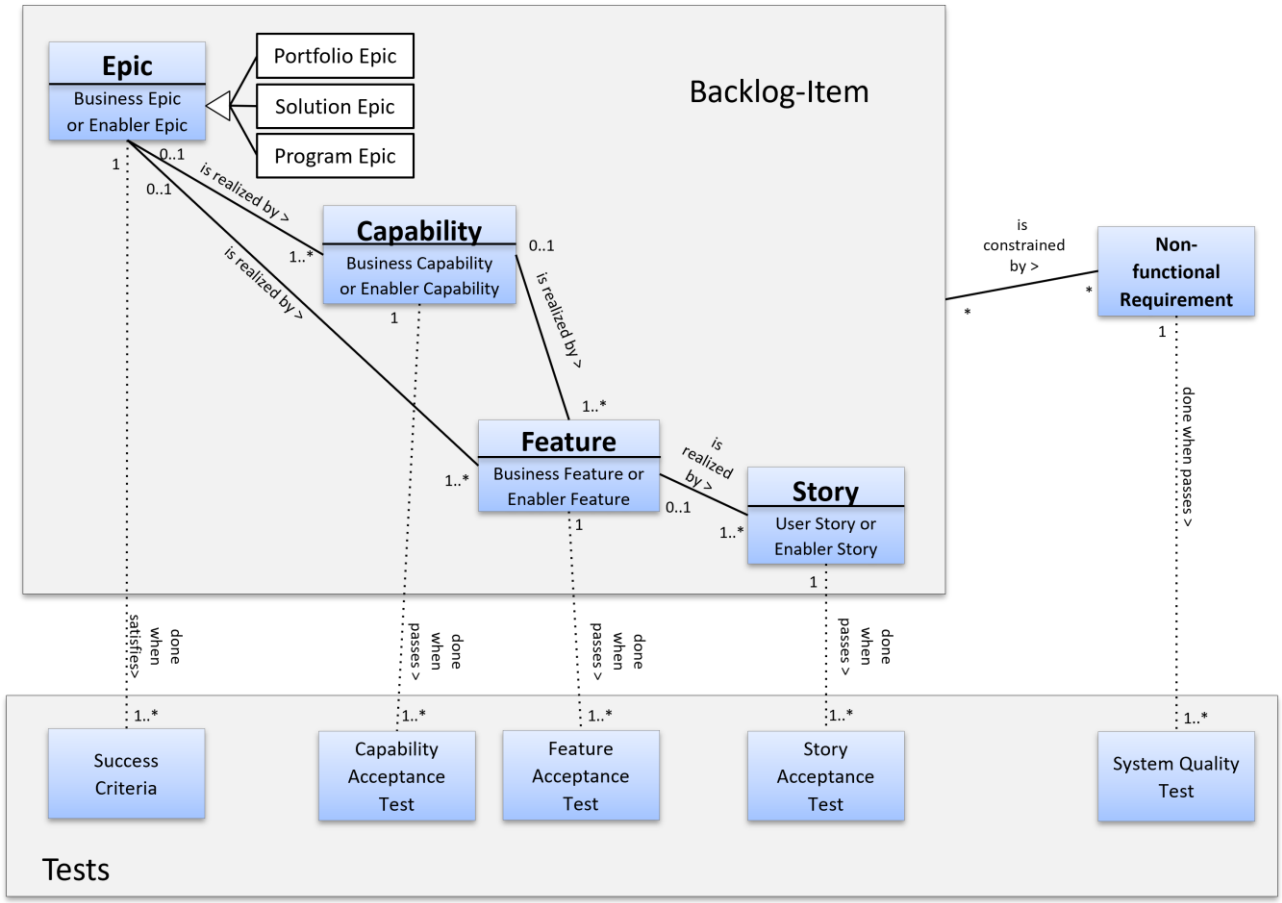

Abbildung 33: SAFe-Terminologie für Anforderungen

Obwohl viele der heutigen agilen Anforderungswerkzeuge nicht in der Lage sind, die vier Granularitätsebenen dieses Metamodells out-of-the-box zu verarbeiten, bieten die meisten von ihnen die Möglichkeit, die Hierarchie anzupassen.

Um langwierige Diskussionen über die Terminologie (und Methodenkriege zwischen Ihren Teams!) zu vermeiden, schlagen wir vor, dass Sie sich auf eine interne Terminologie für die Granularitätsebenen einigen, die Sie verwenden möchten, und sich dann bei jedem Entwicklungsprojekt an diese halten. Sehr oft geben entweder das Skalierungsframework oder die verwendeten Werkzeuge die Terminologie vor.

6.1.2 Organisieren von Teams

Alle Skalierungsframeworks stimmen darin überein, dass …

- ... unabhängig von den spezifischen Job Titeln Verantwortung auf jeder Ebene der Organisation benötigt wird.
- ... die Arbeit zwischen den Teams gut koordiniert werden muss.

Abgesehen von diesen allgemeinen Punkten unterscheiden sich die Konzepte und die Terminologie jedoch in den einzelnen Ansätzen.

Wenn Scrum für mehrere Teams verwendet wird, wird eine Technik zur Koordinierung dieser Teams häufig als *Scrum of Scrums* bezeichnet. [SofS] Der einzige Unterschied zur Arbeit innerhalb eines Teams besteht darin, dass jedes Team eine Person (einen Botschafter) ernennt, die es bei den Koordinierungssitzungen vertritt, die normalerweise zwei- oder dreimal pro Woche stattfinden. Im Laufe eines Projekts kann das Team verschiedene Personen benennen, die es je nach Thema am besten vertreten können.

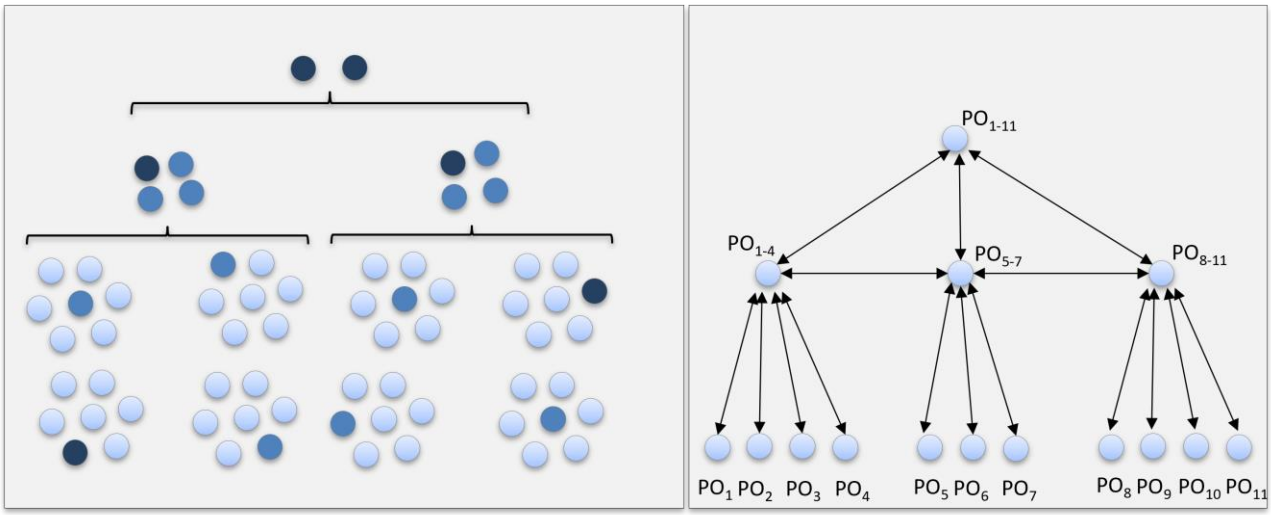

Abbildung 34: Scrum of Scrums als Modell für die Organisation der Anforderungsverantwortung

Die in Kapitel 6.1.1 diskutierte Anforderungshierarchie erfordert neben der allgemeinen Koordination der Entwickler auch eine entsprechende Hierarchie der Anforderungsverantwortung (Abbildung 34, rechts). Grobgranulare Anforderungen und Anforderungen mittlerer Granularität sollten jemandem gehören, Verfeinerungsaufgaben sollten einzelnen Teams zugewiesen werden und Abhängigkeiten zwischen den Teams sollten identifiziert werden.

Die Organisation der Rollen auf den verschiedenen Ebenen der Organisationshierarchie unterscheidet sich von Framework zu Framework: Von der Basisdemokratie bis hin zu eindeutig hierarchischen Strukturen.

Zu den eher demokratischeren Ansätzen gehören Nexus und BOSSA Nova. Sie schlagen keine *PO-Hierarchien* vor. Bei diesen beiden Frameworks ist der Product Owner Teil des Teams, und das Team entscheidet, wie nicht nur die Entwicklung, sondern auch die Anforderungen koordiniert werden sollen.

So kommt Nexus der Idee eines Scrum of Scrums (d.h. selbstorganisierende Teams) mit seinem *Nexus Integration Team* nahe, das dazu da ist, die Anwendung von Nexus und den Betrieb von Scrum zu koordinieren, zu coachen und zu beaufsichtigen, um die besten Ergebnisse zu erzielen. Das Nexus Integration Team besteht aus dem Product Owner, einem Scrum Master und den Mitgliedern des Nexus Integration Teams. Das Nexus Integration Team ist jedoch keine Entscheidungsinstanz: Ähnlich wie ein Scrum Master eines einzelnen Teams sorgt das Integrationsteam vor allem dafür, dass die notwendige Kommunikation zwischen den Teams stattfindet, um gemeinsame Probleme zu lösen.

Eine noch stärkere Basisdemokratie wird von BOSSA Nova [BOSSANOVA] befürwortet. Hier wird eine Soziokratie [SOZIOKRATIE] als die ideale Form für die Organisation im Großen vorgeschlagen. Die Teams wählen ihre Botschafter für den Koordinierungskreis aus, und jeder Koordinierungskreis wählt seinen Botschafter für die übergeordneten Koordinierungskreise aus, usw.

Andere Frameworks schaffen klarere Strukturen für das Anforderungsmanagement mit genau definierten Entscheidungsbefugnissen. Sie weisen den Anforderungskoordinatoren auf höheren Ebenen häufig feste Job Titel zu. Wie wir oben bei den Anforderungshierarchien gesehen haben, variiert auch die genaue Terminologie, die in den Organisationshierarchien verwendet wird, zwischen den verschiedenen Frameworks. Abbildung 35. gibt einen Überblick über einige der in ausgewählten Frameworks verwendeten Job Titeln und Rollennamen.

Framework / Anforderungen Koordinierung...	Nexus	Scrum@Scale	SAFe	LeSS
... auf Portfolio-Ebene (Organisationsweit)			EPIC Owner	
... von Teams von Teams von Teams		EMS (Executive Meta Scrum)	Solution Management	
... der Teams von Teams	Nexus Integration Team	Chief Product Owner	Product Management	Product Owner
... auf Teamebene	Product Owner	Product Owner	Product Owner	Area Product Owner

Abbildung 35: Rollennamen für die Anforderungsverantwortung

Einige Frameworks (Scrum@Scale, Nexus, SAFe) reservieren die Rollenbezeichnung „Product Owner" für das einzelne Team und schlagen neue Rollenbezeichnungen für die übergeordneten Koordinationsrollen vor. Scrum@Scale verwendet zum Beispiel den Begriff Chief Product Owner.

In SAFe ist der *Product Manager* für den Output mehrerer Teams verantwortlich, die zusammen einen *Agile Release Train* bilden. Wenn mehrere Agile Release Trains zusammenarbeiten, um die Anforderungen einer noch größeren Lösung zu erfüllen, werden sie von einem *Solution Manager* gemanagt. Auf der höchsten Granularitätsebene, der unternehmensweiten Agilität, tragen die *Epic-Owner* die Gesamtverantwortung für die Anforderungen und bilden zusammen das *Portfolio-Management*.

LeSS geht den umgekehrten Weg und besagt, dass selbst bei großen Teams die Verantwortung beim Product Owner liegt. Einzelne Teams können dann *Area Product Owner* zuweisen, die die Anforderungen für den Teil des Produkts verwalten, der kleineren Teams zugewiesen ist.

Sie sollten sich merken: Stellenbezeichnungen spielen keine Rolle, solange es eine Person (oder eine kleine Gruppe) gibt, die für die Verwaltung der Anforderungen zuständig ist. Alle Frameworks schlagen vor, mit einem einzigen Product Backlog zu arbeiten, unabhängig von der Größe des Teams (weitere Einzelheiten zu logischen Backlogs finden Sie in Kapitel 6.2). Teile dieses einen Backlogs können dann Unterteams zugewiesen werden.

