

2 Requirements Engineering – kurz und knapp

Ihr Nutzen aus diesem Kapitel:

»It isn't that they can't see the solution. It is that they can't see the problem.« Der berühmte Autor Gilbert Keith Chesterton adressierte damit eine wesentliche Herausforderung. Requirements Engineering identifiziert systematisch Probleme und Ziele – bevor Lösungen vorschnell entwickelt werden. Was sollten Sie für die eigene Praxis direkt übernehmen? Welche Faustregeln helfen, um das Requirements Engineering richtig zu justieren? Worauf sollten Sie in Ihren Projekten achten? In diesem Kapitel fasse ich kurz die wichtigsten Gesetzmäßigkeiten des Requirements Engineering zusammen. Systematik heißt nicht Formalismus oder gar Dogmatismus, sondern muss sich pragmatisch auf vorliegende Probleme einstellen. Zunehmend arbeiten wir im RE agil, also flexibel und situativ. Daher zeige ich hier die Essenz des Requirements Engineering. Sie lernen, was Anforderungen sind, wie verschiedene Typen von Anforderungen ganz verschieden auf das Projekt wirken und wie Sie Requirements Engineering leben können. Ein durchgängiges Beispiel transportiert das Vorgehen immer wieder in der Praxis.

2.1 Was ist eine Anforderung?

Eine Anforderung beschreibt, was der Kunde oder Benutzer vom Produkt erwartet, also Bedingungen, Attribute, Ziele und vor allem Nutzen. Anforderungen sind definiert als:

- Eine Eigenschaft oder Bedingung, die von einem Benutzer (Person oder System) zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.
- Eine Eigenschaft oder Bedingung, die ein System oder eine Systemkomponente erfüllen muss, um einen Vertrag, eine Norm oder andere, formell vorgegebene Dokumente zu erfüllen.
- Eine dokumentierte Repräsentation einer Eigenschaft oder Bedingung wie in den ersten beiden Punkten beschrieben.

Wenn wir hier von **Produkt** sprechen, umfasst dies Anwendungen, IT-Systeme, eingebettete Software bis hin zu großen IT-Lösungen. Auch Dienstleistungen sind

Produkte. Die **Benutzer** oder Anwender des Produkts sind diejenigen, die nach der Auslieferung damit in irgendeiner Form in Berührung kommen. Wir wollen dieses »in Berührung kommen« später nochmals aufgreifen (siehe Kap. 3), denn es ist bei der Ermittlung von Anforderungen wichtig, hier die Basis nicht zu sehr einzuschränken. Beispielsweise ist es relevant, zu unterscheiden, wer exakt **Kunde** ist (also vertraglich in die Entstehung eingebunden ist und dafür bezahlt) und wer das Produkt später nutzt.

Wert ist, wofür ein Kunde zahlt, und keine lange Funktionsliste. RE fokussiert Anforderungen auf den zu liefernden Wert und erreichbare Ziele. Anforderungen können mehrdeutig sein, sie können überspezifiziert sein, sie können unvollständig sein, sie können kontextspezifisch sein, sie können sich widersprechen, sie können nicht machbar oder schlichtweg falsch sein. In aller Regel jedoch sind es zu viele, um unter gegebenen Randbedingungen realisiert werden zu können. Das kommt deutlich am Beispiel eines Wunschzettels zum Ausdruck (Abb. 2–1).

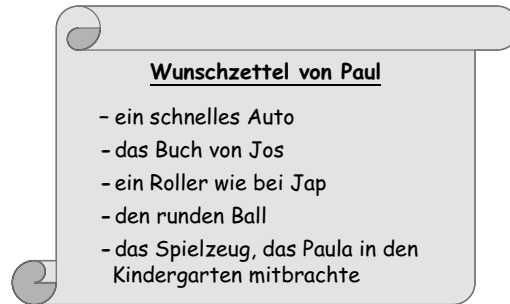


Abb. 2–1 Anforderungen sind der »Wunschzettel« des Kunden.

Das Problem in vielen Unternehmen ist, dass zu oft Funktionen und zu selten Träume adressiert werden. Als Ingenieure sind wir darauf geeicht, Lösungen zu finden. Wir definieren Funktionen und implementieren sie. Allerdings führen solche – angenommenen – Lösungen nicht immer zum Markterfolg und zu zufriedenen Kunden. Das überrascht uns, wo doch die Lösung so viele interessante Funktionen hat. Aber hatten wir wirklich ein Problem und einen Bedarf adressiert? Werden durch unser Produkt eine Vision und ein Traum wahr oder ersticken die Benutzer in Komplexität?

Wir stürzen uns viel zu schnell auf eine Lösung, weil es das ist, was wir dank Ausbildung und unter Projektdruck als Ergebnis sehen wollen. Ein Projekt, das von einer – angenommenen – Lösung aus startet, führt dazu, dass man einer Fata Morgana nachläuft, die sich ständig ändert. Wenn es uns gelingt, die Ziele und Anforderungen zu verstehen und systematisch umzusetzen, dann können wir jedes Projekt beherrschen.

Ein Produkt ist dann erfolgreich, wenn es den Bedürfnissen seiner Benutzer und seiner Umgebung gerecht wird. Anforderungen kommunizieren diese Bedürfnisse,

und Requirements Engineering ist die Disziplin, die die Behandlung von Anforderungen über den gesamten Lebenszyklus des Produkts hinweg umfasst.



Beispiel:

Apple unter Steve Jobs zeigte, wie man mit wertorientiertem Requirements Engineering hervorragende Produkte macht. Steve Jobs gelang es, Träume zu verkaufen und diese Träume in Funktionen zu übersetzen. Er schockierte seine Ingenieure regelmäßig mit der einfachen Frage: Kann man im Design noch etwas weglassen? Unnötige Funktionen sind Ballast und verursachen Kosten im gesamten Lebenszyklus. Ein Produkt war für Steve Jobs erst gut genug, wenn jede Funktion einen Wert lieferte – und die Komplexität auf ein Minimum reduziert war.

Wir trennen daher klar zwischen **Anforderung** und **Lösung**. Eine Anforderung beschreibt ein Bedürfnis oder einen Nutzen, der erreicht werden soll. Sie beschreibt nicht, wie dieser Nutzen zu realisieren ist. Diese Implementierungssicht wird durch die Lösung beschrieben. Abbildung 2–2 veranschaulicht diesen Unterschied durch die Trennung zwischen Problemraum (oberer Teil: Marktanforderungen, Lastenheft etc.) und Lösungsraum (unterer Teil: Lösungsspezifikation, Pflichtenheft, Design, Fachkonzept etc.). Der Problemraum ist zunächst immer nur unscharf umrissen und wird im Verlauf der Lösungskonzeption eingeschränkt. Wert existiert nur im Auge des Betrachters. Daher sind Lösungen oftmals am erfolgreichsten, wenn sie nicht alle Anforderungen des Kunden adressieren. Der Erfinder und Unternehmer Henry Ford hat das früh begriffen und festgestellt: »Wenn ich die Leute gefragt hätte, was sie wollen, hätten sie gesagt: schnellere Pferde.« Steve Jobs hat Apple damit erfolgreich gemacht, dass er innovative Lösungen lieferte und Altlasten rigoros hinterfragt und reduziert hat. Legendar sind die frühen iPhone-Telefone, die zwar kaum telefonieren konnten, aber sich hervorragend verkauften, weil sie ganz andere neue Bedürfnisse adressierten.

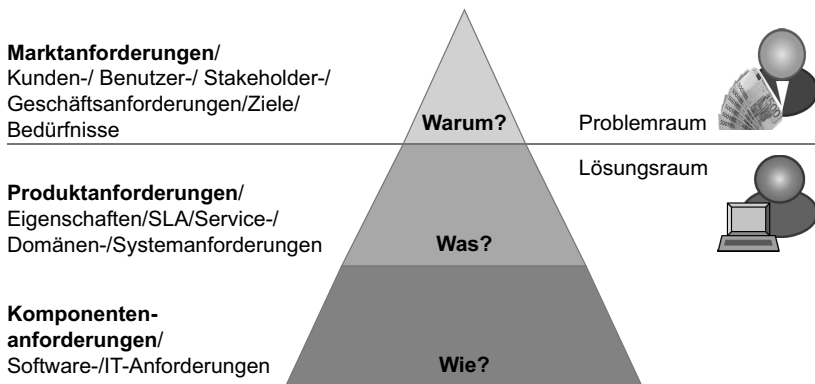


Abb. 2–2 Anforderungen und Lösungen

Es gibt nicht die »Anforderung« schlechthin. Zu einer Anforderung gehört immer die Perspektive, aus der sie beschrieben wird. Eine Anforderung ist eine Bedingung oder eine Fähigkeit, die ein Benutzer benötigt, um ein Problem zu lösen oder um ein Ziel zu erreichen. Das heißt, sie hängt von der Perspektive ab. Ein Benutzer kann der Kunde sein, der für die Lösung bezahlt, aber es kann auch ein Entwickler sein, der daraus eine Architektur ableitet. Entsprechend unterschiedlich sind die Schwerpunkte und Inhalte, die durch diese Anforderung beschrieben werden. Was dem einen die Anforderung ist, ist dem anderen die Lösung. Man trennt daher in der Praxis unterschiedliche Arten von »Anforderungen«, beispielsweise Marktanforderungen oder Komponentenanforderungen, und vermeidet, von einer »Anforderung« ohne Präzisierung zu sprechen.

2.2 Perspektiven: vom Markt zur Realisierung

Drei verschiedene Sichten auf Anforderungen werden im Laufe der Lösungskonzeption unterschieden (Abb. 2–2):

- Marktanforderungen
- Produktanforderungen
- Komponentenanforderungen

Diese drei Sichten entstehen durch Verfeinerung bzw. Abstraktion. Offensichtlich ist diese Dreiteilung rekursiv: Eine Komponentenanforderung an einen Lieferanten ist dort wiederum eine Marktanforderung.

Marktanforderungen

Marktanforderungen beschreiben Anforderungen an ein Produkt aus der Sicht des Kunden. Sie werden daher oft auch als Kunden-, Benutzer- oder Geschäftsanforderungen oder als Bedürfnisse bezeichnet. Sie beschreiben den Nutzen und die Erfahrungen mit dem Produkt in der Sprache des Kunden oder Benutzers, also **warum** ein Projekt überhaupt durchgeführt wird. Einziger Maßstab an Wert und Erfüllungsgrad ist daher die Wahrnehmung oder Spezifikation des Kunden. Marktanforderungen werden im Lastenheft dokumentiert.



Beispiel:

Marktanforderung_1:

Der Datentransfer muss geschützt erfolgen, um Missbrauch zu verhindern.

Viele Projekte umfassen Änderungen an Bestehendem. **Marktanforderungen adressieren gerade auch solche Änderungen und nicht nur Projekte und Produkte auf der »grünen Wiese«.** Änderungen verlangen eine genaue Abstimmung der Bedürf-

nisse und Nutzen mit den jeweiligen Zielgruppen oder Marktsegmenten. Häufig realisieren wir Funktionen oder Änderungen, die interessant scheinen, deren Markt aber zu klein ist. Hier ist eine Priorisierung aus betriebswirtschaftlicher Sicht wichtig.

Marktanforderungen sind Bestandteil von Verträgen, Entwicklungsaufträgen, Projektplänen, Teststrategien etc. Sie dienen als Basis für Abschätzung, Planung, Durchführung und Verfolgbarkeit der Projektaktivitäten. Sie sind in der Sprache und im Kontext des Kunden formuliert. Wenn wir bei der Ermittlung der Marktanforderungen nicht aufpassen, haben wir die gleichen Schwierigkeiten, mit denen auch Eltern sich auseinandersetzen müssen, die einen Wunschzettel ihres Sprösslings in der Hand halten (Abb. 2–1). Es gibt Widersprüche, Inkonsistenzen und verborgene Prioritäten. Die Anforderungen sind zu umfangreich, und das Budget ist limitiert.

Marktanforderungen machen Wünsche erfüllbar und erlebbar. Das Ziel der Anforderungsermittlung ist es, aus verschiedenen Perspektiven möglicher Stakeholder und deren Vorgaben eine tragfähige Basis realisierbarer Anforderungen zu entwickeln. Abbildung 2–3 zeigt exemplarisch drei Benutzergruppen und deren Wünsche, Bedürfnisse und Geschäftsvorgaben als Punkte. Gutes RE schafft gemeinsam mit dem Produktmanagement einen Schnitt dergestalt, dass einige wesentliche Funktionen so ausgewählt werden, dass möglichst viel Wert erreicht wird, bei gleichzeitiger Optimierung der Kosten. Das erfordert Verhandlungsgeschick und Durchsetzungskraft, denn wir können es nicht jedem Beteiligten recht machen. Wesentlich ist, dass wir dabei klar trennen, was unrealistische Wünsche sind, die oftmals nur als Testballons platziert werden, was tatsächliche Bedürfnisse sind und was die Geschäftsvorgaben sind. Letztere sind Pflicht, Bedürfnisse werden priorisiert und gegenübergestellt, und Wünsche fallen in der Regel heraus, wenn kein Business Case nachvollziehbar ist. Wir sprechen im Requirements Engineering von einer Kundenbeziehung. Dabei kann der Kunde ein externer Benutzer sein oder eine interne Abteilung.

**Beispiel:**

Marktanforderungen im Consumer-Bereich, wie bei einer digitalen Kamera, gehen sehr konkret auf Benutzungsaspekte ein, also auf die Lebensdauer der Akkus oder die Pixelzahl und damit auf die Bildschärfe und -auflösung. Diese Anforderungen an die digitale Kamera werden im Unternehmen, das sie herstellt, in Produktanforderungen übersetzt, die dann die Sprache des internen Produktmarketings oder Produktmanagements sprechen. Schließlich werden sie in Anforderungen an Komponenten übersetzt und sprechen die Sprache von Entwicklungsingenieuren oder Einkäufern, die diese Komponenten entwickeln oder beschaffen. Änderungen der Anforderungen werden hinsichtlich ihres potenziellen Einflusses auf bestehende Pläne und Produkte abgeschätzt, geprüft und in



die bestehenden Anforderungen aufgenommen. Anforderungen werden durch das ganze Projekt hindurch kontrolliert, um ihren Status zu kennen und um beurteilen zu können, wie weit das Projekt – aus Kundensicht – fortgeschritten ist.