Welchen Mechanismus Sie auch immer verwenden, stellen Sie sicher, dass die Unterteams (oder ihre Vertreter) regelmäßig über Überschneidungen, Abhängigkeiten und Prioritäten sprechen, um das beste Ergebnis für die Gesamtentwickler zu erzielen.

6.1.3 Organisieren von Lebenszyklen/Iterationen

In unserer Definition in Kapitel 1.3 haben wir erläutert, dass RE@Agile ein iterativer Prozess ist. Für große Projekte schlagen die meisten Frameworks zwei verschiedene Arten von Iterationen vor:

- Kurze Iterationen (oft Sprints genannt): Hier versuchen einzelne Entwickler, die in der Sprint-Planungssitzung zugewiesenen Backlog Items zu implementieren. Diese kurzen Iterationen dauern in der Regel zwischen zwei und vier Wochen.
- Längere Iterationen (oft als Releases bezeichnet): Hauptsächlich um die Integration der Ergebnisse mehrerer Teams zu gewährleisten. Releases können eine Reihe kurzer Iterationen enthalten. Verschiedene Frameworks legen unterschiedliche Regeln für die Häufigkeit der Integration fest, dies reicht von Integration in jeder Iteration bis zur Integration mindestens in jedem Release. Release-Iterationen sollten nicht länger als zwei bis drei Monate dauern.

Weitere Informationen zu Release-Planung und Roadmapping finden Sie in Kapitel 6.3.

6.2 Kriterien für die Strukturierung von Anforderungen und Teams im Großen

Bei der Produktentwicklung in großem Maßstab müssen mehrere Teams gemeinsam am gleichen Produkt arbeiten. In der Praxis entwickelt jedes Team ein spezifisches Produkt-Stück, das mit anderen Stücken integriert werden muss, um eine funktionierende Lösung zu schaffen. Nur das integrierte Produkt hat einen Wert für die Stakeholder.

Bei der Skalierung der Produktentwicklung auf mehrere Teams reicht es nicht aus, dass sich alle Product Owner einfach treffen und irgendwie besprechen, welche Teams welchen Teil des Produkts entwickeln sollen und dann auf das Beste hoffen! Ausgefeilte Strukturen und Praktiken werden benötigt, um die Zusammenarbeit in Teams, Anforderungsänderungen und eine schnelle Produktlieferung zu ermöglichen. Andernfalls verschwenden Entwickler möglicherweise Aufwand bei der Koordination mit anderen Teams, die für ihre Arbeit nicht relevant sind.

Aus der Anforderungsperspektive müssen wir den „Kreislauf" schließen: Von der anfänglichen (Business-)Anforderung der Stakeholder, über die Aufteilung komplexer Anforderungen in kleinere, von jeweils einem Team an Entwicklern handhabbare Teile, bis hin zur Sicherstellung, dass die zusammengesetzten Ergebnisse eine Lösung bilden, die für das Business freigegeben werden kann.

6.2.1 Produktorientiertes Backlog

Product Owner brauchen ein gemeinsames Verständnis für das Produkt und seinen geschäftlichen Kontext. Dies ist wichtig, da sie gemeinsam an den Anforderungen auf verschiedenen Abstraktionsebenen arbeiten und sich auf die Prioritäten der einzelnen Teams einigen müssen, die auch die allgemeinen Geschäftsprioritäten widerspiegeln sollten.

Außerdem müssen agile Teams Anforderungsüberschneidungen und Abhängigkeiten erkennen, um Unterbrechungen während der Entwicklung zu minimieren.

Um diese Art von Produktfokus zu unterstützen, müssen die Anforderungen in einem logischen Backlog verwaltet werden. Der Kerngedanke ist, dass jede Anforderung nur an einer Stelle verwaltet wird, um Redundanzen und Widersprüche zu vermeiden. Dies kann auch bei einer weiteren Unterteilung des Backlogs in Team-Backlogs erreicht werden, wie in Abbildung 36 dargestellt. Während der Verfeinerung grober Anforderungen können Product Owner an Backlog Items arbeiten, die noch keinem Team zugeordnet sind (siehe (a) in Abbildung 36), oder sie können komplexe Anforderungen aufteilen und die daraus resultierenden Backlog Items an die Teams zur weiteren Verfeinerung weitergeben (siehe (b) und (c) in Abbildung 36). Um die Verfolgbarkeit zwischen Anforderungen auf verschiedenen Abstraktionsebenen sicherzustellen, sollten Product Owner die Backlog Items miteinander verknüpfen.

Nehmen wir zum Beispiel eine komplexe Anforderung, die die Verbindung eines speziellen Hardware-Devices mit einer Computeranwendung über ein proprietäres Protokoll beschreibt. Diese Anforderung wird zunächst im Product Backlog gespeichert (siehe (a) in Abbildung 36). Angenommen, Team A und B entwickeln das System, wobei Team A Erfahrung mit dem Hardware-Device hat. Nun kann die komplexe Anforderung aufgeteilt werden in eine kleinere Anforderung, die sich auf die Schnittstelle des Hardware-Devices konzentriert und im Backlog von Team A verwaltet wird, und eine weitere Anforderung, die die Handhabung der Verbindung innerhalb der App beschreibt (siehe (c) in Abbildung 36), die im Backlog von Team B verwaltet wird.

Je nach dem welches Tool für die Backlog-Verwaltung verwendet wird, können Sie entweder Teamfilter auf dem gemeinsamen Product Backlog definieren, oder Sie können (virtuelle) Backlogs für jedes Team erstellen. Unabhängig vom gewählten Tooling bilden alle Backlog Items zusammen ein logisches Backlog.

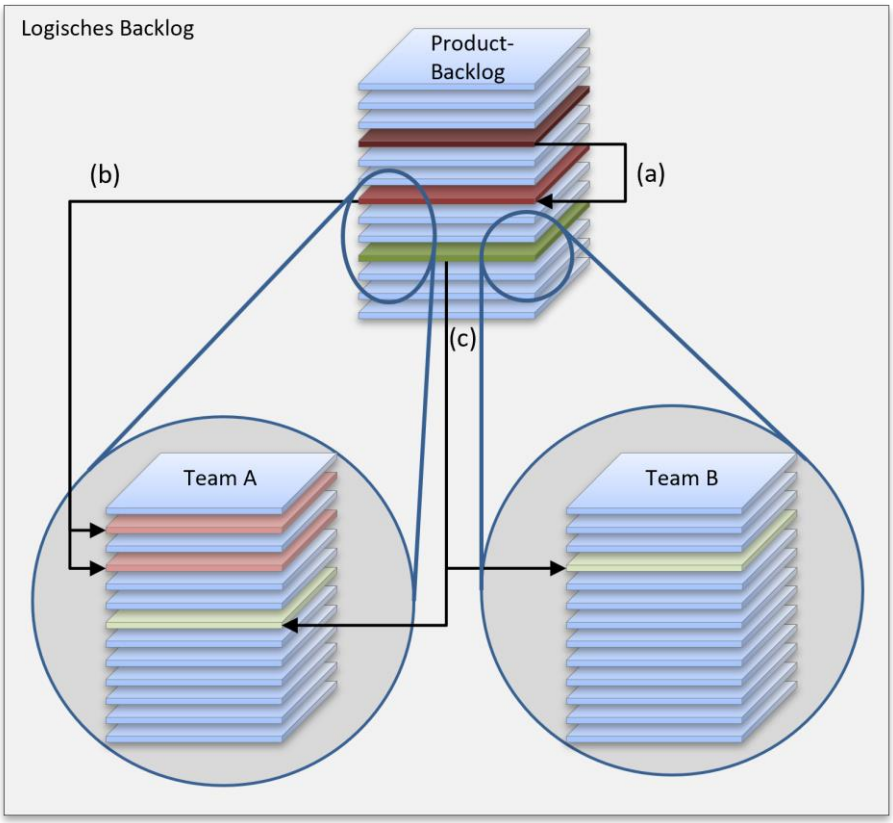

Abbildung 36: Schlüsselidee des logischen Backlog-Ansatzes.

In Skalierungs-Frameworks wie Nexus, SAFe und Less wird ebenfalls ein logisches Product Backlog empfohlen.

In SAFe wird das logische Backlog in verschiedene Backlogs aufgeteilt, die entsprechend ihrer Skalierungsebene verknüpft sind (z.B. Portfolio Backlog, Solution Backlog, Program Backlog, mehrere Team Backlogs). Jedes Backlog enthält Anforderungen von angemessener Granularität entsprechend der Skalierungsstufe. So werden beispielsweise Backlog Items aus dem Program Backlog in Team Backlogs verfeinert, während zusätzliche Elemente, die sich aus dem lokalen Kontext des Teams ergeben, auch direkt in die Team Backlogs aufgenommen werden können.

6.2.2 Selbstorganisierte Teams und kollaborative Entscheidungsfindung

Die Produktentwicklung wird es schwer haben zeitnah auf Änderungen zu reagieren, wenn jedes Team auf ein kompliziertes Geflecht von Interaktionen mit anderen Teams angewiesen ist, um jede Entscheidung abzustimmen. Es ist eine Teamstruktur erforderlich, die es den Teams ermöglicht, sich selbst um die Wertschöpfung herum zu organisieren: Um besser auf das Feedback der Stakeholder reagieren zu können, um unabhängig vernünftige Entscheidungen zu treffen und um End-to-End-Features zu liefern [Anderson2020].

Die Vorteile die selbstorganisierende Teams mit sich bringen ist eines der agilen Prinzipien [AgileManifestoPrinciples]. Eine lokalisierte, direkte Kommunikation innerhalb von Teams (Intra-Team) ermöglicht Optimierungen und eine effektive Entscheidungsfindung, während die Kommunikation zwischen verschiedenen Teams (Inter-Team) langsamer ist und im Allgemeinen auf ein Minimum beschränkt werden sollte [Reinertsen2008].

Dennoch wird es immer einen Bedarf an Zusammenarbeit innerhalb eines Netzwerks von Teams geben, die auf ein gemeinsames Ziel hinarbeiten. Es ist ein Maß an Kommunikation und Koordinierung erforderlich, was zwangsläufig den Freiheitsgrad der einzelnen Teams einschränkt.

Um gemeinsam an den Anforderungen zu arbeiten und eigenständig vernünftige Entscheidungen zu treffen, brauchen Teams ein allgemeines Verständnis für die Anforderungen der anderen Teams, mit denen sie zusammenarbeiten müssen, ohne jedoch mit allen Details überfordert zu sein. Die Product Owner sollten daher einen angemessenen Detaillierungsgrad finden, der es den Teams ermöglicht, die Auswirkungen ihrer Entscheidungen auf andere Teams zu verstehen.

6.2.3 Verständnis der merkmalbasierten Anforderungsaufteilung

Die Aufteilung von Anforderungen ist in der agilen Entwicklung notwendig, um größere Anforderungen in feingranulare Anforderungen aufzuteilen, die in einer Iteration umgesetzt werden können. Wie in Kapitel 3.4. erörtert, gibt es verschiedene Aufteilungstechniken, die in der agilen Entwicklung unabhängig von der Anzahl der beteiligten Teams angewendet werden sollten. Die Aufteilung von Anforderungen ist jedoch in der groß angelegten Produktentwicklung viel grundlegender, da sie selbstorganisierende Teams ermöglicht, die in der Lage sein müssen, Anforderungen unabhängig voneinander umzusetzen.

Um lieferbare Produktinkremente mit minimalen Abhängigkeiten zu anderen Teams zu liefern, sollten agile Teams an lose gekoppelten End-to-End-Features arbeiten. In unserem Kontext bezieht sich der Begriff „End-to-End-Feature" auf eine Reihe von zusammenhängenden Funktionen, die eine bestimmte Aufgabe erfüllen und den Stakeholdern einen geschäftlichen Nutzen bieten. Je nach Abstraktionsebene, auf der die Aufteilung erfolgt, kann die Definition der Aufgabe jedoch von spezifischen Benutzerfunktionen bis hin zu ganzen Geschäftsprozessen reichen.

Um End-to-End-Features zu identifizieren, müssen Product Owner den Produktumfang in Einheiten lose gekoppelter und intern konsistenter Funktionalität (d.h. funktionale Grenzen) zerlegen, wie in Abbildung 37 dargestellt. Wenn der Umfang nach diesen funktionalen Grenzen aufgeteilt wird, können die einer bestimmten Einheit zugewiesenen Product Owner an den Anforderungen mit einem höheren Grad an Unabhängigkeit arbeiten. Entsprechende Teams werden oft als Feature-Teams [Larman2017] bezeichnet.

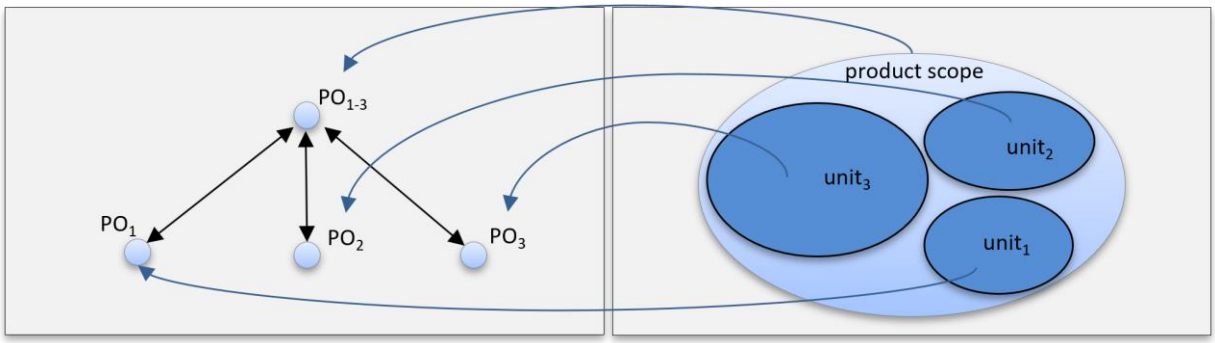

Abbildung 37: Der Umfang wird in kleinere Einheiten der End-to-End-Funktionalität unterteilt und unter den Product Ownern aufgeteilt.

Ein Product Owner und in der Regel ein agiles Team werden einer Einheit von End-to-End-Funktionalität zugewiesen. Die Grenzen zwischen den Einheiten helfen bei der Festlegung der Kommunikationswege. Die Grenzen sollten klar sein, um eine effektive Zusammenarbeit zu ermöglichen. Die Product Owner können sich auf die detaillierten Anforderungen konzentrieren, die ihrer Einheit zugewiesen sind, anstatt viel Zeit damit zu verbringen, den gesamten Umfang und den geschäftlichen Kontext zu verstehen. Sie müssen nur bei Anforderungen, die benachbarte Einheiten betreffen, mit anderen Product Ownern zusammenarbeiten. Anforderungen können hierarchisch auf der Grundlage unabhängiger Einheiten organisiert werden, wie in Kapitel 6.2.1 erläutert.

Die Aufteilung des Produktumfangs kann entlang von Geschäftsprozessen erfolgen, wie in Kapitel 3.2 beschrieben. Wenn ein Geschäftsprozess aus mehreren Prozesslinien besteht, kann jede Linie durch End-to-End-Produktfeatures auf Geschäftsebene unterstützt werden. Idealerweise sollten die verschiedenen Prozesslinien innerhalb eines Geschäftsprozesses lose gekoppelt sein, was es den Product Ownern in der Regel ermöglicht, unabhängig an den Anforderungen ihrer Features zu arbeiten. In diesem Fall müssen sie sich nur auf Features einigen, die das Zusammenspiel der Prozesslinien betreffen.

Use Cases sind ein Ansatz zur Strukturierung von Anforderungen, der nicht immer mit Agilität in Verbindung gebracht wird, aber dennoch von einer Reihe von Autoren empfohlen wird (z.B. Jacobsen, Cockburn, Leffingwell). Use Cases betrachten das System als Blackbox und betrachten die Aktionen, die zwischen einem Akteur (Mensch oder ein anderes System) und der Lösung stattfinden.

Use Cases können als Teil der vorbereitenden Aktivitäten zum Ermitteln des Umfangs und zur Strukturierung eines Projekts verwendet werden, wie in Kapitel 0 beschrieben, oder als Teil der laufenden Produktentwicklung ausgearbeitet werden. Im Gegensatz zu Prozesslinien kann ein Use Case als eine End-to-End-Funktionalität des Produkts auf Benutzerebene betrachtet werden. Die Product Owner müssen sich nur auf Anforderungen einigen, die sich auf mehrere Use Cases beziehen (z. B. Schnittstellen oder gemeinsame Geschäftseinheiten).

6.2.4 Überlegungen, wenn eine Feature-basierte Aufteilung der Anforderungen nicht möglich ist

Leider ist es in vielen Fällen nicht so einfach, Anforderungen basierend auf lose gekoppelten Einheiten von einer End-to-End-Funktionalität zu zerlegen. Aufgrund des architektonischen Designs (z. B. Technologie, Infrastruktur, Systemkomponenten, gemeinsame Plattform, Architekturschichten wie Front- und Backend) sowie organisatorischer Überlegungen (fachliche Kompetenzen, Standort des Teams, Sub-Auftragnehmer) können sich Funktionalitäten überschneiden wie in Abbildung 38 dargestellt. Das bedeutet, dass verschiedene agile Teams zusammenarbeiten müssen, um bestimmte Features zu implementieren und, dass ihre jeweiligen Product Owner bei den Anforderungen enger zusammenarbeiten müssen [Abbildung 38]. Alternativ kann ein spezielles Team eingerichtet werden, das sich mit der Überschneidung befasst und mit jedem der ursprünglichen Teams zusammenarbeitet, die sich auf eine Funktionseinheit konzentrieren.

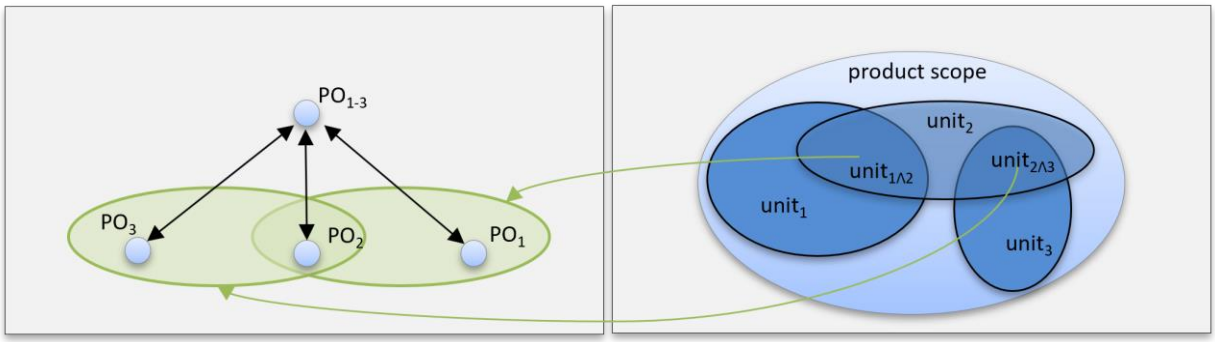

Abbildung 38: Sich überschneidende Einheiten weisen auf eine enge Zusammenarbeit der Product Owner in Bezug auf die Anforderungen hin.

Um Features kollaborativ zu implementieren, benötigen agile Teams ein gemeinsames Verständnis der Anforderungen und deren geschäftlichen Kontextes. Sie müssen sich auch auf übergreifende (cross-cutting) Anforderungen, Constraints und gemeinsame technische Schnittstellen einigen, damit die Ergebnisse verschiedener Teams zu funktionierenden Inkrementen integriert werden können. Die Integration und das Testen von Features werden komplexer und die Synchronisierung der Teams mithilfe von Backlogs und Roadmaps ist noch wichtiger (s. Kapitel 6.3).

Verteilte Teamstandorte in verschiedenen Zeitzonen stellen eine besondere Herausforderung für die Kommunikation dar und erfordern einen höheren Koordinationsaufwand. Wenn beispielsweise Entwickler aus mehreren verteilten Teams bestimmte Features gemeinsam implementieren müssen, müssen die Product Owner mehr Zeit für die Zerlegung der Anforderungen dieser Features aufwenden, um den Kommunikationsaufwand zu minimieren. Meetings (virtuell oder physisch!) müssen explizit mit zusätzlichem Planungsaufwand und zu möglicherweise ungünstigen Zeiten organisiert werden. Unterschiedlich gesprochene Sprachen oder Kulturen können weitere Probleme aufwerfen.

Teams, die an verschiedenen Standorten, aber in der gleichen oder einer benachbarten Zeitzone arbeiten, haben diese Schwierigkeiten nicht, müssen aber dennoch einige Anstrengungen unternehmen, um eine effektive Kommunikation zu organisieren, sei es durch virtuelle oder physische Meetings oder durch den Einsatz anderer Collaboration-Tools. Videokonferenzen und Collaboration-Tools können hier von großem Nutzen sein.

Eine besondere Form von verteilten Teams sind Teams von Unterauftragnehmern. Solche Teams sind nicht unbedingt geografisch verteilt, sondern eher organisatorisch, d. h. die Teammitglieder sind Mitarbeiter einer anderen Organisation, die in einem Vertragsverhältnis zu anderen Teams steht.

Idealerweise sollten Product Owner nicht Unterauftragnehmer sein, da Interessenkonflikte sie daran hindern könnten, die volle Produktverantwortung zu übernehmen. Unterauftragnehmer haben oft ihre eigenen Ziele, die mitunter nicht vollständig mit der allgemeinen Produktvision oder den Zielen übereinstimmen.

Jedes Team muss einen Mehrwert für die Produktinkremente liefern. Einige Teams implementieren keine Features, sondern konzentrieren sich auf die Verwaltung der Infrastruktur oder helfen anderen Teams bei der Integration von Ergebnissen in Produktinkremente. SAFe schlägt zum Beispiel vor, ein dediziertes Systemteam zu haben, das die Integration aller Teamartefakte in ein releasefähiges Produktinkrement vornimmt. Das Nexus-Framework sieht ein „Nexus-Integrationsteam" vor, das die Arbeit nicht ausführt, sondern die Entwickler berät, wie sie dies selbst tun können. Sie tragen also implizit zum Produktinkrement bei.

Weitere Details zu agilem Organisationsdesign und -verfahren finden Sie bei [Anderson2020].

Schließlich sollten wir uns der Beobachtung von Conway bewusst sein, der ein sehr häufiges Muster beschrieben hat, das als „Conway's Law" bekannt ist. Es weist darauf hin, dass die Organisationsstruktur einen Einfluss auf die Systemgestaltung und die Produktstruktur hat. In seinem Artikel [Conway1968] stellt Conway fest, dass Organisationen, die neue Systeme oder Produkte entwickeln, dazu neigen, ihre Produkte auf die gleiche Weise zu strukturieren, wie sie selbst derzeit organisiert sind und kommunizieren. Die sich daraus ergebende Teamstruktur ist im Hinblick auf eine effiziente Entwicklung und Auslieferung in einem groß angelegten agilen Kontext oft suboptimal.

6.2.5 Beispiel eines Telekommunikationsunternehmens

In diesem Beispiel veranschaulichen wir den oben genannten Ansatz zur Feature basierten Anforderungsaufteilung und erörtern den Einfluss des organisatorischen Kontexts auf die Struktur agiler Teams und ihre Fähigkeit, dem Kunden funktionierende Produktfeatures zu liefern.

Nehmen wir das Beispiel eines Telekommunikationsunternehmens, das zwei neue Breitbandprodukte entwickeln und seinen Kunden anbieten möchte:

1. Ein neues Hochgeschwindigkeits-VDSL-Produkt (Internet über die Telefonleitung) „VDSL100"
2. Ein Fiber-to-the-Home-Produkt (Internet über Glasfaser) „FTTH1000". In einer ersten Phase analysierten die Product Owner die beiden neuen Produkte und erstellten gemeinsam die Anforderungshierarchie entsprechend den wichtigsten Geschäftsprozessen, wie in Abbildung 39 dargestellt:

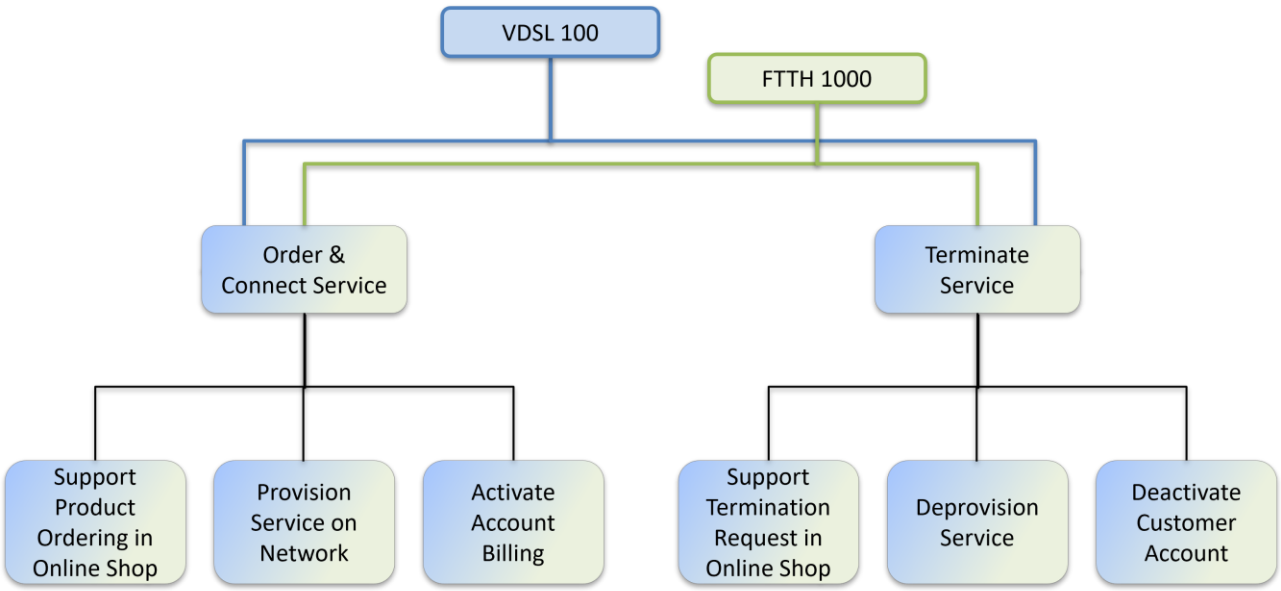

Abbildung 39: Struktur der Breitband-Produktanforderungen

Auch wenn die Details der einzelnen Anforderungen für die beiden Produkte unterschiedlich sein können, ist die Organisation der grobgranularen Anforderungen dieselbe.