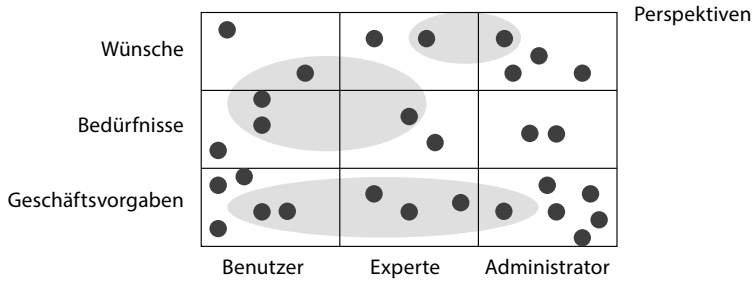


Abb. 2-3 Aus verschiedenen Perspektiven belastbare Geschäftsanforderungen entwickeln

Produktanforderungen

Produktanforderungen beschreiben Anforderungen an ein Produkt aus der Sicht der Realisierung einer späteren Lösung. Produkthanforderungen beschreiben, was verschiedene Benutzer mit dem Produkt machen können und wie Marktanforderungen und Kundenbedürfnisse in ein Produkt umgesetzt werden. Sie beschreiben eine Eigenschaft in der Sprache des Produkts und werden daher auch als Funktionen, Eigenschaften oder Systemanforderungen bezeichnet. Sie definieren den Lösungsraum und die Prioritäten. Produkthanforderungen werden im Pflichtenheft dokumentiert.



Beispiel:

Produkthanforderung_1:

Jede einzelne Transaktion zwischen baulich getrennten Komponenten wird individuell verschlüsselt.

Produkthanforderungen betrachten das Softwareprodukt oder den Dienst innerhalb eines größeren Kontextes, also der Umgebung, in der das System einmal arbeiten muss. Das kann ein PC sein, vor dem ein einzelner Benutzer sitzt (z.B. bei einem Computerspiel), eine Rechnerumgebung mit verschiedenen Benutzern (z.B. bei einem Ressourcenplanungssystem in einem Unternehmen), eine interaktive Online-Umgebung mit sehr vielen unbekannt Benutzern (z.B. bei einem Online-Buchungssystem), ein gemischtes Hardware-Software-System (z.B. bei einer Gebäudeautomatisierung) oder auch ein eingebettetes System, bei dem man kaum noch an Software denkt (z.B. ein Getränkeautomat, der in die Logistikkette eines Getränkelieferanten eingebaut ist).

Komponentenanforderungen

Komponentenanforderungen beschreiben Anforderungen an eine Komponente eines Produkts. Sie erläutern aus der Sicht der Realisierung und der späteren Lösung, *wie* Produkthanforderungen durch eine Komponente des Produkts (z.B. Benutzerschnittstelle, Betriebssystem) adressiert werden.

**Beispiel:****Komponentenanforderung_1:**

Der Datenaustausch an der externen Schnittstelle xyz wird mit 128 Bit PGP-verschlüsselt.

Komponentenanforderungen dienen zur rekursiven Verfeinerung einer Produktanforderung oder eines Systems in handhabbare Teile. Aus der Sicht eines Lieferanten, der diese Komponente liefert, ist dies wiederum eine Marktanforderung. Komponentenanforderungen werden wie die Produkthanforderungen auch im Pflichtenheft (in IT-Projekten oftmals auch Fachkonzept genannt) spezifiziert.

Viele Komponenten, wie Hardware und externe Software-Stacks, werden zugekauft. Das reduziert Kosten, verbessert die Qualität und erlaubt, sich auf den eigenen Wertbeitrag zu fokussieren. Daher müssen diese Anforderungen an externe Komponenten frühzeitig präzise spezifiziert werden, um danach die Entwicklungsarbeit verteilen und später die Ergebnisse integrieren zu können.

Die drei Sichten von Markt, Produkt und Komponenten sind nicht im Voraus oder von außen definiert, sondern hängen von der Lösungsstruktur ab.

**Beispiel:**

Der Benutzer oder Kunde stellt die folgende Anforderung:

M-Req-1: Das System verwaltet die Kundendaten im Format Name, Vorname, Adresse.

Der Requirements-Ingenieur spezifiziert dazu eine Lösung mit der folgenden Komponentenanforderung:

K-Req-1: Die Datenbank zur Verwaltung der Kundendaten wird mit Oracle 10 realisiert.

Offensichtlich sind dies zwei Anforderungen an die Datenhaltung, die aus verschiedenen Perspektiven beschrieben sind, nämlich Nutzung und Realisierung. Nun könnte aber auch der Kunde bereits eine technische Anforderung zur Realisierung haben, da er eine MySQL-Umgebung einsetzt und Kompatibilität sowie günstigere Lizenzkosten will. Er spezifiziert:





M-Req-2: Die Datenbank zur Verwaltung der Kundendaten wird mit MySQL realisiert.

Nun wird aus der bisherigen Komponentenanforderung (K-Req-1) eine Marktanforderung (M-Req-2). Ebenso gibt es Situationen, in denen das Lastenheft und die Geschäftsanforderungen so vage sind, dass das Pflichtenheft auch als Problembeschreibung fungiert. Anstatt über solche Feinheiten zu debattieren, ist es wichtig, dass die Anforderungen immer mit ihrer jeweiligen Quelle spezifiziert werden. Dann ist zu jedem Zeitpunkt klar, wie es zur Anforderung kam und welche Freiheitsgrade für eine Änderung bestehen, was die Realisierung erleichtert. M-Req-2 lässt sich sehr viel schwerer beeinflussen als die konzeptionell gleiche K-Req-1, da die M-Req-2 direkt vom Kunden stammt und daher die Lösung bereits definiert.

Anforderungen und Lösungen bedingen sich gegenseitig, und dürfen nicht vermischt werden. Die zwei Sichten auf Anforderungen und Lösungen sind in Abbildung 2–4 anhand typischer Arbeitsergebnisse konkretisiert. Horizontal sind Funktionen und Ziele aus Sicht des Markts (in B2C) oder eines Auftraggebers (in B2B) beschrieben. Vertikal ist die Lösung als spätere Realisierung dargestellt. Verschiedene Komponenten tragen dazu bei, dass Funktionen umgesetzt werden. Offensichtlich ist das keine 1:1-Beziehung, sondern eine bunt gemischte n:m-Darstellung.

Werden diese beiden sehr verschiedenen Sichten vermischt, führt das zu einem Strukturbruch (siehe auch Abb. 5). Dieser Strukturbruch ist einer der häufigsten Gründe für verkorkste Projekte und inflationäre Komplexität. Bereits in den späten Jahrzehnten des vergangenen Jahrhunderts war das als »Softwarekrise« bekannt. Aufgelöst wurde diese Krise erst durch modernes Requirements Engineering, das es mit seinen Perspektiven und der Verfolgbarkeit ermöglichte, eine Sicht in eine andere zu übertragen. Modelle helfen dabei, den Strukturbruch zu beherrschen und Konsistenz zwischen den Sichten (Problem vs. Lösung) zu erreichen (siehe Kap. 5). Beispielsweise muss die Systemumgebung sehr frühzeitig definiert werden. Ein Architekturmodell wiederum hilft bei der Identifizierung von Komponentenanforderungen.

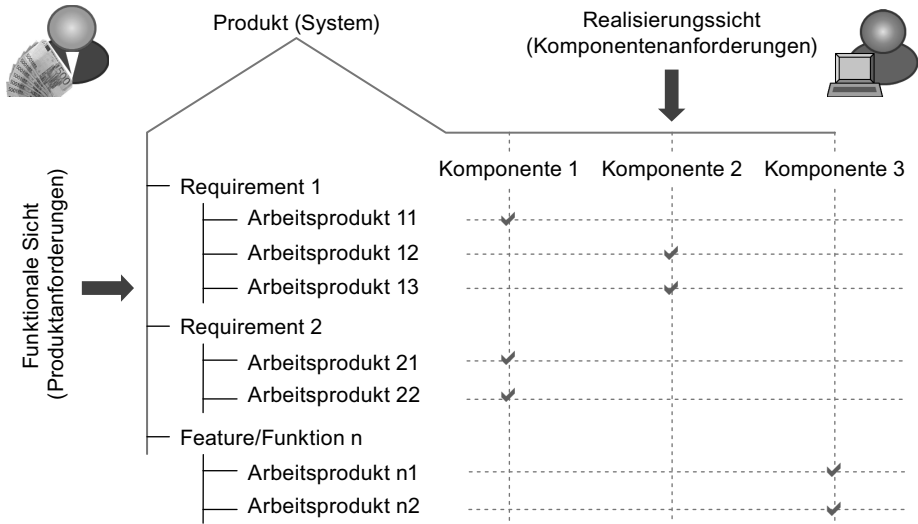


Abb. 2-4 Komponentenanforderungen werden aus Produktanforderungen abgeleitet.

Die hier beschriebene Systematik und Methodik gilt für jede Art von Produkten: Software, Services, Hardware, Mechatronik, künstliche Intelligenz – oder auch für ganz banale Konsumgüter und moderne Bauwerke. Wir unterscheiden kein Requirements Engineering anhand von Branchen oder Produkten. Requirements Engineering hat sich aus einem interdisziplinären Diskurs entwickelt, beispielsweise aus Patterns von (Gebäude-)Architekten und aus der Serviceorientierung in der Medizin. Insofern wollen wir hier auch keine künstlichen Gräben zwischen greifbaren Systemen und virtuellen Diensten aufreißen. Aus der Sicht des Requirements Engineering ist die Vorgehensweise identisch. Und Hand aufs Herz: Wo ist die Trennung in der Lösung? Die enge Verknüpfung von Systementwicklung und Dienstentwicklung, die gerade im Requirements Engineering explizit adressiert werden muss, zeigt Abbildung 2-5.



Beispiel: Serviceorientierung

Viele heutigen Dienste bestanden früher aus Hardware oder Software. Betrachten wir multimodale Transportlösungen, die verschiedene Verkehrsträger verschmelzen: Bis vor einigen Jahren standen an den Haltestellen von Bussen und Bahnen Telefenzellen, damit man anrufen konnte, um sich abholen zu lassen oder um ein Taxi zu bestellen. Heute nutzt man einen Mobilitätsservice, der einem die beste Verbindung von A nach B mit unterschiedlichen Verkehrsträgern darstellt. Das verknüpft Hardware (Zugsignalisierung) und Software (Apps und Betriebsleitzentralen) zu Diensten – mit einem serviceorientierten Requirements Engineering (siehe auch Abschnitt 11.5).

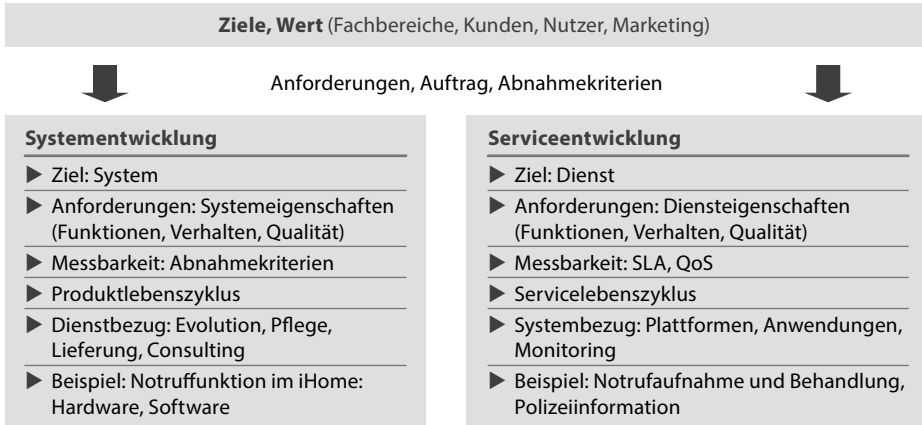


Abb. 2-5 Requirements Engineering für Systeme und Dienste

2.3 Arten von Anforderungen

Man unterscheidet drei Arten von Anforderungen (Abb. 2-6):

- Funktionale Anforderungen
- Qualitätsanforderungen
- Randbedingungen

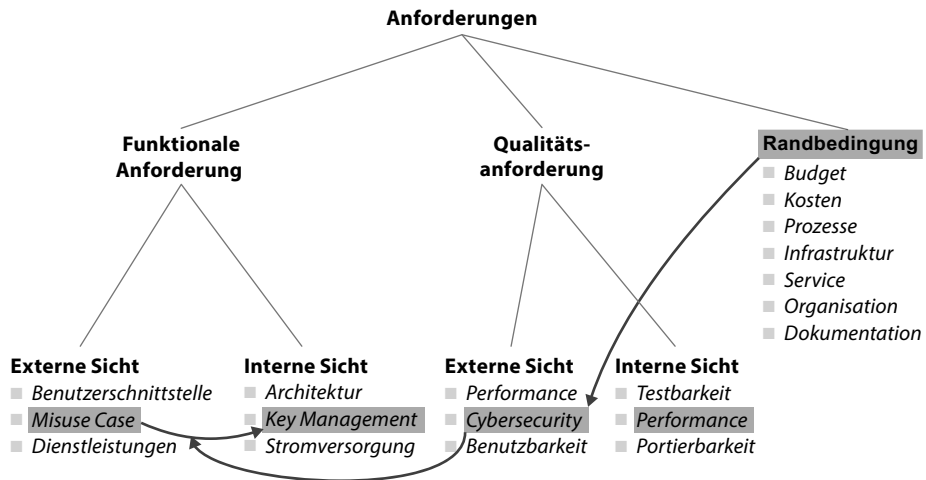


Abb. 2-6 Unterschiedliche Typen von Produktanforderungen

Funktionale Anforderung

Eine funktionale Anforderung beschreibt eine vom System oder einer Systemkomponente bereitzustellende Funktion des betrachteten Systems. Sie erläutert in der Sprache des Systems, was das System tun soll, beispielsweise die Berechnung einer Ausgangsgröße aus Eingangsgrößen durch Anwendung eines Algorithmus.

**Beispiel:**

Der Datenaustausch an der externen Schnittstelle xyz wird mit 128 Bit PGP-verschlüsselt.

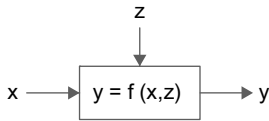
Funktionale Anforderungen beschreiben funktionsorientiert und in der Sprache des Produkts, was das Produkt tut. Sie lassen sich in der Entwicklung verfolgen und auch validieren. Man kann für eine bestimmte Funktion einen Testfall schreiben, der später in verschiedenen Testphasen geprüft wird.

Beispiele für solche funktionalen Anforderungen sind Funktionen, Ablaufbeschreibungen und Szenarien, die angeben, wie ein System auf bestimmte Eingangsgrößen oder Eingaben zu reagieren hat (Abb. 2–7). Dazu gehören auch Daten- oder Schnittstellenanforderungen, die nötig sind, um das zu entwickelnde System in einer bestimmten Umgebung einsetzen zu können. Geschäftsprozesse und Workflows sind ebenfalls funktionale Anforderungen.

Ein **Geschäftsprozess** beschreibt eine Folge zusammengehöriger Aktivitäten, die schrittweise ausgeführt werden, um ein geschäftliches oder betriebliches Ziel zu erreichen. Er beschreibt als Anforderung, wie das System intern operiert, um die Anforderungen der Umwelt zu erfüllen. Zur Vereinfachung werden Geschäftsvorfälle genutzt, die den Geschäftsprozess als Instanz konkretisieren. Geschäftsvorfälle sind Vorgänge, die die Vermögenszusammensetzung in einem Unternehmen beeinflussen oder verändern.

Workflows als Anforderungen beschreiben eine inhaltlich abgeschlossene, zeitlich zusammenhängende Folge von Aktivitäten, die zur Bearbeitung eines betriebswirtschaftlich relevanten Objekts notwendig sind und deren Funktionsübergänge von einem Informationssystem gesteuert werden. Der Workflow beschreibt eine Prozesssicht, während der Geschäftsprozess die Sicht auf betriebswirtschaftliche Faktoren betrachtet.

Ein **Anwendungsfall** oder **Use Case** schließlich als Beispiel für eine funktionale Anforderung beschreibt den Bezug einer Systemleistung durch die Außenwelt. Anwendungsfälle vermitteln also, was die Umwelt vom System erwartet. Sie sind daher vor allem zur Beschreibung von Abläufen an Schnittstellen geeignet.



- ▶ x: Eingangsgröße
- ▶ y: Ausgangsgröße
- ▶ z: Randbedingungen, z.B. Datenformate, Ereignisse
- ▶ f: Algorithmus, Abbildung von Eingangsgröße unter gegebenen Randbedingungen auf Ausgangsgröße
z.B. berechnen, übertragen, wandeln
- ▶ Quantifizierung
z.B. 10V, 200Nm/s etc.

Abb. 2-7 Funktionale Anforderungen

Qualitätsanforderung

Eine Qualitätsanforderung beschreibt eine qualitative Eigenschaft, die das betrachtete System oder einzelne Komponenten des Systems aufweisen müssen (siehe auch Abschnitt 3.5). Qualitätsanforderungen (manchmal auch **nichtfunktionale Anforderungen** genannt) ergänzen die funktionalen Anforderungen.¹ Beispiele sind Zuverlässigkeit, Verfügbarkeit, Wartbarkeit, funktionale Sicherheit und Cybersecurity.²



Beispiel:

Die Verschlüsselung und Entschlüsselung einer Transaktion muss innerhalb von einer Millisekunde abgeschlossen sein.

Qualitätsanforderungen sind nur aus der System Sicht beschreibbar. Oft werden nur die funktionalen Anforderungen hinreichend präzise spezifiziert, während die Qualitätsanforderungen vage bleiben. Bestimmte funktionale Anforderungen können spezifische Qualitätsanforderungen bedingen, beispielsweise die Robustheit gegenüber unzulässigen Eingaben oder Signalrauschen.

Zur besseren Verfolgbarkeit sollten Qualitätsanforderungen in funktionale Anforderungen »übersetzt« werden. Das erleichtert die Verfolgbarkeit und Validierung. Beispielsweise lässt sich Wartbarkeit durch Lesbarkeitsindizes und Reviews prüfen. Anforderungen an die Sicherheit lassen sich durch Reviews und Verweise auf einschlägige Standards prüfen. Mit wachsender Produkthaftung und expliziten Vertragsstrafen, die sich auf Nichterfüllung von Anforderungen beziehen, wächst das Interesse bei Kunden und Lieferanten, alle Anforderungen so präzise zu definieren, dass sie eindeutig implementiert, geprüft und abgenommen werden können.

1. Wir verwenden in diesem Buch konsequent die Bezeichnung »Qualitätsanforderungen«.
2. Im Englischen auch als RAMSS (Reliability, Availability, Modifiability, Safety, Security) bezeichnet.

Bei den funktionalen Anforderungen und Qualitätsanforderungen unterscheiden wir eine interne (Entwicklung) und eine externe (Kunde, Benutzer) Sichtweise (Abb. 2–6, unten). Die interne Sichtweise beschreibt Anforderungen, die vor allem aus Entwicklungssicht eine Rolle spielen. Dazu gehören beispielsweise konkrete Anforderungen zum Stromverbrauch oder zur Architektur, aber auch Qualitätsanforderungen, wie die Validierbarkeit. Die externe Sichtweise spiegelt die Kunden- oder Benutzersicht wider. Dabei geht es um direkten Zusatznutzen aus der Sicht dessen, der dafür bezahlen soll. Die meisten explizit beschriebenen Anforderungen in einem Software- oder Systemprojekt fallen in diese Kategorie der funktionalen Anforderungen aus Benutzersicht.

Randbedingung

Randbedingungen sind Anforderungen, die die Art und Weise einschränken, wie das betrachtete System realisiert werden kann. Randbedingungen ergänzen die funktionalen Anforderungen und die Qualitätsanforderungen. Beispiele sind Kosten, Geschäftsprozesse und Gesetze.



Beispiel:

Die Verschlüsselung einer Transaktion muss den gesetzlichen Anforderungen des BSI genügen.

Randbedingungen beschreiben Grenzen, Vorgaben und Beschränkungen, beispielsweise durch Geschäftsprozesse oder Infrastruktur. Häufig umfasst dies heutzutage eine ganze Anzahl von Lieferanten, Unterauftragnehmern oder Offshore-Outsourcing-Partnern. Bereits hier werden die Randbedingungen des späteren Projekts definiert, denn schließlich müssen der Preisrahmen und die geplanten Umsatzzahlen im Marketing lange vor der exakten Spezifikation bekannt sein.

Auch organisatorische Randbedingungen spielen eine Rolle, obwohl sie nur ungern als Anforderungen zugegeben werden. Jedoch gibt es seit Jahren Untersuchungen über den Einfluss der Organisationsform auf die Architektur. Melvin Conway hat daraus bereits vor knapp 50 Jahren ein Gesetz abgeleitet, das bis heute nach ihm benannt ist [Conway1968]. Das Gesetz von Conway (Conway's Law) besagt, dass die Struktur und Architektur eines Produkts die organisatorische Struktur widerspiegeln. Eine Matrixorganisation unterstützt beispielsweise eine modulare Struktur. Ist die Organisation eher informell, wie beispielsweise bei Start-ups, wird man vergeblich nach klaren Schnittstellen innerhalb des Produkts suchen.

Gesetzliche Vorgaben oder Standards sind Randbedingungen, die direkt zu funktionalen Anforderungen oder Qualitätsanforderungen führen. Viele Ausschreibungen im System- und Softwarebereich fordern heute eine hinreichende Prozessfähigkeit des Lieferanten. Hintergrund dafür ist, dass die Kunden immer weniger in der Lage sind, genau zu beurteilen, ob die Lieferanten tatsächlich auf der Höhe

der Zeit sind und ob sie ihre Zusagen auch einhalten können. Die Produkthaftung verlangt beispielsweise, dass der Lieferant den Stand der Technik beherrscht. *Dieses Buch beschreibt den Stand der Technik im Requirements Engineering.*

2.4 Was ist Requirements Engineering?

Requirements Engineering (RE) ist das disziplinierte und systematische Vorgehen (d.h. »Engineering«) zur Ermittlung, Dokumentation, Analyse, Prüfung, Abstimmung und Verwaltung von Anforderungen unter kundenorientierten, technischen und wirtschaftlichen Zielvorgaben. Das Ziel von RE ist es, qualitativ gute – nicht perfekte – Anforderungen zu entwickeln und sie in der Umsetzung risiko- und qualitätsorientiert zu verwalten. Systematisches RE macht den Unterschied aus zwischen einem erfolgreichen Produkt und einer Sammlung irrelevanter Funktionen.

Diese Systematik mit sechs konkreten Aktivitäten zeigt Abbildung 2–8. Wir haben bewusst keine zeitlichen oder kausalen Abhängigkeiten dargestellt, denn das ist, wie Sie später sehen werden, gar nicht so einfach.

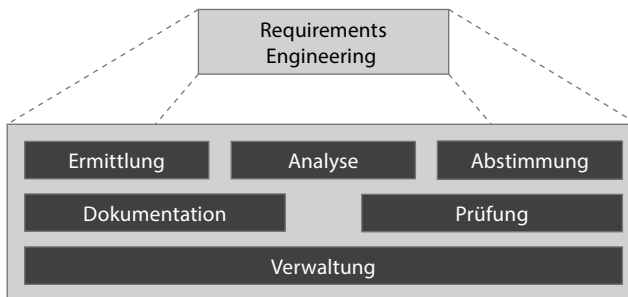


Abb. 2–8 Aktivitäten des Requirements Engineering

Requirements Engineering ist eine fachübergreifende Disziplin. Requirements Engineering bedient sich der Erfahrungen aus der Systemtechnik, der Psychologie, der Betriebswirtschaftslehre, dem Marketing, dem Produktmanagement, dem Projektmanagement und natürlich der Informatik.

Requirements Engineering ist eine Kerndisziplin aller Ingenieurwissenschaften und damit auch der Softwaretechnik und der Systemtechnik. Requirements Engineering ist nicht originär für die Softwaretechnik, und viele Methoden sind in andere Systeme transferierbar. Dieses Buch betrachtet das RE aus diesem Grund ganzheitlich und branchenübergreifend. Wir sprechen hier über Systementwicklung, denn häufig wird Software als Bestandteil eines größeren Systems geliefert [INCOSE 2015]. Von einem *System* ist die Rede, wenn es sich um eine Verbindung von Hardware, Software, Prozessen und Personen handelt, die gemeinsam die Fähigkeit haben, ein bestimmtes Ziel zu erreichen oder bestimmte Eigenschaften auszubilden.

Requirements Engineering schafft eine gemeinsame Basis zu Zielen und Anforderungen zwischen den Benutzern und den Entwicklern eines Produkts. Oft sind

die Kunden nicht die Benutzer, was im RE zu massiven Konflikten führen kann. Kunden und Benutzer sind auch nicht notwendigerweise immer außerhalb der eigenen Firma angesiedelt. Damit spielt RE eine Schlüsselrolle während der gesamten Produktentwicklung (Abb. 2–9).

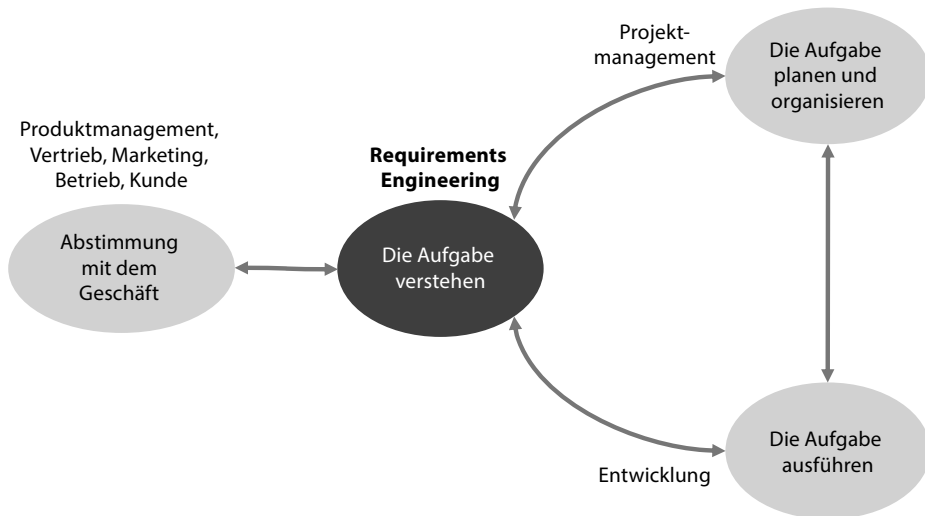


Abb. 2–9 Requirements Engineering im Kontext der Produktentwicklung

Requirements Engineering bringt verschiedene Interessen und Sichtweisen unter einen Hut. Es begrenzt die Probleme, die im Projekt sowieso auftreten. RE ist vor diesem Hintergrund eine Sisyphusarbeit. Egal, wo man anpackt, überall tun sich Lücken, Unklarheiten und Unschärfen auf. Wir können mit den Anforderungen nicht alle Inhalte endgültig klären. Das führt zu Problemen an den Stellen, die wir mit weniger Energie verfolgen. Ob die Probleme in Ihrem Unternehmen auftreten oder auf der Kundenseite, ist nahezu egal. Es trifft Sie. Ein Kunde, der zu lange warten muss und der nicht erhält, was er will, ist unzufrieden – selbst, wenn das Projekt im Budget abgeschlossen hat. Das Gleiche gilt, wenn Sie eine Funktion, die dem Kunden wichtig ist, herunterpriorisiert haben. RE hat daher viel mehr »politische« und psychologische Aspekte, als man gemeinhin wahrhaben will.