Um seinen Kunden die beiden Produkte anbieten zu können, muss das Telekommunikationsunternehmen sein bestehendes IT-System erweitern. Aus Gründen, die mit der Geschichte des Unternehmens zusammenhängen, sind die wichtigsten IT-Systeme sowie die Ressourcen und Kompetenzen innerhalb des IT-Teams wie folgt organisiert: (1) *Online-Shop und Kundendienstportal*, (2) *Kundenkonto- und Abrechnungssystem* und (3) *Netzwerkbereitstellungs- und Installationssysteme*.

Das bedeutet, dass der Online-Shop und das Kundendienstportal als ein einzelnes IT-Produkt betrachtet werden, mit einem vollständigen Technologie-Stack aus Front-End, Geschäftslogik und Persistenzschicht. Dies gilt auch für das Kundenkonten- und Abrechnungssystem. Entwickler spezialisieren sich in der Regel auf das eine oder andere System, aber nicht auf beide.
Die Netzwerk- und Bereitstellungssysteme sind vielfältiger, werden aber in ähnlicher Weise von spezialisierten technischen Rollen betreut.

Im Zuge der Umstellung des Unternehmens auf einen skalierten agilen Ansatz treffen sich Führungskräfte des Telekommunikationsunternehmens mit Product Ownern, um die beste Struktur für agile Teams zu besprechen. Die erste vorgeschlagene Teamstruktur und die zugewiesenen Produktanforderungen sind in Abbildung 40 dargestellt:

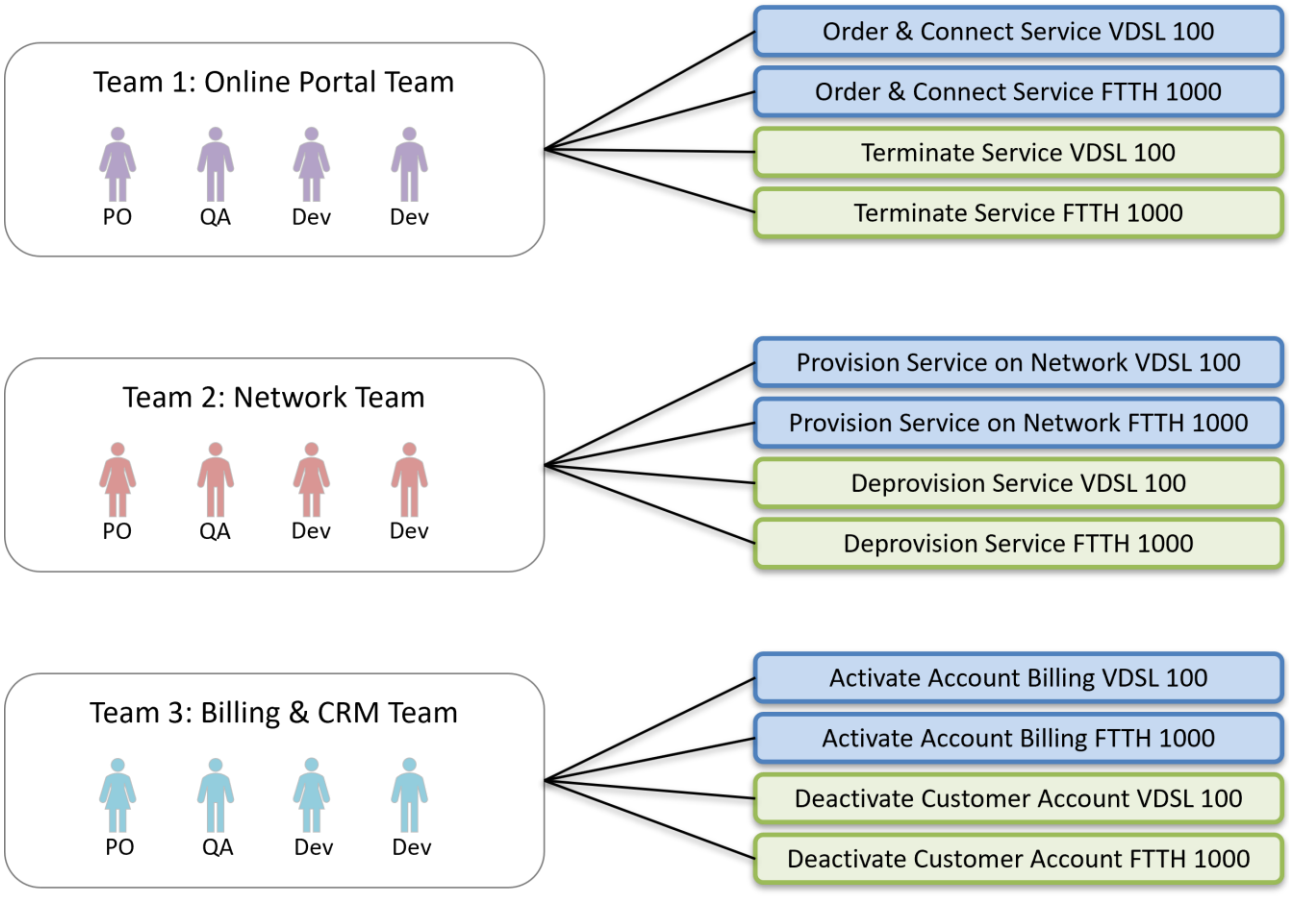

Abbildung 40: Die Teamstruktur entspricht der Organisationsstruktur

Die Zusammensetzung der agilen Teams entspricht weitgehend der bestehenden Organisationsstruktur. Die Teammitglieder sind Spezialisten für das entsprechende System und arbeiten an Anforderungen, die dieses System betreffen. Die Kommunikation zwischen den Teams ist in erster Linie erforderlich, um sicherzustellen, dass die Systeme zusammenarbeiten, um die beiden Dienste erfolgreich zu starten. Kein Team ist in der Lage, unabhängig voneinander funktionierende Features zu liefern, die einen Kunden, der an einem der beiden Produkte interessiert ist, vollständig unterstützen.

Neben dem Product Owner eines jeden Teams, der auf die Anforderungen des jeweiligen Systems spezialisiert ist, können weitere Product Owner erforderlich sein, um die Bereitstellung der grobgranularen End-to-End Prozessanforderungen zu koordinieren.

Um den Kommunikationsaufwand zwischen den Teams zu verringern, wird eine zweite Zusammensetzung der agilen Teams vorgeschlagen, die in Abbildung 41 dargestellt ist:

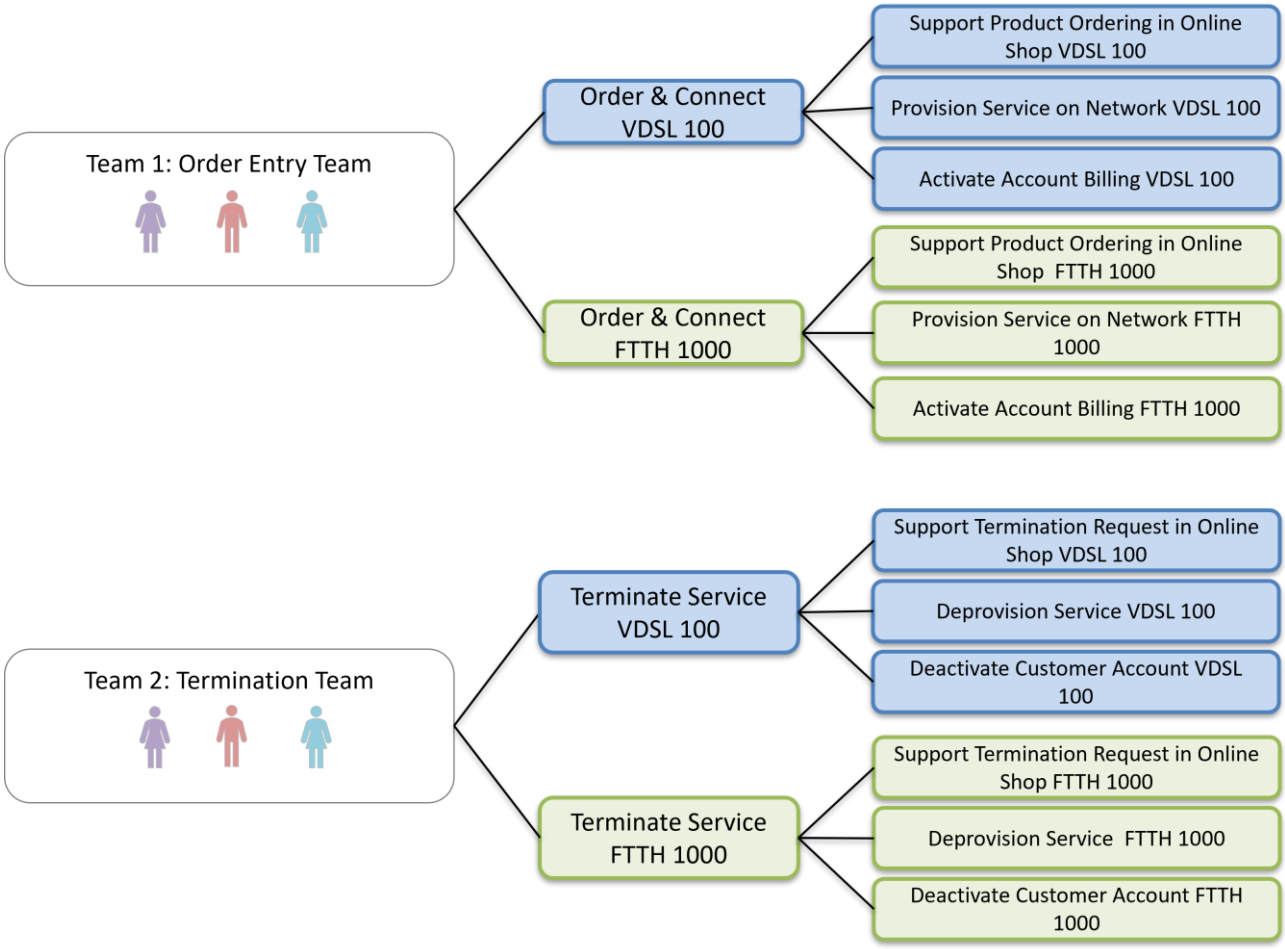

Abbildung 41: Teamstruktur nach Verbindungs- und Kündigungsdiensten

Jedes Team ist für einen wichtigen Geschäftsprozess zuständig, und in jedem agilen Team sind Experten aus den jeweiligen Systemen vertreten. So ist ein Team in der Lage, ein End-to-End-Prozessfeature zu liefern (z. B. die Bestellung eines Breitbandprodukts) und den Kunden einen Mehrwert zu bieten (wie bei den Feature-Teams, die in Kapitel 6.2.3 erörtert werden). Aus Sicht der Anforderungen wird der Koordinierungsaufwand reduziert, da jeder Product Owner sein Produkt mit größerer Autonomie entwerfen kann. Koordinierungsbedarf besteht vor allem auf der Produktebene (VDSL 100, FTTH 1000), um beispielsweise ein einheitliches Produktmodell über die verschiedenen Prozesse hinweg zu gewährleisten. Da die integrierte Lösung drei einzelne IT-Systeme umfasst, ist eine Kommunikation zwischen den Teams erforderlich, um Änderungen und Releases innerhalb eines bestimmten Systems zu koordinieren.