Requirements Engineering bestimmt die Wertschöpfung im gesamten Lebenszyklus. Abbildung 2–10 zeigt, wie Anforderungen zielorientiert die Bereiche des Unternehmens auf den Markt und den Kunden einstellen. Im Marketing werden Kaufkriterien bewertet. Der Vertrieb schafft mit dem Marketing und der Produktentwicklung eine Wertvorstellung, die dann durch die Entwicklung umgesetzt wird. Nachhaltiger wirtschaftlicher Erfolg bei Software entsteht durch ein funktionierendes Servicemodell. Damit wird der Wert gesichert und neue Kaufabsichten werden stimuliert. Die verbindenden Pfeile sind die Anforderungen in verschiedenen Stadien: ein Kreislauf, wie ihn erfolgreiche Unternehmen vorleben.

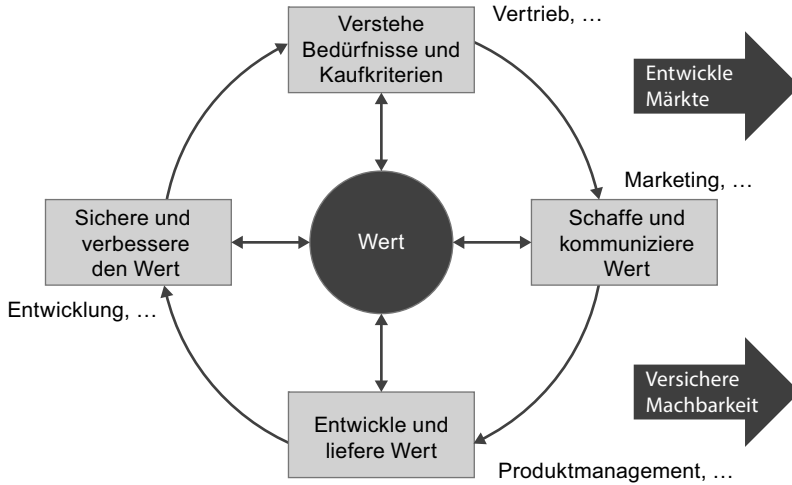


Abb. 2-10 Requirements Engineering und Wertschöpfung im Lebenszyklus

RE begleitet ein Produkt durch seinen gesamten Lebenszyklus. RE wird sowohl bei neuen Produkten als auch bei Änderungen bestehender Produkte angewandt. RE wird vor dem Projektstart und während der gesamten Laufzeit eines Projekts eingesetzt.

2.5 Requirements Engineering in der Praxis

Requirements Engineering ist die Disziplin, die Bedürfnisse auf Lösungen abbildet. Requirements Engineering ist als Disziplin sowohl problem- als auch lösungsorientiert. Als problemorientierte Disziplin beschäftigt sich RE mit den Fragestellungen oder Problemen, die dadurch entstehen, dass Software als Lösung für übergeordnete Ziele eingesetzt wird. Als lösungsorientierte Disziplin hängt RE stark mit der Produktentwicklung zusammen. Beide Perspektiven, also Problem- und Lösungsorientierung, finden im RE zusammen. Ein Schlüsselproblem ist die Unsicherheit von Anforderungen. In agilen Projekten gilt oft IKIWISI (*I know it when I see it*). Das darf nicht zum Postulat werden, denn sonst wächst unnötige Nacharbeit.

**Beispiel:**

Zu Beginn eines Coachings, Workshops oder Projekts höre ich immer genau hin, wie über das RE gesprochen wird. Oft sprechen die Beteiligten vom »Sammeln« der Anforderungen. Das muss schiefgehen, denn Anforderungen liegen nicht einfach im Projekt herum. Auch komplexe Lastenhefte mit vielen Funktionen bringen nichts, denn die entstehen beim Auftraggeber leider zu oft durch hirnloses »Copy-and-paste« aus früheren Spezifikationen. Von Noriaki Kano und Steve Jobs haben wir gelernt, dass ein erfolgreiches Produkt mit jeder seiner Funktionen einen Wert liefern soll. Wert wiederum existiert nur (!) im Auge des Kunden. Wenn der Kunde begeistert ist, kauft er das Produkt, selbst wenn Funktionen im Vergleich zum Mitbewerber fehlen. Die Kunst besteht darin, dieses »Wow!« zu liefern. Wir nutzen in Requirements-Workshops mit Kunden dazu konsequent das Kano-Modell, um solch überzeugende Anforderungen zu entwickeln. Merke: Papier ist geduldig, und lange Feature-Listen stellen keinen Wert dar. Was umfangreich spezifiziert ist, ist selten das, was für den Markt und die Kunden wirklich wichtig ist.

RE braucht eine systematische Methode im Unternehmen, um die Wünsche verschiedener Stakeholder zu erfassen, zu bewerten und zu verbinden. Danach werden die Anforderungen übersetzt, denn nur selten sprechen sie die Sprache des Projektteams, das anschließend mit ihnen arbeiten muss. Das Projektteam setzt die Anforderungen in Funktionen oder Eigenschaften um, die implementiert werden. Die Implementierung wird ständig mit den Anforderungen verglichen. Eventuelle Änderungen werden dokumentiert, mit den bestehenden Anforderungen und Ergebnissen verglichen, mit den Stakeholdern unter Berücksichtigung der Einflüsse auf das laufende Projekt abgestimmt und fließen dann kontrolliert in den Entwicklungsprozess ein.

Mit folgenden wesentlichen Herausforderungen müssen sich alle Stakeholder eines Softwareprojekts auseinandersetzen:

- Zeitraum bis zur Nutzbarkeit der Software (Time-to-Market)
- Zeitraum bis zum wirtschaftlichen Nutzen der Software (Time-to-Profit)
- Produktqualität der erstellten Software
- Kosten für die Umsetzung der Anforderungen in der Entwicklung
- Kosten der Anforderungen über den gesamten Produktlebenszyklus
- Anpassbarkeit der Software an neue Anforderungen

Obwohl RE in der Regel iterativ und kontinuierlich verläuft, gibt es doch einige grundsätzliche Abhängigkeiten, die wir in Abbildung 2–12 darstellen. Hier muss einmal die Warnung angeführt werden, dass es hier nicht um feste zeitliche Abhängigkeiten geht, sondern um Aktivitäten und Ergebnisse, die aufeinander aufbauen.

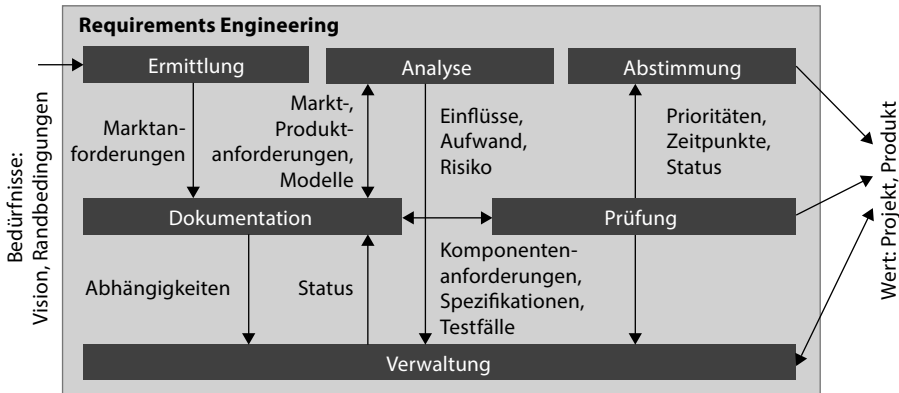


Abb. 2-11 Aktivitäten und Ergebnisse im Requirements Engineering

Wir beginnen links im Bild mit einem Bedarf aus einem Markt. Dieser Bedarf wird zunächst ermittelt (siehe Kapitel 3) und spezifiziert. Aus den Marktanforderungen entstehen dann in der Analyse zuerst Produkthanforderungen und danach Komponentenanforderungen. Parallel dazu modelliert man den Problem- und den Lösungsraum. Diese Modelle, selbst wenn sie nur temporär bestehen, sind wichtig, um Abhängigkeiten und Einflüsse zu erkennen. Die Analyse liefert bereits einen abgeschätzten Aufwand, der in der Regel aus dem Lösungsmodell mithilfe von Analogieschlüssen, Erfahrungswerten oder anhand des angenommenen funktionalen Umfangs berechnet wird.

Die Analyse schafft ein Lösungsmodell und bewertet das Projektrisiko. Oftmals bedingt ein zu hohes technisches Risiko, dass alternative Lösungsmodelle entwickelt werden. Die Analyse sollte daher prinzipiell verschiedene Alternativen entwickeln und eventuell sogar zwei Sätze von Produkthanforderungen pflegen, bis die Entscheidung für eine bestimmte Lösung gefallen ist.

Nach der Analyse werden die Anforderungen anhand ihrer gegenseitigen Abhängigkeiten, Aufwände und Prioritäten zu Paketen zusammengefasst, die in einem oder mehreren Projekten bearbeitet werden. Die Paketgröße wird von den Randbedingungen an das Projekt bestimmt, also von dem verfügbaren Budget, den Ressourcen mit ihren Fähigkeiten und dem erlaubten Zeitraum. Der Liefertermin ist die dominierende Zielvorgabe, und die anderen Parameter sind untergeordnet. Man wird also den Liefertermin festlegen und danach anhand des vorhandenen Budgets entscheiden, welche Anforderungen pro Release geliefert werden können.

Die Vereinbarung von Anforderungen folgt primär den Geschäftskriterien. Sie sollte strategischen und operativen Zielen untergeordnet sein und niemals lokal optimieren. Viele Projekte scheitern daran, dass die Beziehungen zu anderen Projekten, Kunden, Wettbewerbern oder Märkten zu spät erkannt wurden. Wir gehen in Abschnitt 3.3 detaillierter auf das Thema der Verantwortungen und Schnittstellen ein.

Anforderungen sind unsicher und ändern sich mit einer monatlichen Rate von ca. 1–5 % des gesamten Projektaufwands [Ebert2007]. Das heißt, dass sich bei Anforderungen, für deren Umsetzung 100 Personenwochen geschätzt wurden, pro Monat Inhalte im Umfang von bis zu 5 Wochen ändern. Diese 5 Wochen relative Änderung sind nicht immer mit 5 Personenwochen Aufwand zu erledigen. Wenn die Änderung erst kommt, nachdem die Anforderung bereits integriert ist, wird diese Änderung ein Vielfaches des ursprünglich dafür erforderlichen Aufwands benötigen. Diese Hebelwirkung unterstreicht nochmals den Geschäftsnutzen eines systematischen RE mit frühzeitiger Abschätzung von möglichen Änderungen (siehe auch Abschnitt 1.3). Was über dieser 5 %-Änderungsrate pro Monat liegt, gefährdet den Projektverlauf massiv und kann nur mit evolutionären Vorgehensweisen abgefangen werden (siehe Kap. 9).

Eine gute Verfolgbarkeit schafft Projektkontrolle und das Änderungsmanagement (siehe Abschnitt 8.2). Professionelles Änderungsmanagement überzeugt Kunden von der notwendigen Projektdisziplin. Wenn ein Projektmanager allzu leicht Änderungen akzeptiert, entwickelt der Kunde das Gefühl, dass ein Telefonanruf ausreicht, um nochmals »abzustimmen«. Diese Mehrkosten sind dann meistens versteckt, tragen aber zur mangelnden Profitabilität oder zu Zeitverschwendung und Verzögerungen bei. Vor jeder Änderung müssen deren Einflüsse auf Projektpläne und davon betroffene Entwicklungsergebnisse betrachtet werden. Beispielsweise müssen die Teststrategie oder Benutzerdokumente angepasst werden. Nur wenn diese Einflüsse akzeptabel sind, werden die Änderungen angenommen.

**Beispiel:**

In einem aktuellen Fall hatte ein Zulieferer einige späte Änderungsanforderungen erhalten, die allesamt angeblich ganz kritisch waren. Die Zeit für eine Einflussanalyse fehlte, und in der letzten Integrationsstufe gab es immense Verzögerungen. Wir haben dann für den Lieferanten mit Workshops die Abhängigkeiten herausgearbeitet, die aufgrund unzureichender Verfolgbarkeit nicht offensichtlich waren. Damit konnten wir mit dem Auftraggeber abstimmen, was noch möglich ist, ohne Termine zu gefährden. Unser Fazit: RE ist so gut wie immer die schwächste Stelle im Prozess – und diese Stelle befindet sich an der Schnittstelle zwischen Auftraggeber und Kunde. Fehler werden in der Regel auf beiden Seiten gemacht. Der häufigste Fehler auf beiden (!) Seiten ist, dass Änderungen unter Druck kurzfristig durchgewunken werden.

Es gibt keinen Standardprozess für das RE. Jeder RE-Prozess ist eine Anpassung an die spezifischen Randbedingungen im Unternehmen und am Markt. Systematisches RE, wie wir das im Buch vorstellen, passt für ein agiles Ein-Personen-Projekt genauso wie für ein komplexes Programm mit vielen Lieferanten. Wir legen daher keine bestimmte Organisationsstruktur zugrunde, sondern zeigen, wie die betroffenen Personen zusammenarbeiten sollten, um aus Projektsicht den größt-

möglichen Erfolg zu erzielen. Die Rollen und Aufgaben leiten sich aus den Geschäftsprozessen (z.B. Marketing) und den unterstützenden Prozessen (z.B. Qualitätssicherung) ab.

2.6 Terminologie

Unsere Sprache ist unscharf, und oft reden wir aneinander vorbei. Doch gerade im RE sollten wir präzise sein. Das gilt nicht nur für das Glossar im Kundenprojekt (siehe Abschnitt 4.6), sondern gerade auch in unserer eigenen Fachsprache. Was ist der Unterschied zwischen einem Modell, einer Notation und einer Methode? Ist UML eine Methode? Wir wollen in diesem Abschnitt kurz auf solche Fachbegriffe eingehen, um sie später im Buch konsistent zu verwenden.

Alle relevanten Begriffe, insbesondere auch für die IREB-Zertifizierung, sind im Glossar am Ende des Buches beschrieben. Wichtige Begriffe, die uns im RE immer wieder begegnen, sind in Abbildung 2–12 in einen Zusammenhang gebracht (siehe auch [Balzert2023]). Die grafische Darstellung bewegt sich von abstrakt (unten) nach konkret (oben). Die Umsetzung bewegt sich von links nach rechts, d.h., links stehen die Vorgehensweisen und rechts deren Ergebnisse.

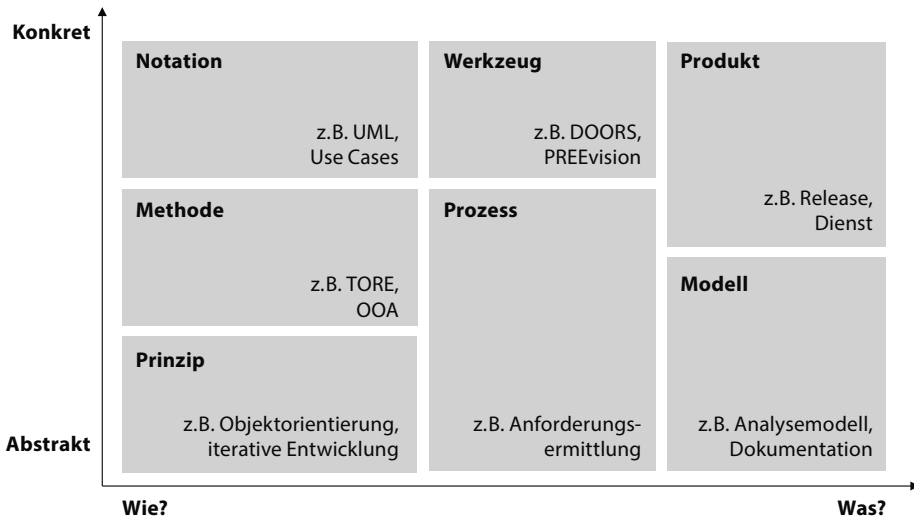


Abb. 2–12 Terminologie in der Softwaretechnik

Ein **Prinzip** liefert grundlegende zusammenhängende Regeln für die Arbeit (Abb. 2–12, links unten). Diese Regeln sind im Allgemeinen kaum empirisch belegbar. Damit werden die Argumentationsketten einzelner Regeln vereinfacht, da die Prinzipien wie mathematische Axiome nur im Zusammenhang angewandt werden. Prinzipien versuchen generell zu sein, hinreichend von konkreten Anwendungsfällen zu abstrahieren und nicht zu beantworten, wie die Ziele erreicht werden können. Nur durch diese Verallgemeinerung können sich Prinzipien überhaupt

halten. Ein bereits klassisches Prinzip ist die Objektorientierung. Ein modernes Prinzip zur Definition und Realisierung von Informationssystemen ist die serviceorientierte Architektur (SOA). Was das Prinzip bewirkt, wird separat untersucht und hängt von den Methoden ab, die zur Umsetzung des Prinzips eingesetzt werden.

Eine **Methode** ist eine definierte, systematisch eingesetzte Vorgehensweise, um vorgegebene Ziele zu erreichen. Methoden werden aus Prinzipien abgeleitet. Innerhalb der Objektorientierung (als Prinzip) gibt es verschiedene spezifische Methoden, um ein System beispielsweise objektorientiert zu analysieren. Die Methode beschreibt, wie Objekte aus den Gegenständen des Anwendungsbereichs extrahiert werden können. Eine Methode verfeinert ein Prinzip und macht es praktisch anwendbar. Methoden müssen ein Ziel erreichen, das nachprüfbar ist. So kann eine Methode vorgeben, die Wartbarkeit zu verbessern. Dann müssen die daraus resultierenden Artefakte daraufhin prüfbar sein, ob dieses Ziel auch wirklich erreicht wurde. Methoden tragen ihre Anwendbarkeit nicht notwendigerweise implizit mit sich (also ihren typischen Einsatzbereich oder Ausschlussgründe, wo die Methode nicht anwendbar ist). Sie müssen allerdings insoweit geschlossen beschrieben sein, um zu erkennen, wann sie mit welchem Erfolg einsetzbar sind. Ein Prinzip ohne jegliche Methodik zur konkreten Anwendung bleibt ein theoretisches Gerüst. Methoden sind nicht immer nur einem Prinzip zuzuordnen. Die Top-down- oder Bottom-up-Methoden sind für die Produktentwicklung wichtig. Beide können für sehr unterschiedliche Prinzipien eingesetzt werden. Insofern bilden Methoden einen Baukasten, der bei der Umsetzung von Prinzipien hinzugezogen wird.

Eine **Notation** ist eine Menge von Symbolen, die es erlaubt, ein oder mehrere Konzepte zu repräsentieren. Innerhalb der strukturierten Programmierung ist ein Strukturdiagramm eine brauchbare Notation. UML ist heute eine sehr häufig eingesetzte Notation, um ganz unterschiedliche Konzepte zu beschreiben. Notationen werden im Unternehmen oder im Projekt standardisiert, um Verständlichkeit zu gewährleisten. Schließlich sollte sich nicht jeder Entwickler zuerst mit einer neuen Nuance der Modellierungssprache auseinandersetzen müssen.

Ein **Prozess** ist die definierte Abfolge von Tätigkeiten, die der Erreichung eines Ziels dient. Er beschreibt Eingaben oder Voraussetzungen und Ausgaben, die vor bzw. nach Abschluss des Prozesses generiert werden. Prozesse können dazu dienen, Methoden und Konzepte umzusetzen. Beispielsweise kann ein Prozess die Arbeit der Anforderungsanalyse beschreiben. Prozesse reduzieren Überraschungen und schaffen Vertrauen.

Werkzeuge bieten eine automatisierte Unterstützung bei der praktischen Arbeit mit Prozessen, Methoden, Konzepten und Notationen. Werkzeuge sind aufwendig in der Herstellung und später im Einsatz, sodass die Hersteller versuchen, sie hinreichend allgemeingültig zu belassen, um unterschiedliche Notationen damit umsetzen zu können. Gute Werkzeuge forcieren den Einsatz einer Methode und erzwingen den korrekten Umgang mit einer Notation. Werkzeuge verbessern die Produktivität der Entwickler, denn sie bieten Bibliotheken, um häufig gewünschte Konstrukte wiederzuverwenden oder um Fehler frühzeitig zu finden. Im Unter-

schied zu Ad-hoc-Dokumenten helfen Werkzeuge dabei, die Modelle und Spezifikationen wartbar und damit konsistent zu späteren Änderungen zu halten.

Ein **Modell** ist eine abstrakte Repräsentation einer realen Sache in einer beliebigen Form (z.B. mathematische Symbolik, physikalische Formel, grafische Darstellung, verbale Beschreibung), um einen bestimmten Aspekt dieser Realität vereinfachend darzustellen. Modelle werden eingesetzt, um komplizierte oder komplexe Sachverhalte in ihrer Schwierigkeit einzuschränken und damit beschreibbar zu machen. Ein Modell ist das Ergebnis des Einsatzes einer Methode und häufig stark mit ihr gekoppelt. Wir gehen in Kapitel 5 auf verschiedene Analysemodelle und die zugehörigen Methoden ein. Modelle vereinfachen immer und sind daher prinzipiell falsch, da sie bestimmte Aspekte der Realität zur besseren Veranschaulichung eines anderen Aspekts ignorieren. Modelle helfen, die Aufgabe zu erfassen und daraus eine mögliche Lösung zu generieren.

Ein **Produkt** (lat. *produco* = erzeugen, liefern) ist ein Wirtschaftsgut, das in einem Wertschöpfungsprozess geschaffen wird, in dem Produktionsfaktoren umgewandelt werden. Es ist charakterisiert durch Merkmale, die einen Wert für die Benutzer liefern. Ein Produkt kann eine Kombination von Systemen, Lösungen, Materialien und Dienstleistungen sein, die intern (z.B. interne IT-Lösung) oder extern (z.B. Softwareanwendung) direkt genutzt werden oder als Komponente für ein anderes Produkt (z.B. IP-Stack) dienen.

Ein **Dienst** (engl. *Service*) ist ein nicht greifbares, temporäres Produkt, das das Ergebnis zumindest einer Aktivität an der Schnittstelle zwischen Kunde und Lieferant darstellt und keinen Eigentumsübergang beinhaltet. Dienste sind für nachhaltige Geschäftsmodelle in Hochlohnländern unabdingbar und sollten bei der Produktgestaltung – als Marktanforderungen – von Anfang an berücksichtigt werden.

2.7 Durchgängiges Beispiel: iHome

Ein Buch für die Praxis braucht ein praktisches Beispiel. Wir haben eine Hausautomatisierung gewählt, da dieses Beispiel mit IoT (Internet of Things) alle Dimensionen zwischen IT und Embedded beinhaltet. Diese moderne Hausautomatisierung nennt sich iHome (Abb. 2–13). iHome bietet Komfortfunktionen für Klima und Beleuchtung, aber auch praktische Steuerungen wie einen Aufzug sowie Alarm- und Überwachungsfunktionen, beispielsweise Rauchmelder und Alarmanlagen. Aus Sicherheitsgründen sind verschiedene Schutzfunktionen eingebaut, beispielsweise Rauchmelder oder eine Notstromunterstützung für einen etwaigen Stromausfall. Ein redundant ausgelegtes Steuerungsgerät übernimmt sämtliche Steuerungsaufgaben. Eine Machine-to-Machine-(M2M-)Schnittstelle erlaubt die Fernwartung und Ferndiagnose über Ethernet. Verschiedene Abhärtungen für Security werden verbaut, damit Angreifer nicht in die teilweise offenen Schnittstellen eindringen können. Abbildung 2–13 zeigt exemplarisch einen möglichen Angreifer, der ein Rollladensteuergerät zum Öffnen des Rollladens missbrauchen will.

Unser Beispiel deckt alle Facetten aus Projektarbeit und Entwicklung ab, also beispielsweise Systemtechnik, IT, eingebettete Systeme, Qualitätsanforderungen verschiedener Art und Wartungsaufgaben. iHome wird als Beispiel an verschiedenen Stellen in diesem Buch aufgegriffen.

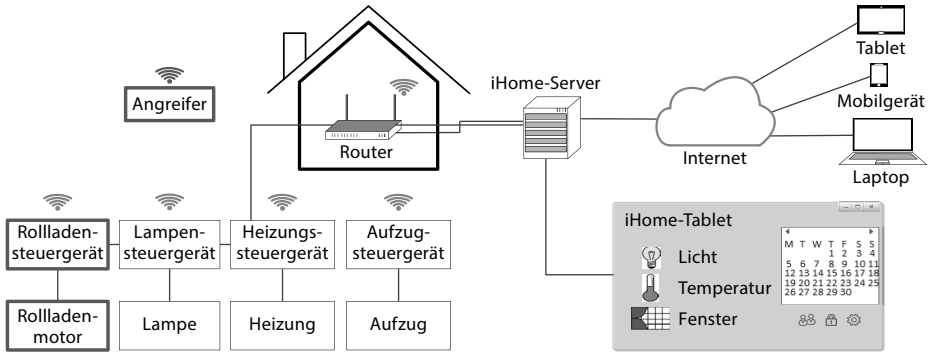


Abb. 2-13 Durchgängiges Beispiel: iHome

Im Folgenden stellen wir das RE mit allen Schritten am Beispiel iHome als Projekt vor – mit einer kombinierten Spezifikation aus Lastenheft und Pflichtenheft, die schrittweise entwickelt wird. Man findet diese Situation vor allem in kleineren Projekten und Unternehmen häufig vor.

Wir betrachten innerhalb von iHome die Beleuchtung eines Gebäudes. Dabei geht es um die Wahl von Beleuchtungsszenarien und die Abhärtung gegen Angriffe. Zuerst wird eine Produktvision vorgegeben. Darauf werden Anforderungen aufgebaut, und daraus werden schließlich Testfälle abgeleitet. Die folgenden Abbildungen (Abb. 2-14 und 2-15) und die Tabellen (Tab. 2-1 bis 2-8) zeigen anhand von Ausschnitten aus der Projektdokumentation, wie sich dieses Beispiel langsam entwickelt. Jeweils neue Teile einer Phase während der Ermittlung, Analyse und Zuweisung zum Projekt sind grau hinterlegt, um den Unterschied zu veranschaulichen.