Eine andere Zusammensetzung agiler Teams wird unter Abbildung 42 erörtert, wobei ebenfalls das Konzept von Teams mit vollständigen End-to-End-Fähigkeiten betont wird:

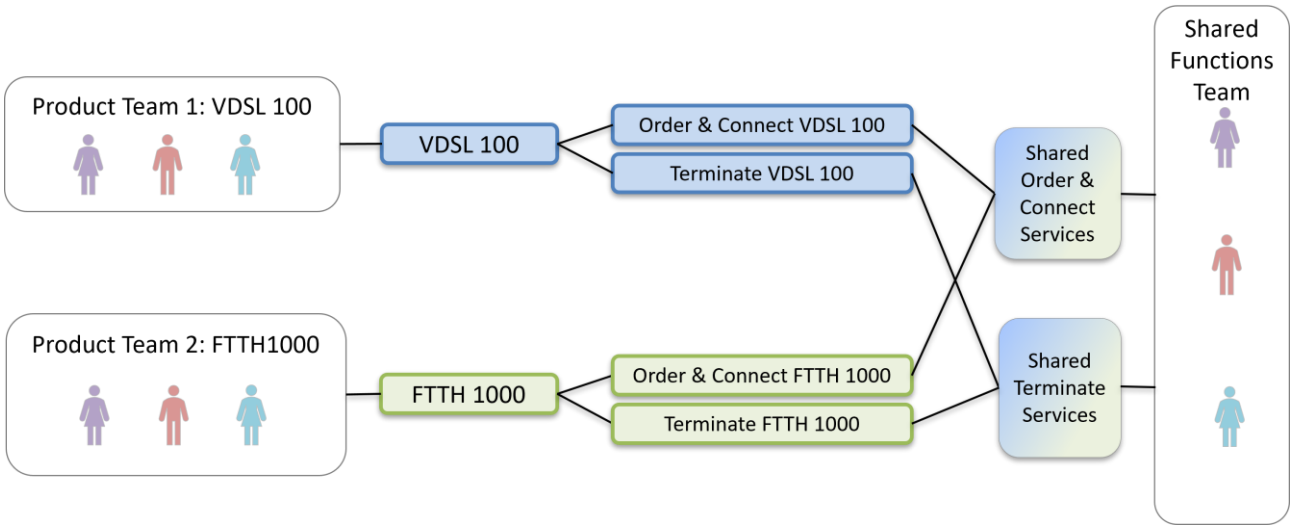

Abbildung 42: Teamstruktur mit vollständigen End-to-End-Fähigkeiten

Hier ist jedes agile Produktteam in der Lage, ein marktfähiges Produkt mit all seinen Features (VDSL 100, FTTH 1000) zu liefern. Mit Expertise über alle Systeme und Geschäftsprozesse ist jedes Team in der Lage, unabhängig voneinander geschäftlichen Mehrwert zu schaffen. Aus agiler Sicht sollte diese Teamstruktur bevorzugt werden. In der Praxis besteht jedoch ein hohes Risiko, dass diese Teams bei der Bearbeitung der sich überschneidenden Anforderungen Funktionen doppelt ausführen. Um dieses Problem anzugehen, wird ein Team für gemeinsame Funktionen vorgeschlagen, das sich auf genau diese Überschneidungen spezialisiert und die Aufgabe hat, allgemeine Lösungen für beide Produkte zu finden: Nutzung bestehender Systeme und Dienste, wo dies möglich ist, oder Entwicklung von Enabler-Features, wo dies zur Unterstützung dieser und anderer Produkte angebracht ist (siehe die Unterscheidung zwischen Geschäftsfunktionen und Enabler Features unter 6.1.1).

Welchen Ansatz sollten wir also wählen? Leider gibt es keine einfache Antwort. Wie bereits erwähnt, hängt der bevorzugte Ansatz von vielen Faktoren ab: Der bestehenden Organisationsstruktur, der Bereitschaft zur Veränderung, den technischen und architektonischen Beschränkungen sowie dem Grad der gemeinsamen Nutzung von Funktionalität in den verschiedenen Produkten und Prozessen. Idealerweise würden wir zuerst die Anforderungen strukturieren und dann versuchen, so weit wie möglich Feature-Teams zu bilden, aber in Wahrheit muss nach sorgfältiger Abwägung all dieser Faktoren ein Gleichgewicht gefunden werden.

6.3 Roadmaps und umfangreiche Planung

In der groß angelegten Produktentwicklung verwalten Product Owner Anforderungen im produktfokussierten Backlog, wie in Kapitel 6.2.1 diskutiert. Im Gegensatz zum Backlog wird eine Roadmap zur inkrementellen Planung der Produktentwicklung verwendet. Eine Roadmap ist eine Vorhersage, wie das Produkt wachsen wird [Pichler2016]. Roadmaps verändern nicht den Inhalt der Backlog Items, sondern ordnen sie auf einer Zeitachse an. Sie beantworten die Frage, wann wir welche Features ungefähr erwarten können.

Eine Roadmap ist ein nützliches Mittel, um (strategische) Ziele und Entscheidungen an die Entwickler und andere Stakeholder zu kommunizieren. Sie bricht ein langfristiges Ziel in überschaubare Iterationen herunter, stellt Abhängigkeiten zwischen den Teams dar und gibt den Stakeholdern Orientierung und Transparenz.

Eine Roadmap ist das Ergebnis einer Planung, wie sie unter Abbildung 43 dargestellt ist. Die Basis für die Planung ist einerseits das sortierte und geschätzte Product Backlog und andererseits die verfügbaren Entwickler und deren Kapazitäten.

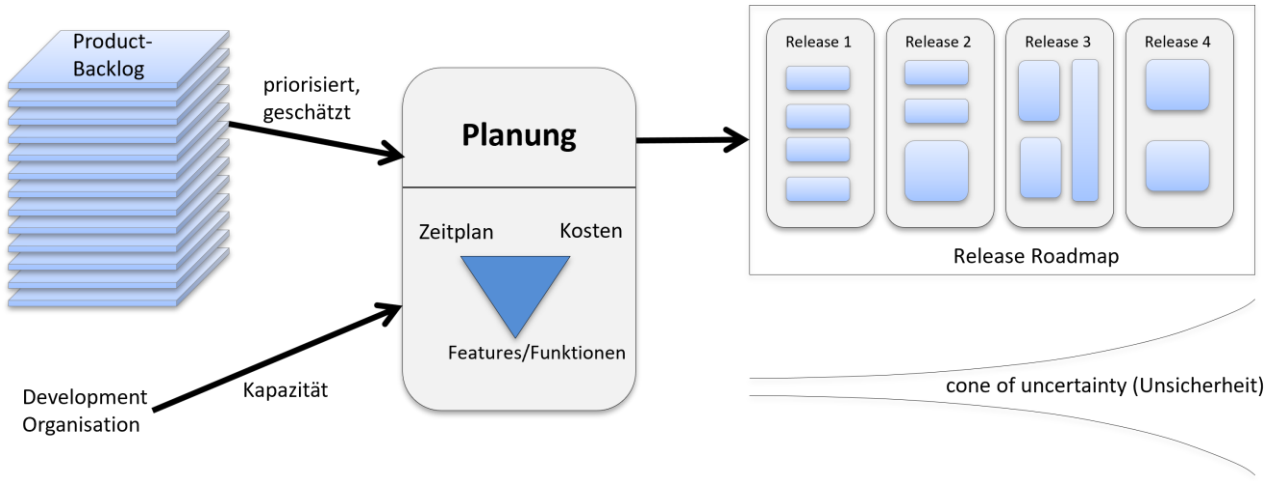

Abbildung 43: Beispiel einer Planung

Dieser Input konfrontiert den Product Owner mit dem typischen Projektmanagement-Dreieck, dem Abwägen von Systemumfang (Features oder Funktionalität des Produkts), Kosten (verfügbare Ressourcen) und Zeitplan (Auslieferungszeitpunkt). Wir haben das Dreieck bewusst auf den Kopf gestellt, um zu zeigen, dass in agilen Projekten Kosten und Zeitplan sehr häufig festgelegt sind und daher die geplanten Features die einzige Variable darstellen.

Zu Beginn der agilen Produktentwicklung ist wenig über das Produkt oder die Arbeit der Teams bekannt. Der Umfang des Produkts sowie die Kostenschätzungen sind daher mit einem hohen Maß an Unsicherheit behaftet. Je mehr Iterationen durchgeführt werden und je mehr Feedback von den Stakeholdern eingeholt wird, desto mehr nimmt die Unsicherheit ab, was zu einer zuverlässigeren Planung und einer stabilen Roadmap führt. Dieses Prinzip ist als *Cone of Uncertainty* [Boehm 1981] bekannt. Der Cone of Uncertainty zeigt jedoch auch, dass Releases, die in Kürze veröffentlicht werden, eine größere Gewissheit darüber bieten, welche Funktionalitäten enthalten sein werden, während weiter in der Zukunft liegende Releases nur vage definiert werden können (siehe Abbildung 43). Obwohl dieser Grundsatz generell für alle agilen Entwicklungsprojekte gilt, wird er bei der Entwicklung umfangreicher Produkte noch wichtiger, da die Risiken aufgrund der Produktkomplexität und der potenziellen schlechten Abstimmung über mehrere Teams hinweg - und folglich der Bedarf an mehr Planung - noch größer sind.

6.3.1 Darstellen von Roadmaps

Eine Roadmap zeigt strategische Ziele, Meilensteine und grobe Anforderungen (beispielsweise eine Menge an Features). Wichtige Meilensteine können entweder intern oder durch externe Ereignisse, wie z.B: eine Fachausstellung oder die Einführung einer neuen Vorschrift, bestimmt sein.

Die Darstellung einer Roadmap ist abhängig von ihrem Zweck, ihrer Zielgruppe und ihrem Planungshorizont. Für Kunden, Management-Sponsoren und das Business ist oft eine langfristige *Produkt-Roadmap* mit strategischen Zielen, groben Produktanforderungen und Features, normalerweise in Fachsprache beschrieben, ausreichend [Pichler2016].

In SAFe wird die Produkt-Roadmap als „Solution Roadmap" bezeichnet und stellt langfristige Meilensteine, strategische Themen und Releases dar. Eine „Solution-Roadmap" bietet in der Regel einen Überblick über einen Zeitraum von einem bis drei Jahren, wobei der Detaillierungsgrad in der nahen Zukunft höher und auf lange Sicht niedriger ist.

SAFe unterteilt eine „Lösung" in kleinere "Programminkremente", die den Kunden einen Mehrwert in Form von funktionierenden Features bieten. Um den kürzeren Planungshorizont darzustellen, führt SAFe die 'Program Increment Roadmap' ein, die bis zu vier Iterationen umfasst. Diese bietet einen detaillierteren Überblick über die in den kommenden Monaten zu leistende Arbeit.

Eine andere Art von *Roadmap*, die in SAFe als „Program Board" [Leffingwell2017] bezeichnet wird, konzentriert sich auf die Bereitstellung. Dies bietet Entwicklern und ihren Product Ownern einen Überblick über feingranulare Backlog Items (z. B. Storys oder Tasks) und die Abhängigkeiten zwischen ihnen.

Eine Produkt-Roadmap unserer Fallstudie iLearnRE, die strategische Ziele und grobgranulare Features enthält, ist in Abbildung 44 dargestellt.

Sie sehen hier die nächsten drei Releases, für das erste gibt es bereits ein Commitment, die beiden anderen sind Prognosen. Jedes Release ist einem vordefinierten Planungshorizont zugeordnet. Die Features werden in fachlichen Begriffen beschrieben und nicht in Form von Epics und Storys.