Zielgruppe	Hausbesitzer und Geschäftsleute
Bedarf	Hausautomatisierung Mehr Sicherheit, Komfort, Behaglichkeit, Einfachheit, Einsparen von Energie
Produkt	iHome-Hausautomatisierung
Funktion	Hausautomatisierung mit zentraler Bedienung Hier fokussieren wir auf Beleuchtung, z. B. automatische Anpassung von Beleuchtungsmustern, einfach zu bedienen, günstiger Preis.
Wettbewerb	IBN (Integrated Building Networks): IBN2000 Microhard: Houses Pears: MacHouse
Wettbewerbsmerkmale	Preis: ca. 30% unter IBN2000, Bedienbarkeit, Installierbarkeit, Einfachheit; Wartungsfreundlichkeit; Kosten pro Betriebsstunde; Energiemanagement

Tab. 2-1 Requirements Engineering konkret: die Produktvision

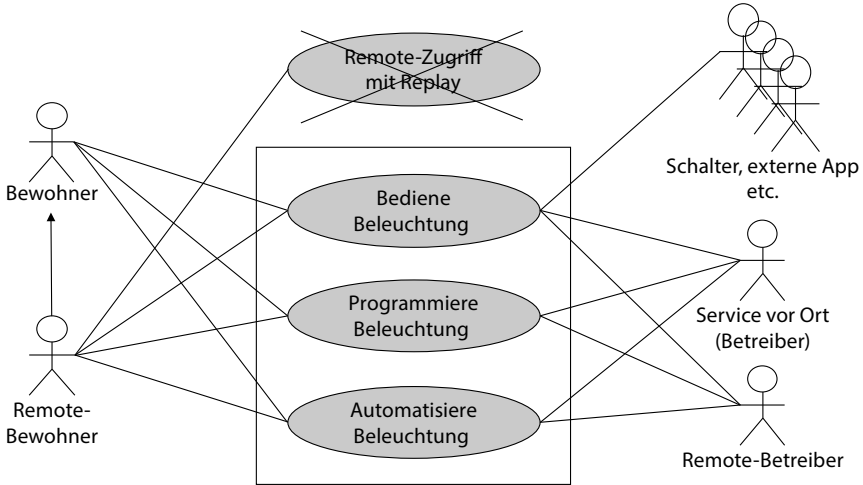


Abb. 2-14 Requirements Engineering konkret: initialer Use Case mit Kontext und Misuse Case

ID	Funktion
1	Erlaube die Definition von spezifischen Lichtszenen
2	Automatisiere die Beleuchtung für Beleuchtungsdauern und Beleuchtungsmuster
3	Stelle Sicherheitsfunktionen zur Verfügung
4	Stelle 99,9% Zuverlässigkeit sicher
5	Erlaube einfache Programmierung (ohne PC)
6	Unterstütze spezifische Komforteinstellungen
7	Erlaube einfache Bedienung mittels konventioneller Schalter
8	Erlaube handelsüblichen PC und Smartphone-App als komfortable Benutzerschnittstellen
9	Erstelle Berichte zu Energieverbrauch, Energieverbrauch an Tagen/Wochen/Monaten
10	Erkenne defekte Leuchtmittel und veranlasse automatisierte Bestellung von Ersatzleuchten und automatische Anpassung der Beleuchtungsmuster
11	Erlaube Sprachsteuerung über bestehende Spracherkennungssysteme für <ul style="list-style-type: none"> ■ Bedienung der Beleuchtung ■ Auswahl des Programms der automatischen Steuerung

Tab. 2-2 Requirements Engineering konkret: die Anforderungsermittlung

ID	Funktion	Prio	Aufwand	Risiko
1	Erlaube die Definition von spezifischen Lichtszenen	1	Mittel	Niedrig
2	Automatisiere die Beleuchtung für Beleuchtungsdauern und Beleuchtungsmuster	1	Niedrig	Niedrig
3	Stelle Sicherheitsfunktionen zur Verfügung	2	Mittel	Niedrig
4	Stelle 99,9% Zuverlässigkeit sicher	1	Hoch	Hoch
5	Erlaube einfache Programmierung (ohne PC)	1	Hoch	Mittel
6	Unterstütze spezifische Komforteinstellungen	2	Niedrig	Niedrig
7	Erlaube einfache Bedienung mittels konventioneller Schalter	1	Mittel	Niedrig
8	Erlaube handelsüblichen PC und Smartphone-App als komfortable Benutzerschnittstellen	2	Mittel	Niedrig
9	Erstelle Berichte zu Energieverbrauch, Energieverbrauch an Tagen/Wochen/Monaten	1	Mittel	Mittel
10	Erkenne defekte Leuchtmittel und veranlasse automatisierte Bestellung von Ersatzleuchten und automatische Anpassung der Beleuchtungsmuster	2	Hoch	Hoch

Tab. 2-3 Requirements Engineering konkret: die Anforderungsanalyse

ID	Funktion	Prio	Aufwand	Risiko	Release
1	Erlaube die Definition von spezifischen Lichtszenen	1	Mittel	Niedrig	1.0
2	Automatisiere die Beleuchtung für Beleuchtungsdauern und Beleuchtungsmuster	1	Niedrig	Niedrig	1.0
3	Stelle Sicherheitsfunktionen zur Verfügung	2	Mittel	Niedrig	1.0
4	Stelle 99,9% Zuverlässigkeit sicher	1	Hoch	Hoch	1.0
5	Erlaube einfache Programmierung (ohne PC)	2	Hoch	Mittel	2.0
6	Unterstütze spezifische Komforteinstellungen	2	Niedrig	Niedrig	1.0

→

ID	Funktion	Prio	Aufwand	Risiko	Release
7	Erlaube einfache Bedienung mittels konventioneller Schalter	1	Mittel	Niedrig	1.0
8	Erlaube handelsüblichen PC und Smartphone-App als komfortable Benutzerschnittstellen	1	Hoch	Hoch	1.0
9	Erstelle Berichte zu Energieverbrauch, Energieverbrauch an Tagen/Wochen/Monaten	2	Mittel	Hoch	2.0

Tab. 2-4 Requirements Engineering konkret: Bewertung und Abstimmung der Anforderungen

ID	#007
Titel	Erlaube einfache Bedienung mittels konventioneller Schalter
Versionskontrolle	1.0 15.03.2018 J. Sterna
Beschreibung	Der Use Case beschreibt, wie das Licht ein- und ausgeschaltet oder gedimmt wird.
Ablauf	Beginnt mit Druck auf Schalter. Sobald Benutzer auf die Taste drückt, läuft ein Timer an ...
Ausnahmen	1. Falls Schalter mehr als 2 Sekunden ... 2. Falls Glühbirne defekt, ...
Vorbedingungen	Der Schalter muss programmiert sein ...
Nachbedingungen	Die Helligkeit wird gespeichert.
Erweiterungen	...

Tab. 2-5 Requirements Engineering konkret: ein konkreter Use Case

1.	Versionskontrolle		
2.	Inhaltsverzeichnis		
3.	Use Cases	(Übersicht, Klassifikation)	} »Lastenheft«
4.	Kontext des Systems		
5.	Bediener	(Übersicht, Klassifikation)	
6.	Anforderungen Funktionale Anforderungen Qualitätsanforderungen	(mit Vorlage)	
7.	Randbedingungen	(Kosten, Sicherheit, Standards, ...)	
8.	Benutzerdokumentation	(Online, Handbuch, Hilfsfunktionen)	} »Pflichtenheft«
9.	Architektur, Komponenten	(Einkauf, Auswahl, ...)	
10.	Schnittstellen	(Benutzer, Hardware, Software, ...)	
11.	Lizenzierung, Patente, Copyright		
12.	Index, Glossar, Anhänge		

Tab. 2-6 Requirements Engineering konkret: Dokumentation

ID	Anf	Ereignis	Eingabe 1	Eingabe 2	Erwartetes Ergebnis
21	007	Benutzer tippt Lichtschalter	Jeder aktivierte Schalter	Licht war vorher an	Licht geht aus
22	007	Benutzer tippt Lichtschalter	Jeder aktivierte Schalter	Licht war vorher aus	Licht geht an
23	007	Benutzer tippt länger als 1 Sekunde	Licht an	Dimmerfunktion aktiviert	Licht wird dunkler
24	007	Benutzer tippt länger als 1 Sekunde	Licht an, in Dimmstellung		Licht wird heller bis Maximalwert; dann dunkelt es ab.
25	007	Benutzer tippt Lichtschalter	Lampe kaputt	Ersatzmuster aktiviert	Ersatzmuster leuchtet

Tab. 2-7 Requirements Engineering konkret: Testspezifikation und Verfolgbarkeit

ID	#0072		
Titel	Erlaube einfache Bedienung mittels konventioneller Schalter		
Versionskontrolle	1.0	15.03.2018	J. Sterna
	2.0	17.04.2018	R. Bischof
Beschreibung	Der Use Case beschreibt, wie das Licht ein- und ausgeschaltet oder gedimmt wird.		
Ablauf	Beginnt mit Druck auf Schalter. Sobald Benutzer auf die Taste drückt, läuft ein Timer an ...		
Ausnahmen	<ol style="list-style-type: none"> 1. Falls Schalter mehr als 1 Sekunde ... 2. Falls Glühbirne defekt, ... 		
Vorbedingungen	Der Schalter muss programmiert sein ...		
Nachbedingungen	Die Helligkeit ...		
Erweiterungen	... Geändert auf 1 Sekunde. In Test und Dokumentation berücksichtigen.		

Tab. 2-8 Requirements Engineering konkret: spezifischer Use Case mit Änderung

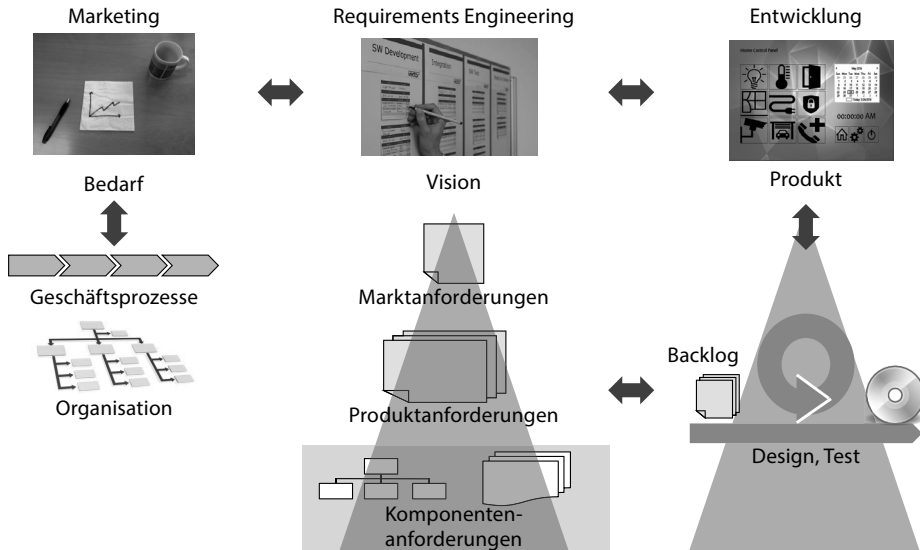


Abb. 2-15 Requirements Engineering konkret: Projekt und Organisation

2.8 Tipps für die Praxis

- RE begleitet das Produkt durch den Lebenszyklus. RE wird vor dem Projektstart und während der gesamten Laufzeit eines Projekts eingesetzt. Kritisch sind vor allem späte Anforderungen und Änderungen. Daran scheitern viele Projekte – oder werden unnötig teuer.
- Trennen Sie sauber zwischen »Was« und »Wie«. Bedürfnisse kommen vor der Lösung. Beides ist wichtig – und weil es so wichtig ist, gehört es in zwei verschiedene Dokumente.
- Unterscheiden Sie drei Typen von Anforderungen, nämlich Marktanforderungen, Produkthanforderungen und Komponentenanforderungen. Diese drei Sichtweisen sind explizit verschieden und sollten verknüpft werden, aber nicht vermischt.
- Berücksichtigen Sie alle Anforderungen. Fragen Sie relevante Stakeholder, ob etwas übersehen worden ist. Machen Sie dabei klar, dass die Ermittlung von Anforderungen noch keine Garantie für deren Lieferung ist. Geliefert wird nur, was vereinbart und bezahlt wird.
- Stellen Sie sicher, dass Ihre Anforderungen umsetzbar und testbar sind. Mogeln Sie sich nicht mit vagen und oberflächlichen Beschreibungen durch, die man »später klären kann«.

- Bewerten Sie Änderungen, bevor sie übernommen werden. Prüfen Sie die Einflüsse – technisch und organisatorisch. Zeigen Sie als Projektmanager immer die Auswirkungen im Projektplan.
- Bleiben Sie konstruktiv und positiv. Änderungen sind nötig, da wir uns stets weiterentwickeln. Mauern schafft da nur Verdrossenheit. Durchwinken schafft aber auch Risiken. Ein systematisch gelebtes, nicht dogmatisches RE erfreut Ihre Kunden – die in der Regel dabei auch vieles lernen werden.

2.9 Fragen und Impulse

- Was funktioniert im RE in Ihrem Unternehmen? Was sind Ihre eigenen Herausforderungen?
- Wie sehen die Anforderungen in Ihrem Unternehmen aus? Woher kommen sie?
- Decken die typischen Projektanforderungen die gesamte Bandbreite möglicher Kundenwünsche ab? Denken Sie an Kunden innerhalb und außerhalb Ihres Unternehmens.
- Qualitätsanforderungen spielen gerade in der Software- und Systementwicklung eine große Rolle. Welche Erfahrungen haben Sie bei der Ermittlung, Spezifikation, Umsetzung und Validierung von Qualitätsanforderungen gemacht?
- Können Sie sich Projekte vorstellen, in denen organisatorische Randbedingungen zu Anforderungen werden?
- Spielen gesetzliche Randbedingungen (als Anforderungen) in Ihrem Umfeld eine Rolle? Welche Anwendungsbereiche werden in Ihrem Umfeld durch gesetzliche Anforderungen beeinflusst?
- Weshalb wird zwischen Marktanforderungen und Produkthanforderungen konzeptionell und in der Spezifikation unterschieden? Was würde geschehen, wenn es keine solche Trennung gäbe?
- Entwickler fühlen oftmals, dass sie technisch versiert sind, aber nicht wirklich die Unternehmensziele und wirtschaftlichen Zusammenhänge verstehen (wollen). Vieles wird als »Politik« abgetan. Woher kommt diese »Isolation« – und wie lässt sie sich verringern?