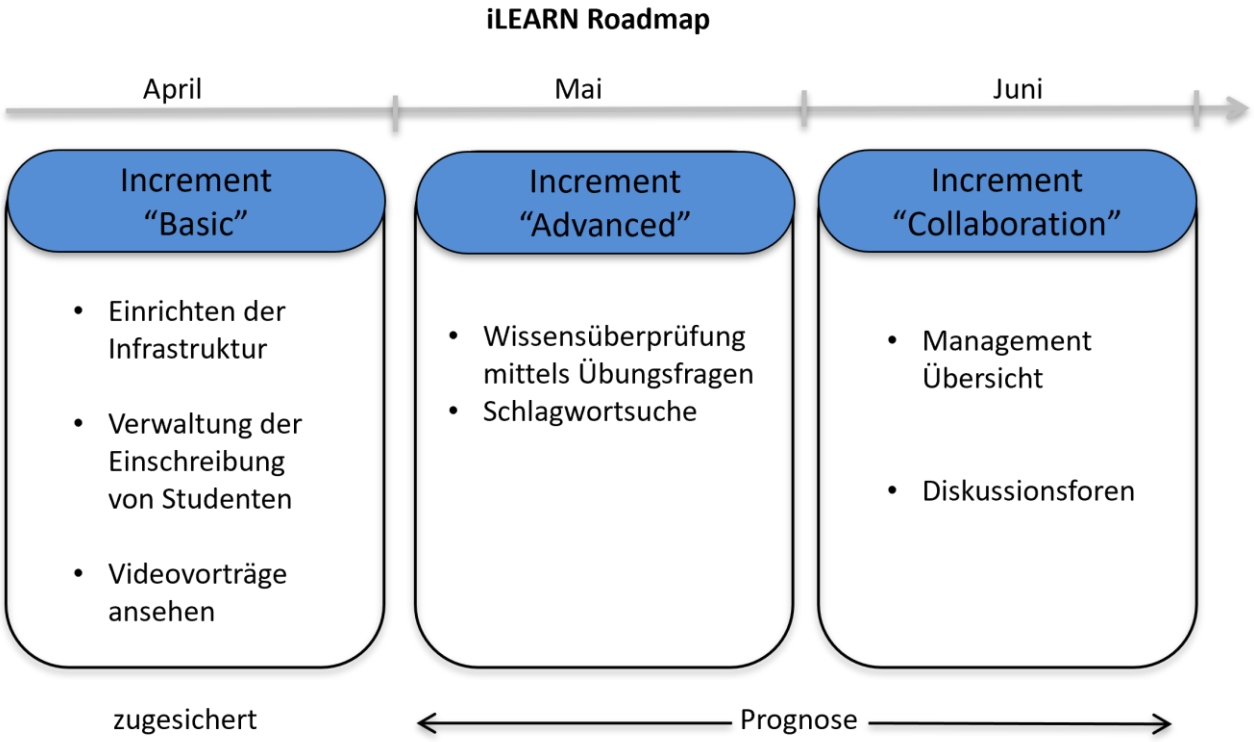

Abbildung 44: Eine Roadmap für die Fallstudie iLearnRE

Im Kapitel 3 haben wir Story-Maps als eine Möglichkeit eingeführt, Ihr Product Backlog zu strukturieren. Diese Maps können erweitert werden, um die Roadmap für die nächsten Releases anzuzeigen, indem man einfach die vertikale Achse benutzt, um Epics, Features und Storys bestimmten Releases zuzuordnen, um so individuelle Release Backlogs zu erzeugen. Dies wird in Abbildung 45 verdeutlicht. Die Elemente auf der Story Map können gröber sein, wenn das Release noch einige Zeit in der Zukunft liegt.

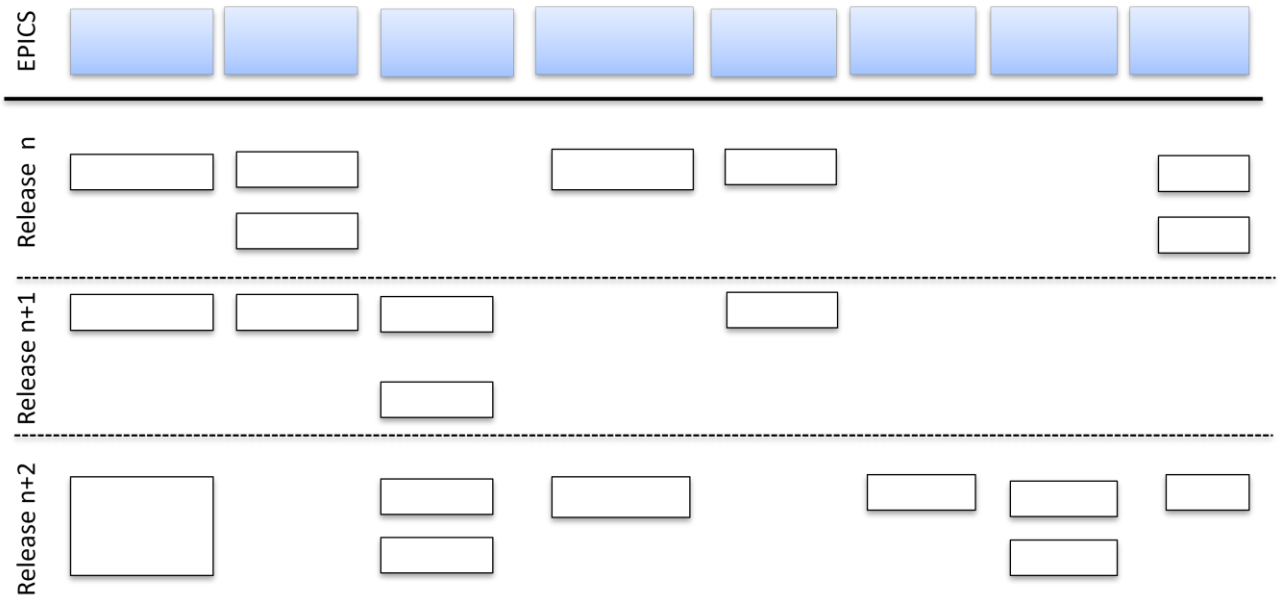

Abbildung 45: Story-Maps mit Release-Einblendung

Die Verwendung von Storys und Epics zur Darstellung der Produkt-Roadmap hat mehrere Nachteile. Für die meisten Geschäfts-Stakeholder ist es schwer zu verstehen, wie sich das Produkt als Ganzes entwickelt, da zu viele Details enthalten sind. Außerdem sind diese Roadmaps anfällig für Änderungen und müssen regelmäßig aktualisiert werden, was zeitaufwändig ist.

Abbildung 46 zeigt eine Roadmap, die nicht nur die geplanten Iterationen enthält, sondern auch auf der vertikalen Achse eine Zuordnung der Backlog Items zu mehreren Teams, wie in Kapitel 6.2 erläutert. Program Boards sind feingranulare Delivery Roadmaps die in SAFe während des „Program Increment Planning" verwendet werden. Sie enthalten die Sprache der Entwickler, ausgedrückt durch Backlog Items.

Das Board stellt die zu implementierenden Features dar (F1...F4). Die Features werden in Backlog Items unterteilt, die hier farblich gekennzeichnet sind. Ihre Reihenfolge wird durch die Nummer angegeben. Das Board wird verwendet, um kritische teamübergreifende Abhängigkeiten zwischen den Work Items zu identifizieren, wie durch die Pfeile angezeigt.

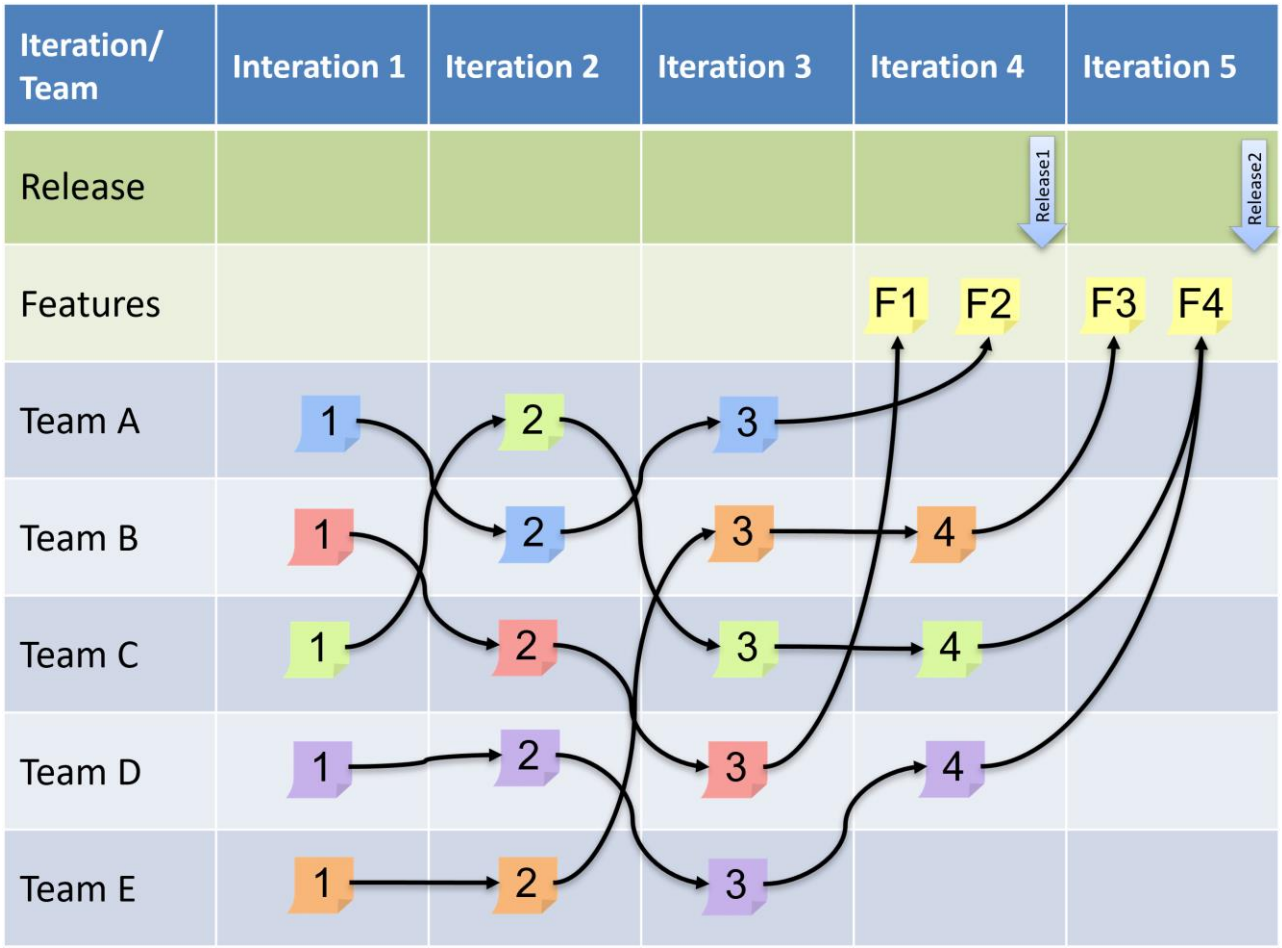

Abbildung 46: Eine Roadmap mit expliziten Abhängigkeiten

Wenn sich Ihre Teams am selben Standort befinden, könnten Sie Ihre Roadmap an der Wand oder an einer Pinnwand präsentieren und bearbeiten. Wenn Sie mit verteilten Teams arbeiten müssen, finden Sie Dutzende von Roadmapping-Tools zur Unterstützung der visuellen Planung mehrerer Releases, von denen viele mehr oder weniger gut mit den Tools zur Verwaltung des Backlogs harmonieren.

Im Gegensatz zu SAFe schlagen andere Frameworks wie Less und Nexus keine spezifische Verwendung von Roadmaps vor. Das bedeutet nicht, dass Roadmaps innerhalb dieser Frameworks nicht verwendet werden können, sondern es liegt an den Entwicklern zu entscheiden, ob eine Roadmap erforderlich ist und welche Art von Roadmap die Planungs- und Integrationsarbeit am besten unterstützt.

6.3.2 Synchronisierung von Teams mit Roadmaps

Agile Entwicklung konzentriert sich auf kurze Iterationen mit schnellen Feedback-Zyklen, so dass die ideale Situation eine ist, in der das Produkt in enger Zusammenarbeit kleiner Gruppen in einem kurzen Rhythmus entwickelt werden kann.

Wichtig ist auch, dass ein *regelmäßiger Rhythmus* für die Entwicklungs-Iterationen und Releases festgelegt wird [DeMarco et al. 2011]. Unregelmäßige Zyklen irritieren das Team, erschweren die Planung und gestalten auch die Nachverfolgung der Arbeitsgeschwindigkeit der Entwickler schwieriger.

Diesen Rhythmus bezeichnet man auch als Kadenz. In der Musik ist eine Kadenz eine melodische Struktur, die ein Gefühl der Auflösung oder Endgültigkeit vermittelt. Bei der Softwareentwicklung wird dieses Gefühl der Auflösung auf mehreren Abstraktionsebenen erzeugt: Unter den Entwicklern durch tägliche Standup-Meetings, für die Entwickler als Ganzes bei der Übergabe an den Product Owner am Ende einer Sprint-Iteration und möglicherweise für die skalierte Entwicklungsorganisation bei der Erstellung eines auslieferungsfähigen Produktinkrements für jeden Release-Zyklus.

Wenn Sie nur ein Team haben, können Sie nach jeder Iteration ein neues Produktinkrement liefern, ohne sich mit anderen Teams abzustimmen. Daher ist neben der Iterationskadenz (in Scrum: die Länge des Sprints) keine andere Kadenz nötig. Wenn mehrere Teams an demselben Produkt arbeiten, müssen Sie alle Teamergebnisse in ein neues Produktinkrement integrieren. Da die End-to-End-Tests und die Arbeit, die erforderlich ist, um alle Ergebnisse in einer Version zusammenzufassen, einen gewissen zusätzlichen Aufwand bedeuten, kann eine zusätzliche Kadenz für Kundenversionen eingeführt werden.

In diesem Sinn lässt sich eine umfangreiche agile Organisation mit einem großen Orchester vergleichen, das komplexe Musik spielt. In einer gut funktionierenden, großen agilen Organisation herrscht eine gewisse Harmonie. Wenn die Organisation nicht gut funktioniert, dann kommt diese Harmonie nicht zum Vorschein, genau wie bei einem Orchester, das nicht gut spielt. Wenn Sie mit mehreren Teams arbeiten müssen, müssen die Iterationen für jedes Team nicht gleich lang sein, aber die Zyklen sollten in dem Sinne kompatibel sein, dass sie auf der Ebene der größeren Kadenz synchronisiert werden können. So können einzelne Teams zum Beispiel eine Sprintlänge von zwei oder vier Wochen innerhalb eines vier- (oder acht-) wöchigen Release-Zyklus wählen (siehe Abbildung 47).

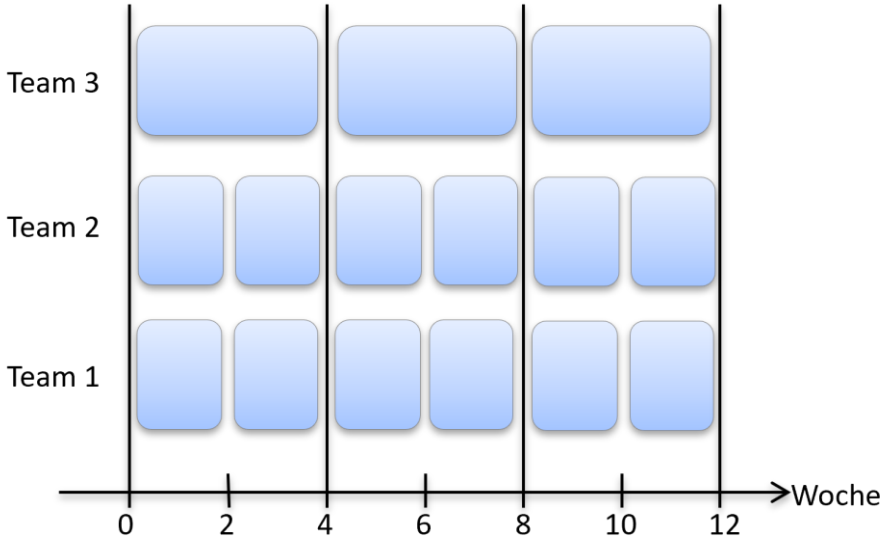

Abbildung 47: Verschiedene, aber kompatible Iterationszeiten

Manuelle Integration und Tests führen wahrscheinlich zu längeren Release-Zyklen. Automatisierung kann helfen, die Release-Zyklen zu verkürzen: Kontinuierliche Integrationsansätze und kontinuierliche Bereitstellungsfunktionen können es Teams ermöglichen, Features in kürzeren Zyklen bereitzustellen.

6.3.3 Entwicklung von Roadmaps

In der groß angelegten Produktentwicklung wird die Anforderungsarbeit von verschiedenen Product Owner-Rollen auf der Grundlage einer Anforderungshierarchie durchgeführt, wie in Kapitel 6.2.1 erläutert. Auch die Zuständigkeiten in Bezug auf die Roadmaps werden auf jeder Hierarchieebene unterschiedlich sein. Auf einer höheren Ebene können Product Owner beispielsweise für die Produkt-Roadmap verantwortlich sein, während sie sich auf einer niedrigeren Ebene eher auf die Delivery-Roadmap konzentrieren.

Um eine langfristige *Produkt-Roadmap* zu entwickeln, muss ein Product Owner zunächst eine Produktvision und eine Strategie definieren (s. Kapitel 2). Dies ist notwendig, damit die richtigen Stakeholder für die Arbeit an der Produkt-Roadmap gewonnen werden können (Stakeholder-Management).

Nach der Festlegung einer Produktvision und -strategie müssen die Product Owner dann die grobgranularen Anforderungen ermitteln (siehe Kapitel 3), indem sie sich mit den erforderlichen Stakeholdern auseinandersetzen. Es ist nicht nötig, an dieser Stelle Zeit in detaillierte Anforderungen zu investieren. Später, bei der Verfeinerung des Backlogs, werden weitere Details entdeckt.

Um volle Unterstützung für die Produktentwicklung zu erhalten, müssen die verschiedenen Stakeholder frühzeitig einbezogen werden und sollten die Geschäftsziele des Produkts verstehen. Die Produkt-Roadmap sollte daher auf ihre besonderen Interessen und ihren Informationsbedarf zugeschnitten sein und regelmäßig mit ihnen geteilt und validiert werden. Häufige Stakeholder sind z. B. die Geschäftsleitung und das Senior Management, Vertrieb und Marketing sowie die Entwickler.

Product Owner weisen grobgranulare Anforderungen über einen breiten Planungshorizont zu und zeigen gleichzeitig strategische Ziele auf der Zeitachse auf. In einer ersten Produkt-Roadmap sollten Product Owner harte Deadlines vermeiden. Stattdessen sollten die Features auf Monats- oder Quartalsebene geplant werden. Mit zunehmender Reife der Produktentwicklung können konkrete Termine und Deadlines hinzugefügt werden.

Um eine mittelfristige *Delivery-Roadmap* zu erstellen, müssen die Product Owner die Backlog Items aus der bestehenden Produkt-Roadmap verfeinern. Diese Items müssen von den Entwicklern grob geschätzt werden, auch wenn die Schätzungen in diesem Stadium noch ungenau sind (z. B. T-Shirt-Größen). Die Schätzung muss nur gut genug sein, um einen Überblick über die kommenden Iterationen zu ermöglichen.

Unsere praktischen Erfahrungen haben gezeigt, dass sich bei den meisten umfangreichen Schätzungen die Fehler der einzelnen individuellen Schätzungen gegenseitig aufheben. Die Gesamtschätzung ist also in der Regel hinreichend korrekt, selbst wenn die individuellen Schätzungen nicht korrekt sind.

In Kapitel 3 haben wir Schätztechniken für Backlog Items besprochen. Diese Techniken können Sie auch für längerfristige Schätzungen und Planungen einsetzen. Diese Schätztätigkeit geht über den Umfang des herkömmlichen Requirements Engineering hinaus, doch sie gewinnt im Kontext von RE@Agile an Bedeutung, da hier die Anforderungsarbeit Hand in Hand mit der Planung geht. Viel mehr zu diesem Thema finden Sie unter [Cohn2006].

Das Erstellen und Aktualisieren von *Delivery-Roadmaps* geschieht typischerweise bei Face-to-Face Planungsrunden, die als Big Room Planning (oder PI Planning in SAFe) bekannt sind und in regelmäßigen Abständen stattfinden. Bei solchen Veranstaltungen planen, schätzen und priorisieren die Entwickler die Features gemeinsam. Die Product Owner bereiten die Backlog Items im Vorfeld vor und richten sie an der Vision sowie an der bestehenden Produkt-Roadmap aus. Die Teams arbeiten zusammen, um die wichtigen Risiken und Abhängigkeiten zu ermitteln. Die *Delivery-Roadmap* wird aktualisiert, um die verfeinerten Backlog Items, die Abhängigkeiten zwischen ihnen und ihre Übereinstimmung mit der Produktvision zu zeigen.

6.3.4 Validierung von Roadmaps

Die *Produkt-Roadmap* sollte auch aus der Sicht des Unternehmens sowie des Fachbereichs überprüft werden: Kundenfeedback, Marktveränderungen, neue Ideen und Markttrends sowie ähnliche Produkte, die auf den Markt kommen, sollten berücksichtigt werden. Zu diesem Zweck ist das MMP (wie in Kapitel 5.5 vorgestellt) ein guter Ausgangspunkt. Die Validierungsintervalle hängen von der Stabilität des Marktes ab: In einem hochdynamischen Markt sollte die Produkt-Roadmap beispielsweise mindestens monatlich überprüft werden, andernfalls können vierteljährliche Intervalle ausreichend sein. Die wichtigsten Stakeholder sollten in die Entwicklung der Roadmap einbezogen werden, um die Akzeptanz zu erhöhen und Änderungen zu kommunizieren.

Um den Cone of Uncertainty zu verkleinern, sollten auch die *Delivery-Roadmaps* regelmäßig aktualisiert werden, entweder auf der Grundlage des Feedbacks der Stakeholder zu integrierten Produktinkrementen (siehe MVP in Kapitel 5.5) oder auf der Grundlage der Ergebnisse von Prototypen.

Die Validierungsintervalle hängen vom Reifegrad der Produktentwicklung und von Änderungen der Produkt-Roadmap ab. In einem ausgereiften Entwicklungsprozess, in dem erfahrene Entwickler bereits seit einiger Zeit gemeinsam an einem Produkt arbeiten, muss die Delivery-Roadmap möglicherweise erst nach einem Release überprüft werden. Zu Beginn der Produktentwicklung sollte die Delivery-Roadmap nach der Integration des ersten Produktinkrements validiert werden. Die Validierung von Delivery-Roadmaps kann in die oben beschriebenen regelmäßigen Planungsereignisse einbezogen werden.

6.4 Produkt-Validierung

Ein Grundgedanke der agilen Entwicklung ist es, einen kleinen Teil des Produkts zu entwickeln, durch die Einbindung von Stakeholdern Feedback zu generieren und die Produktentwicklung entsprechend der Erkenntnisse und Ergebnisse anzupassen. Nach dem Prinzip des Build-Measure-Learn-Zyklus [Ries2011] ist die Produktvalidierung ein wichtiger Schritt, um schnelles Feedback zu erhalten. Jedes Mal, wenn eine neue Produktversion freigegeben wird, verwenden die Product Owner dieses Produktinkrement, um seinen Geschäftswert zu prüfen und um festzustellen, ob die Produktanforderungen richtig verstanden wurden.

Die Validierung auf Produktebene ist eine wichtige Methode in der groß angelegten Produktentwicklung, da sie sicherstellt, dass die Product Owner gemeinsam die volle Verantwortung von den Geschäftsanforderungen bis zur Produktintegration tragen. Es ist das gesamte Produkt, das für die Stakeholder einen Wert hat, nicht nur kleine Teile des Produkts.

In Scrum ist ein Sprint-Review eine geeignete Maßnahme, um ein Produktinkrement den relevanten Stakeholdern zu präsentieren. In der groß angelegten Produktentwicklung kann eine ähnliche Idee angewandt werden: Anstatt einen einzelnen, von einem Team entwickelten Produkt-Teil zu überprüfen, werden alle Teamergebnisse zu einem funktionierenden Produktinkrement integriert, das es wert ist, validiert zu werden. Das Produktinkrement wird in einem *Produktreview (Demonstration)* vorgeführt, in dem die End-to-End-Features vorgestellt werden. So erhalten die Stakeholder einen besseren Eindruck vom gesamten Produkt [SAFe1], [Larman2017], [LeSS].

Zur Koordinierung der Integrationsarbeiten, die die Grundlage für die Validierung auf Produktebene bilden, kann eine Delivery-Roadmap mit Meilensteinen für die Freigabe verwendet werden, um die Teams zu synchronisieren (siehe Kapitel 6.2.3).

Die Herausforderungen müssen in der groß angelegten Produktentwicklung (wie in Kapitel 6.1 erwähnt) auch bei der Validierung auf Produktebene berücksichtigt werden. Das bedeutet, dass Sie eine große Anzahl von Stakeholdern und Nutzern effektiv einbeziehen und deren Feedback an die Entwickler weitergeben müssen. Darüber hinaus müssen Sie ein Gesamtverständnis des integrierten Produkts erreichen, indem Sie die verschiedenen Perspektiven und Kenntnisse der Stakeholder berücksichtigen.