9 Agiles Requirements Engineering

Ihr Nutzen aus diesem Kapitel:

Agile ist vom Hype in der Realität angekommen. Das ist gut, denn oft wird es im Sinne von »alles geht« missinterpretiert oder durch komplexe Modelle, wie SAFe, ad absurdum geführt. Agilität ohne Planung führt ins Chaos, weswegen wir hier auch nicht das sinnleere Scheingefecht Planung versus Agilität aufgreifen. Komplexe unternehmensweite Rahmen auf der anderen Seite bringen deren Erfindern zwar Lizenzgebühren, aber den Nutzern nur Overhead. »Hirn vor Hype« sollte die Richtschnur für gelebte Agilität sein. Daher übersetze ich »agil« in ein einfaches Paradigma: RACE – *Reduce Accidents, Control Essence*. In diesem Kapitel zeige ich Ihnen einige Grundprinzipien agiler Entwicklung und was sie im Requirements Engineering bedeuten. Es geht um Schlagkraft und Wettbewerbsvorteile. Fehler, Blindlast, Nacharbeiten und Reibungsverluste müssen bestmöglich reduziert werden, wohl wissend, dass wir nie fehlerfreie Software haben werden. Unser Ziel ist die Essenz eines Projekts, die bereits in der Vision beschrieben ist. Der Kunde zahlt für Leistungsfaktoren und vor allem für Wohlfühlfaktoren.

9.1 Agile Entwicklung

»Less is more« war für den bekannten Architekten Ludwig Mies van der Rohe das Grundprinzip gelebter Agilität, das er in vielen Bauwerken zeigte. Keine goldenen Wasserhähne, keine Verschnörkelungen, die niemand braucht, sondern echte Funktionalität. Lean und agil bedeutet, frühzeitig und schlank die Geschäftsziele zu erreichen. Der eingesetzte Aufwand muss den erreichbaren Wert maximieren. Zu oft bewegt man sich weit weg vom Optimum – und wundert sich über zu hohe Kosten, zu lange Reaktionszeiten und fehlende Innovationen.

Agile Entwicklung balanciert Aufwand und Wert aus. Abbildung 9–1 zeigt diese Optimierung anhand der Balance zwischen unnötigem Aufwand aufgrund verfehlter Leistungsfaktoren (linke Seite) und Zusatzaufwand durch Over-Engineering (rechte Seite). In beiden Fällen leidet das Produkt und damit der wahrgenommene Wert, denn es wird zu teuer und kommt verspätet. Offensichtlich gibt es keinen definierten Lösungspunkt, aber es gibt einige Prinzipien, die erkennen lassen, wo man steht [Cao2008, Davis2005, Ebert2021b].

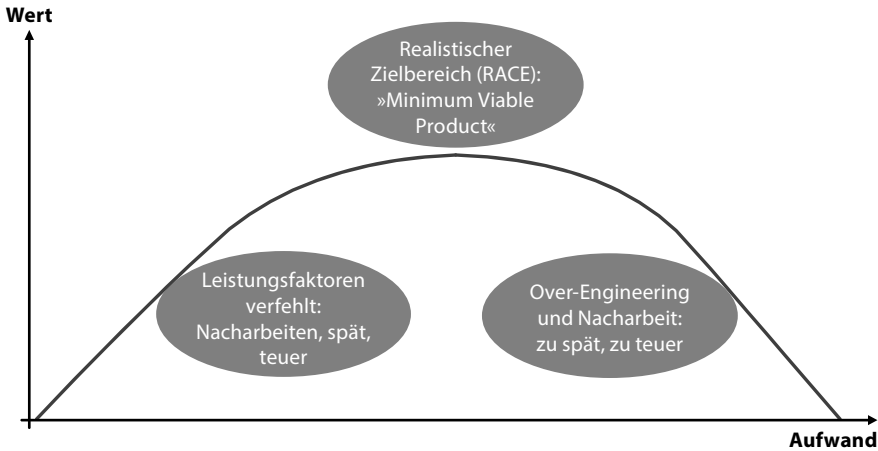


Abb. 9-1 Lean und agil: Wert für den Kunden optimieren

Agile Entwicklung erlaubt eine flexible Projektsteuerung. Abbildung 9-2 zeigt die Unterschiede zwischen klassischen Projekten und agilen Projekten anhand des magischen Dreiecks. Das magische Dreieck fasst bildhaft zusammen, was wir aus allen Projekten kennen. Projekte sind hinsichtlich Inhalten, Dauer und Aufwandsschätzung unsicher. Sie brauchen eine Möglichkeit, diese Unsicherheiten zu steuern. In klassischen Projekten wird die Funktionalität frühzeitig festgelegt und daraus werden Aufwand und Dauer geschätzt. Da die Funktionen mit ihren vielfältigen Abhängigkeiten monolithisch gesehen werden, führen spätere Änderungen zu Erhöhungen der Kosten und Lieferzeit. Abbildung 9-2 zeigt diesen Zusammenhang. Das ursprünglich geschätzte und vereinbarte Projekt (graues Dreieck links) bläht sich nach unten auf, da Funktionalität und Umfang nicht reduziert werden können.

Agile Projekte fixieren Zeit und Kosten und erlauben eine gezielte Anpassung der Inhalte. Schätzungenauigkeiten und Überraschungen führen im Projektverlauf zu Änderungen. Die Variabilität ist aber bereits von Anfang an geplant und erlaubt Anpassungen in beide Richtungen. Wenn das Budget oder die Zeit knapp werden, dann werden Inhalte späterer Inkremente reduziert. Trotzdem wird ein *Minimum Viable Product* (MVP) geliefert, also ein Ergebnis, mit dem der Kunde leben kann. Hintergrund ist die Beobachtung aus vielen Produkten in B2B und B2C, dass ungefähr die Hälfte der Anforderungen und Funktionen keinen wirklichen Wert erreichen (vgl. Abb. 1-4). Wenn einige dieser sowieso unnötigen Funktionen weggelassen werden, kann mit unveränderten Kosten und Terminen geliefert werden (Abb. 9-2 rechts).

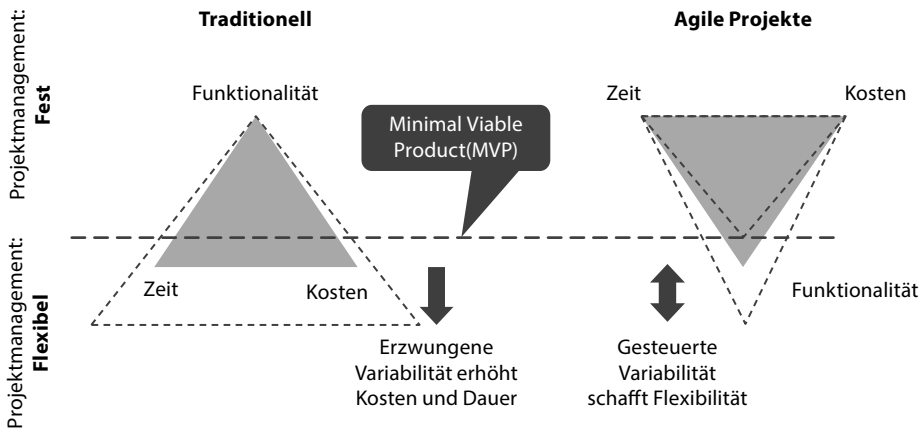


Abb. 9-2 Agile Projektsteuerung: das magische Dreieck

Agile Praktiken fanden ihren Weg ab den Achtzigerjahren in die Softwareentwicklung. Microsoft und IBM waren der Trendsetter für Agilität [Cusumano1998]. Das agile Manifest sammelte diese bewährten Prinzipien und Praktiken.

Inzwischen hat Agile seine wilden Zeiten hinter sich und ist in der Realität angekommen. Das ist gut, denn zuerst wurde es im Sinne von »alles geht« missinterpretiert und später durch komplexe Modelle wie SAFe ad absurdum geführt. Agile Evangelisten haben das eine Dogma durch das nächste ersetzt, wie im »Extreme Programming« (XP) mit der Vorgabe, dass man nur genau 40 Stunden pro Woche arbeiten darf. Danach kamen komplexe unternehmensweite Prozesse wie SAFe, die immense Prozess-Overheads schaffen.

»Alles geht« führt ins Chaos, weswegen wir hier auch nicht das Mantra Planung versus Agilität aufgreifen, sondern eher »Hirn vor Hype«. 25 Jahre nach dem agilen Manifest wissen wir, dass »agil« und »lean« immer eine Anpassung an unternehmensspezifische Randbedingungen brauchen (siehe Abschnitt 9.5).

Daher übersetze ich »agil« in ein einfaches Paradigma: RACE – *Reduce Accidents, Control Essence*. Fehler, Blindlast, Nacharbeiten und Reibungsverluste müssen reduziert werden. Dazu braucht es Planung und Prozesse.



Beispiel:

Viele Unternehmen meinen, agil zu arbeiten, und verwalten nur Chaos. Agil wird subsumiert als »anything goes«. Ein Kunde führte Scrum in einem über zwei Standorte verteilten Unternehmensbereich selbstständig ein. Wir sollten das Projekt kurzfristig reparieren. Einige Probleme fielen sofort auf. Die Standup-Meetings hatten zu viele Teilnehmer und waren dadurch langwierig und wenig wirksam. Es gab kaum Verständnis für Zusammenhänge außerhalb der Teams, und Nachfragen brauchten viel Zeit. Die Koordination der vom Kunden geforderten Arbeitsergebnisse mit den internen Sprints der verteilten Teams funktionierte nicht und lähmte den Fortschritt.



Als Maßnahmen gestalteten wir zunächst den Zuschnitt der Teams um. Scrum funktioniert am besten mit kleinen eigenverantwortlichen Teams. Die Teams wurden daher anhand der Funktionsgruppen neu strukturiert. Jedes Feature-Team plante dann autonom seine Sprints und das Backlog und nutzte Burndown-Charts für die Umsetzung. Auch bestimmte jedes Team einen Verantwortlichen, der zwischen den Teams synchronisierte. Wir coachten diese Feature-Team-Verantwortlichen und halfen vor allem bei der Zusammenarbeit und Abstimmung nach außen.

Nach einem Jahr hatten wir eine reibungslose agile Entwicklung umgesetzt. Die Teams identifizierten sich mit den Inhalten und konnten so auch einfacher zusammenarbeiten. Die Kunden waren zufrieden, da weniger Fehler geliefert wurden und die verbliebenen Fehler rasch abgebaut wurden. Die Produktivität – gemessen anhand der Burndown-Charts – stieg um 30%.

Agiles Arbeiten ist kein Dogma oder starres Framework, sondern ein Baukasten, der situativ an das Unternehmen angepasst wird. Konzentrieren wir uns daher auf die fünf Grundprinzipien, um daraus konkrete Beispiele abzuleiten (Abb. 9–3):

- Kundenwert schaffen
- Verschwendung vermeiden
- Wertflüsse optimieren
- Eigenverantwortung stärken
- Kontinuierlich verbessern



Abb. 9–3 Die fünf Grundprinzipien agiler Entwicklung

Kundenwert schaffen

Kundenorientierung heißt, dass eine Tätigkeit immer auf einen externen oder internen Kunden und Nutzen ausgerichtet sein muss. Betrachten Sie die Entwicklung mit den Augen Ihrer Kunden. Wo wird wirklich Wert geschaffen und wo entsteht Blindlast? Identifizieren Sie wenige kritische Kostentreiber. Eigentlich wissen Sie

es selbst, aber nun muss es auf den Tisch. Ganz wichtig: Nehmen Sie nichts als gegeben hin, nur weil heute so gearbeitet wird. Effizienzverbesserung beginnt damit, seine eigene Position infrage zu stellen. Wie würde ein Wettbewerber arbeiten, der auf der grünen Wiese beginnt und schnell Produkte auf den Markt bringen will?

Verschwendung vermeiden

Verschwendung wird vermieden, wenn Tätigkeiten konsequent an der Wertschöpfung ausgerichtet werden. Konzentration auf die wertschöpfenden Prozesse bedeutet, dass das organisch gewachsene Verhalten rigoros und systematisch abgespeckt wird. Die Wertstromanalyse entdeckt versteckte Unwirtschaftlichkeiten, zum Beispiel Nacharbeiten aufgrund mangelnder Qualität, komplexe Entscheidungsprozesse oder Verschwendung durch Aktivitäten, die keinen Beitrag zur Wertschöpfung leisten. Analysieren Sie gezielt die Kostentreiber in der Entwicklung. Anknüpfungspunkte sind aus unserer Erfahrung eine durchgängige Plattform- und Variantenstrategie, gezielte Wiederverwendung von Komponenten, Testfälle, Testumgebungen etc. sowie frühzeitige Fehlerentdeckung.

Wertflüsse optimieren

Tätigkeiten müssen prozessübergreifend im Fluss bleiben. In vielen Unternehmen wird nur innerhalb der Abteilungsgrenzen optimiert, während es an den Schnittstellen zu Missverständnissen und Abstimmungsproblemen kommt. Der Wertfluss in der Entwicklung beginnt mit der Produktstrategie und endet mit der Produktion, Evolution und Pflege. Wir entdecken viele Verbesserungspotenziale beispielsweise in nicht ausgerichteten Roadmaps, in zu späten Anforderungsänderungen oder in fehlender Abstimmung über Bereichs- und Landesgrenzen hinweg. Zu oft werden Konzepte, Spezifikationen und Anforderungen nur über den Zaun geworfen, ohne einen durchgängigen Eigentümer zu haben, der am erreichten Wert gemessen wird. Standardisieren Sie Ihre Technologien, Prozesse und Werkzeuge. Überlappende Aktivitäten, unklare Aufgaben, heterogene Werkzeuglandschaften und ständig neue Ideen, die nie umgesetzt werden, verschwenden Energie und demotivieren.