Wenn viele Personen an einem umfangreichen Produktreview beteiligt sind, ist es sehr wichtig, in den Diskussionen das richtige Maß an Details zu finden, um das Interesse aller Teilnehmer zu erhalten. Ein Ansatz ist die Verwendung eines „Diverge-and-Converge"-Kollaborationsmusters [Design Council]. Im abweichenden Teil des Reviews wird der Raum in mehrere Bereiche aufgeteilt, in denen die Teams verschiedene Features des Produktinkrements demonstrieren. Wie auf einem Basar gehen die Leute herum, besuchen die Demonstrationen, die sie interessieren, und geben dem entsprechenden Team Feedback. Anschließend kommen die Teilnehmer im konvergenten Teil des Reviews zusammen, um ihre Ergebnisse zusammenzufassen, wichtige Aspekte zu diskutieren und neue Ideen auszutauschen.

Produkt-Reviews sind in mehreren Skalierungs-Frameworks enthalten. In Nexus und Less wird das Review-Meeting als *Sprint Review* bezeichnet. In SAFe wird es als *Systemdemo* bezeichnet. Nach dem Nexus-Leitfaden sollte für das Review als Faustregel ein Zeitrahmen von etwa vier Stunden für einen einmonatigen Sprint angesetzt werden.

Ein weiterer Ansatz für die Produktvalidierung in der umfangreichen Produktentwicklung basiert auf *Datenanalyse* [Maalej et al.2016]. Das integrierte Produktinkrement wird an Benutzer ausgeliefert und anhand deren Verhalten wird gemessen, ob die Produkt-Features einen positiven, neutralen oder negativen Einfluss haben. Datenanalyse-Frameworks werden in der Regel verwendet, um Feedbackdaten systematisch zu analysieren. Beispielsweise können Product Owner die Ergebnisse nutzen, um potenziell schlecht konzipierte Features zu identifizieren. Um die identifizierten Probleme besser zu verstehen, müssen sie möglicherweise erneut die üblichen Techniken zur Anforderungserhebung und -analyse anwenden.

Wie auch immer das Feedback der Stakeholder gesammelt wurde, die Product Owner passen die bestehenden Backlog Items an, setzen neue Prioritäten und fügen bei Bedarf neue Items hinzu. Einige Items können aus dem Backlog gestrichen werden, wenn sich bei der Produktvalidierung gezeigt hat, dass die entsprechenden Features nicht den beabsichtigten Wert schaffen. Änderungen am Product Backlog können wiederum Änderungen an der Produkt- und Delivery-Roadmap auslösen, wie in Kapitel 6.3.4 beschrieben.

Abkürzungsverzeichnis

DSDM	Dynamische Systementwicklungsmethode
DoD	Definition of Done
DoR	Definition of Ready
LeSS	Large-Scale Scrum (https://less.works)
MVP	Minimum Marketable Product
MVP	Minimum Viable Product
PO	Product Owner, Produkteigner
RE	Requirements Engineering
ROI	Return on Investment, Investitionsrendite
SAFe	Scaled Agile Framework (www.scaledagileframework.com)
WSJF	Weighted Shortest Job First

Literaturverzeichnis

[AgileAlliance] Glossary of the Agile Alliance: Definition of term "Definition of Ready": https://www.agilealliance.org/glossary/definition-of-ready, zuletzt besucht im Mai 2021.

[AgileManifestoPrinciples] https://agilemanifesto.org/principles.html, Zuletzt besucht im Mai 2021

[Alexander2005] Alexander, I. F.: A Taxonomy of Stakeholders – Human Roles in System Development. International Journal of Technology and Human Interaction, Vol 1, 1, 2005, pages 23-59.

[AmLi2012] Ambler, S., Lines, M.: Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise, IBM Press, 2012

[Anderson2020] Anderson, J.: Agile Organisationsgestaltung - Growing Self-Organizing Structure at Scale. Leanpub, 2020.

[Beck2002] Beck, K.: Test Driven Development: By Example. Addison Wesley, 2002.

[BiKo2018] Bittner, K.; Kong, P.; West, D.: Mit dem Nexus™ Framework Scrum skalieren: Kontinuierliche Bereitstellung eines integrierten Produkts mit mehreren Scrum-Teams, Dpunkt.Verlag, 2018

[Boehm 1981] Boehm Barry W.: Software Engineering Economics Veröffentlicht 1981 von Prentice Hall

[BOSSANOVA] https://www.agilebossanova.com/#bossanova, zuletzt besucht im Mai 2021

[ClBa1994] Clegg, D.; Barker, R. (09.11.2004). Case Method Fast-Track: A RAD Approach. Addison-Wesley.

[Clements et al.2001] P. Clements et al.: Evaluating Software Architectures, SEI Series in Software Engineering, 2001

[Cohn2010] Cohn, M.: User Storys für die agile Software-Entwicklung mit Scrum, XP u.a., mitp, 2010

[Cohn2006] Cohn, M.: Agile Estimation and Planning, Addison Wesley, 2006

[Conway1968] Conway, Melvin E.: How Do Committees Invent? Datamation Magazine, 1968. http://www.melconway.com/Home/Committees_Paper.html. Zuletzt besucht im Mai 2021.

[Cooper2004] Cooper, A.: The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity

[DeMaLi2003] DeMarco, T.; Lister, T.: Bärentango: mit Risikomanagement Projekte zum Erfolg führen, Hanser, 2003

[DeMarco et al. 2011]: DeMarco, T.; Hruschka, P. Lister, T.; McMenamin, S.; Robertson, J+S.: Adrenalin-Junkies und Formular-Zombies : Typisches Verhalten in Projekten, Kapitel 31: Rhythmus, Carl Hanser Verlag, 2011

[Design Council] A study of the design process; https://www.designcouncil.org.uk/sites/default/files/asset/document/ElevenLessons_Design_Council%20(2).pdf. Zuletzt besucht im Mai 2021

[Doran1981] Doran, G. T: There's a S.M.A.R.T. way to write management's goals and objectives, Management Review. AMA FORUM. 70 (11): 35–36 1981.

[Glinz2014] Glinz, M.: A Glossary of Requirements Engineering Terminology. Standard Glossary for the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version 2.0, 2020. https://www.ireb.org/de/downloads/#cpre-glossary. Zuletzt besucht im Mai 2021.

[HaHP 2003] Hatley, D., Hruschka, P., Pirbhai, I.: Komplexe Softwaresysteme beherrschen : Requirements verstehen - Architekturen konzipieren, mitp, 2000

[HeHe2011] Heath, C., Heath, D.: Switch: Veränderungen wagen und dadurch gewinnen. Crown Business, 2010

[Highsmith2001] Highsmith, J.: Design the Box. *Agile Project Management E-Mail Advisor 2001,* http://www.joelonsoftware.com/articles/JimHighsmithonProductVisi.html. Zuletzt besucht im Mai 2021.

[Hruschka2017] http://www.b-agile.de/Resources/Story-Splitting. Zuletzt besucht im Mai 2021.

[ISO25000] ISO/IEC 25000:2014: Systems and software engineering: System and Software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE: https://www.iso.org/standard/64764.html. Zuletzt besucht im Oktober 2018.

[ISO25010] ISO/IEC 25010:2011: Systems and software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: https://www.iso.org/standard/35733.html. Zuletzt besucht im Mai 2021.

[ISO25012] ISO/IEC 25012:2008: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Data quality model: https://www.iso.org/standard/35736.html. Zuletzt besucht im Mai 2021.

[Jacobson1992] Jacobson, I. Object-oriented Software Engineering - A Use-Case Driven Approach, ACM Press, 1992

[Jacobson2011] https://www.ivarjacobson.com/publications/white-papers/use-case-ebook. Zuletzt besucht im Mai 2021

[HaCh1998] Hammer, M., Champy, J.: Business Reengineering: Die Radikalkur für das Unternehmen. Harper, 1993

[Jeffries2001] Jeffries, R.: Essential XP: Card, Conversation, Confirmation, 2001, https://ronjeffries.com/xprog/articles/expcardconversationconfirmation/ Zuletzt besucht im Mai 2021.

[Kahneman2016] Kahneman D.: Schnelles Denken, langsames Denken Penguin Verlag, 2016.

[KnLe2017] Knaster, R.; Leffingwell, D.: SAFe 4.0 Distilled, Addison Wesley, 2017

[Kniberg] Kniberg, H.: Scaling Agile @ Spotify with Henrik Kniberg https://www.youtube.com/watch?reload=9&v=jyZEikKWhAU&feature=youtu.be ,and https://www.youtube.com/watch?v=4GK1NDTWbkY&t=156s. Zuletzt besucht im Mai 2021

[Larman2017] Larman, C.: Large-Scale Scrum: Scrum erfolgreich skalieren mit LeSS, dpunkt.verlag, 2017

[Lawrence1] Lawrence, R: How to Split a User Story http://agileforall.com/resources/how-to-split-a-user-story. Zuletzt besucht im Mai 2021

[Lawrence2] Lawrence, R: Why Most People Split Workflows Wronghttp://agileforall.com/why-most-people-split-workflows-wrong/. Zuletzt besucht im Mai 2021

[Leffingwell2007] Leffingwell, D.: Scaling Software Agility – Best Practices for Large Enterprises, Addison Wesley, 2007

[Leffingwell2010] Leffingwell, D.: Agile Software Requirements – Lean Requirements Practices for Teams, Programs, and the Enterprise, Addison Wesley, 2010

[Leffingwell2017] Leffingwell, D. et al.: SAFe Reference Guide, Scaled Agile, Inc. 2017

[LeSS] Large-Scale Scrum: https://less.works Zuletzt besucht Mai 2021

[Maalej et al.2016] Maalej, W., Nayebi, M., Johann T., Ruhe, G.: Toward Data-Driven Requirements Engineering. IEEE Software (Volume 33, Issue 1), 2016

[MaKO2016] Maher, R., Kong, P.: Cross-Team Refinement in Nexus, https://www.scrum.org/resources/cross-team-refinement-nexus. Zuletzt besucht im Mai 2021

[McPa1988] McMenamin, S., Palmer, J: Strukturierte Systemanalyse, Hanser - Prentice Hall-Internat., 1988

[Meyer2014] Meyer, B.: Agile! The Good, the Hype and the Ugly, Springer, 2014.

[Nexus Guide] https://www.scrum.org/resources/nexus-guide. Zuletzt besucht im Mai 2021

[OsPi2011] Osterwald, A., Pigneur, Y.: Business Model Generation – Ein Handbuch für Visionäre, Spielveränderer und Herausforderer. Campus Verlag, 2011

[Patton2015] Patton, J.: User Story Mapping: Die Technik für besseres Nutzerverständnis in der agilen Produktentwicklung, O'Reilly, 2015

[Pichler2016]: Pichler, R.: Strategize – Product Strategy and Product Roadmap Practices for the Digital Age, Pichler Consulting 2016.

[Primer2017] CPRE RE@Agile Primer https://www.ireb.org/en/downloads/tag:re-agile-primer. Zuletzt besucht im Mai 2021

[Reinertsen2008] Reinertsen, D.: The Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas Publishing 2008

[Ries2011] Ries, E.: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, New York, NY 2011

[RoRo2013] Robertson S. Robertson J.: Mastering the Requirements Process – Getting Requirements Right, 3rd edition, Addison Wesley, 2013

[RoRo2017] Robertson S. Robertson J.: Volere Requirements Specification Template, https://www.volere.org/requirements-auditing-is-the-specification-fit-for-its-purpose/. Zuletzt besucht im Mai 2021

[Robertson2003] Robertson, S.: Stakeholders, Goals, Scope: The Foundation for Requirements and Business Models, 2003, https://www.volere.org/wp-content/uploads/2018/12/StkGoalsScope.pdf. Zuletzt besucht im Juni 2021

[SAFe1] https://www.scaledagileframework.com/roadmap/. Zuletzt besucht im Mai 2021

[SAFe2] https://www.scaledagileframework.com/pi-planning/. Zuletzt besucht im Mai 2021

[SAFeMDM] https://www.scaledagileframework.com/safe-requirements-model/. Zuletzt besucht im Mai 2021

[S@S Guide] Sutherland, J. and Scrum, Inc: Scrum@Scale Guide: https://www.scrumatscale.com/scrum-at-scale-guide/, zuletzt besucht im Mai 2021

[SOZIOKRATIE] https://sociocracy30.org. Zuletzt besucht im Februar 2022

[SofS] https://scrumguide.de/scrum-of-scrums/. Zuletzt besucht im Mai 2021

[Spotify2012] https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf

[Wake2003] Wake, B: INVEST in Good Stories, and SMART Tasks, 2003, https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Zuletzt besucht im Mai 2021

[WyHT2017] Wynne, M., Hellesøy, A., Tooke, S.: The Cucumber Book - Behaviour-Driven Development for Testers and Developers, The Pragmatic Programmers, 2017

[Yakima 2016] Yakima, A.: The Rollout – A Novel about Leadership and Building a Lean-Agile Enterprise with SAFe