Eigenverantwortung stärken

Wert entsteht durch engagierte und motivierte Personen. Doch viel zu oft werden Aufgaben kleinteilig bearbeitet und Teams haben kaum Entscheidungsspielräume. Ständige Unterbrechungen und neue Aufgaben stören die Kreativität und führen zu Fehlern. Mit dem »Pull«-Prinzip (japanisch: *Kanban*) organisieren Teams die Projekte oder Teilaufgaben termingesteuert selbstständig. Sie legen fest, wer was wann macht, und fordern die gemachten Abstimmungen im Team ein. Verspätung gilt nicht, denn die Teammitglieder haben die Zeitvorgaben untereinander vereinbart. Das aus der agilen Entwicklung bekannte Scrum unterstützt dieses Vorgehen im Kleinen sowie auf Projektebene. Beachten Sie, dass Verantwortung nur dann delegiert werden kann, wenn die Teams dazu befähigt werden. Bauen Sie Kompetenzen gezielt auf und stimulieren Sie das Lernen aus gemachten Erfahrungen. Fehler sind möglich, aber sie sollten sich nicht wiederholen.

Kontinuierlich verbessern

Pflegen Sie Ihre Prozesse kontinuierlich. Prozesse sind kein Buch, das primär der Zertifizierung dient. Ständige Prozessverbesserung ist in Zeiten von hohem Kostendruck in der Softwarebranche überlebensnotwendig. Prozesse »altern«, denn ihre Umgebung entwickelt sich weiter. Auch die Prozesse des Requirements Engineering müssen von Zeit zu Zeit kritisch überprüft und verbessert werden. Schaffen Sie immer eine direkte Verbindung von Verbesserungsinhalten mit Ihren Unternehmenszielen (z.B. Qualität, Durchlaufzeit, Kosten). Verbesserungen von Prozessen werden nicht um ihrer selbst willen durchgeführt. Ein kontinuierlicher Verbesserungsprozess (KVP) fordert die Mitarbeitenden ständig dazu auf, die Abläufe zu hinterfragen und neue Ideen einzubringen. Stimulieren Sie die Teams, mit Kennzahlen zu arbeiten und daraus Verbesserungen abzuleiten und deren Umsetzung zu messen.



Beispiel:

Oft stellt sich die Frage, wie viel Agilität nun notwendig ist oder ob eine bestimmte agile Vorgehensweise erforderlich ist. »Leicht versus bürokratisch« führt da schnell zu polemischen Verirrungen. Wir haben schon viele sogenannte agile Vorgehensweisen in der Praxis gesehen, die so formal und dogmatisch waren, dass sie die Kreativität mehr behinderten als förderten. Es geht also nicht darum, ob systematische RE-Techniken auch in agilen Projekten angewendet werden können, sondern an welchen Stellen Sie wie tief einsteigen und welche der Konzepte und Techniken Sie in welchen konkreten Situationen anwenden. Agile Methoden wie Scrum lassen sich zwar alleine mit Sprints, Anwendungsfällen und Epics umsetzen, werden aber erst mit agilem RE effizient, weil dann wichtige Rückmeldungen und Erkenntnisse vor der Inkrementbildung antizipiert werden können.

Anforderungen bestimmen die Methodik im Projekt. Je nach Unsicherheit und Komplexität des Projekts mit seinen Anforderungen und Randbedingungen muss der entsprechende Prozess angepasst werden. Abbildung 9-4 zeigt entlang dieser zwei Dimensionen, wie Vorgehensmodelle und Methoden ausgewählt und angepasst werden. Daher gibt es auch nicht den einen agilen Referenzprozess aus der Schublade.

Eigenverantwortung schafft die dazu nötigen Freiräume. Teams optimieren die Produktreife mit Blick auf den Wert für den Kunden und können dabei flexibel Änderungen vornehmen und Fehler korrigieren. Nur jene Inhalte werden spezifiziert und entwickelt, für die es im Moment auch einen Bedarf gibt. Im Fokus stehen Flexibilität und Vertrauen mit möglichst wenigen unnötig angenommenen Zusatzarbeiten. Beispiele agiler Methoden sind *Scrum*, *Kanban*, *Feature-Driven Development*, *Test-Driven Development (TDD)* und *Test-Driven Requirements Engineering (TDRE)*. Ein wesentliches Merkmal dieser Methoden ist der ausgesprochene Minimalismus. Änderungen werden schrittweise und durchgängig umgesetzt und bauen aufeinander auf, sodass jederzeit geliefert werden kann.

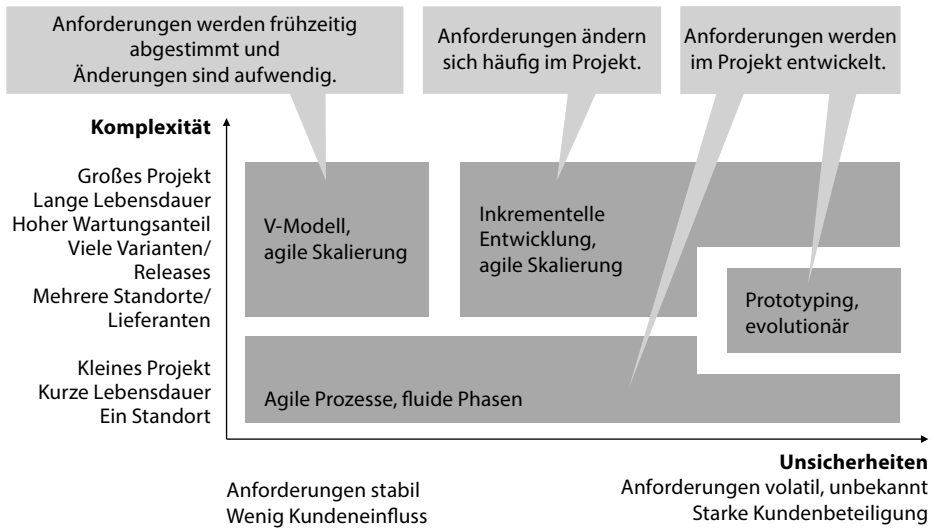


Abb. 9-4 Anforderungen bestimmen die Methodik



Beispiel:

Mein erstes agiles Transformationsprojekt leitete ich im Jahr 1998 bei Alcatel. Damals waren es knapp zehntausend Softwareentwickler, die direkt betroffen waren. Entsprechend komplex war der Change. Wir führten nach einiger Pilotierung Feature-Driven Development (FDD) ein. Seither sind aus vielen solcher Projekte, die ich weltweit begleitet habe, wesentliche Erfahrungen zusammengekommen, die ich kurz zusammenfassen will. Hier sind meine agilen Erfolgsrezepte:

1. Inkrementell arbeiten

Obwohl iterative Entwicklung schon viel älter ist als das agile Manifest, wurde sie dadurch populär. Kurze Iterationen sind für Projekte und komplexe Abhängigkeiten fast ein Allheilmittel.

2. Qualität kontinuierlich absichern

Ursprünglich hieß dieses Rezept »Daily Smoke Test« und kam von Microsoft. Kontinuierliche Integration und kontinuierliches Testen ergänzen Inkremente. Nur wenn die Lieferungen ständig getestet und integriert werden, sind sie wirklich reif.

Es gibt allerdings auch einige agile Denkweisen, die klar zum Scheitern führen:

■ NEIN zu agilen Frameworks wie SAFe

Agile Methoden müssen auf die Umgebung abgestimmt und skaliert werden. Einige Techniken, wie beispielsweise Extreme Programming, sind zu leichtgewichtig und damit in Zeiten von Produkthaftung und Compliance praxisuntauglich. Andere sind viel zu komplex und führen dazu, dass ein Viertel aller agilen Transformationen »Lost in Change« ist.



■ **NEIN zu Bottom-up-Agilität**

Oft möchten die Mitarbeitenden das Richtige, aber stimmen sich nicht ab. Agile Veränderung beginnt jedoch im Kopf und muss die gesamte Organisation erreichen.

■ **NEIN zu Branching**

Agile Entwicklung braucht kein Branching, da Änderungen sofort integriert werden. Branching führt zu einem Variantenchaos, das oft verborgene Defekte hervorruft, weil eine verteilte Designentscheidung, die in einer Verzweigung durchgeführt wird, in einer anderen Verzweigung nicht bekannt ist.

■ **NEIN zu Störungen im aktuellen Sprint**

Die Liste der Aufgaben darf innerhalb einer Iteration nicht wachsen. Egal, wer Änderungen fordert, er erhält ein klares »Nein« zur Antwort. Die vorgeschlagene Funktionalität muss bis zum nächsten Sprint warten.

■ **NEIN zu vermuteten User Stories**

Der schlimmste agile Fehler ist die Ablehnung jeglicher Vorarbeit, vor allem von Anforderungen und Analyse. Man implementiert User Stories als Backlog oder Inkremente, anstatt sie abzustimmen und auch mal auf die Abhängigkeiten und Sonderfälle zu schauen. User Stories eignen sich wie auch Use Cases für initiale Anforderungen, aber sie ersetzen kein Lastenheft.

■ **NEIN zu individuellen Requirements**

Anforderungen beeinflussen sich und dürfen daher nicht separat ermittelt und umgesetzt werden. Diese Interaktionen erfordern eine gründliche Analyse, weswegen in komplexen Projekten die Anforderungen auch gebündelt werden. Wer individuelle Funktionen entwickelt, wird von Komplexität und Abhängigkeiten überrollt und hat jede Menge Nacharbeit. Inkremente sind nötig, aber nicht einfach.

■ **NEIN zu periodischem Refactoring**

Schlechtes Design lässt sich nicht durch periodisches Refactoring in gutes Design verwandeln. Besser ist ein richtiges Redesign, aber das ist halt wieder altmodisches Softwareengineering.

9.2 Komplexität beherrschen

Agile Entwicklung beherrscht die Komplexität, ohne hirnlos zu vereinfachen. Komplexität ist nötig, denn unsere Anforderungen, Systeme und gesellschaftlichen Randbedingungen sind komplex. Wer behauptet, Dinge radikal vereinfachen zu können, hat die Randbedingungen nicht verstanden. Dogmatische Vereinfachung ohne Nachdenken führt zu kurzfristigen Lösungen, die wir später bereuen. Das zeigt sich in der Politik, aber auch in der Entwicklungspraxis. Es werden bestimmte Entwicklungsschritte gekürzt, Spezifikationen vereinfacht und Abhängigkeiten negiert. In der Integration und spätestens in der Praxis zeigen sich die Schwächen und wir

haben immense Zusatzaufwände für Nacharbeit und technische Schulden. Das beschreibe ich mit dem *RACE-Prinzip*: *Reduce Accidents, Control Essence*. Fehler, Blindlast, Nacharbeiten und Reibungsverluste müssen bestmöglich reduziert werden. Das Ziel ist, sich auf den Wert zu fokussieren, also auf die Essenz unserer Arbeit.

Ein System wird als komplex bezeichnet, wenn es vielfältig verknüpft und verflochten ist. Der Begriff der »Komplexität« wird, bezogen auf ein technisches System, umgangssprachlich häufig im Sinne von »Kompliziertheit« verstanden, obwohl dies zwei ganz verschiedene Aspekte sind. Wörterbücher erklären das aus dem Lateinischen stammende Wort »Komplexität« mit »Vielschichtigkeit« oder »dem vielfältigen Ineinander vieler Merkmale« (entsprechend dem lateinischen Wortursprung: »complector« = zusammenflechten oder umschlingen). Der Begriff »komplex« wird hier als Eigenschaft eines technischen Systems (Hardware oder Software) verstanden, das viele verschiedenartige Komponenten hat, das verschiedene Beziehungen zwischen diesen Komponenten aufweist und das unterschiedliche Zustände einnehmen kann. Die Komplexität beschreibt damit den Zusammenhang beziehungsweise das Zusammenwirken eines Systems und seiner Teile als Objekte [Ebert2021b].

Im Requirements Engineering wird Komplexität schon seit den Sechzigerjahren berücksichtigt. »*A user should be able to specify precisely how good a product (must be) he wishes to buy*«, bemerken Rubey und Hartwick schon 1968 [Rubey 1968]. Leider ist die Schlussfolgerung dieses Klassikers – »*the Statement of these objectives (d.h. Qualitätsanforderungen) would almost certainly cause the developer to slant his programming effort in such a way as to achieve higher scores*« – nach wie vor eher ein Ziel als ein Weg.

Ein System ist kompliziert, wenn es schwierig oder verwickelt ist. Das verweist auf den lateinischen Wortursprung »complicare« für zusammenfalten oder verwirren. Der Begriff »kompliziert« wird als zusammenfassende Charakterisierung eines technischen Systems verwendet, das schwer zu verstehen, zu durchschauen oder zu handhaben ist. Damit beschreibt die Kompliziertheit das Zusammenwirken zwischen einem System als Objekt und dem Betrachter als Subjekt. Die Kompliziertheit ist eine wahrgenommene – psychologische – Komplexität und hängt vom Betrachter ab. Die Kompliziertheit eines Softwaresystems hängt ab von den Vorkenntnissen des Beobachters (konkret des Softwareingenieurs), von der Wirkung der Darstellung auf ihn und von der Eignung einer gewählten Darstellung für ein bestimmtes Problem.

Der Phasenübergang von *komplex* zu *kompliziert* ist in Abbildung 9–5 veranschaulicht [Ebert1995]. Während im oberen Bereich komplizierte Objekte zunächst nur eine vage Form haben, sind sie im unteren Bereich als verstehbare Objekte mit erkennbarer Struktur dargestellt. Der Übergang von einfachen zu komplexen Objekten wurde durch Größenänderungen von links nach rechts abgebildet. Die beiden Pfeile in der Mitte symbolisieren Einflussmöglichkeiten, um sich im Koordinatensystem zu bewegen.

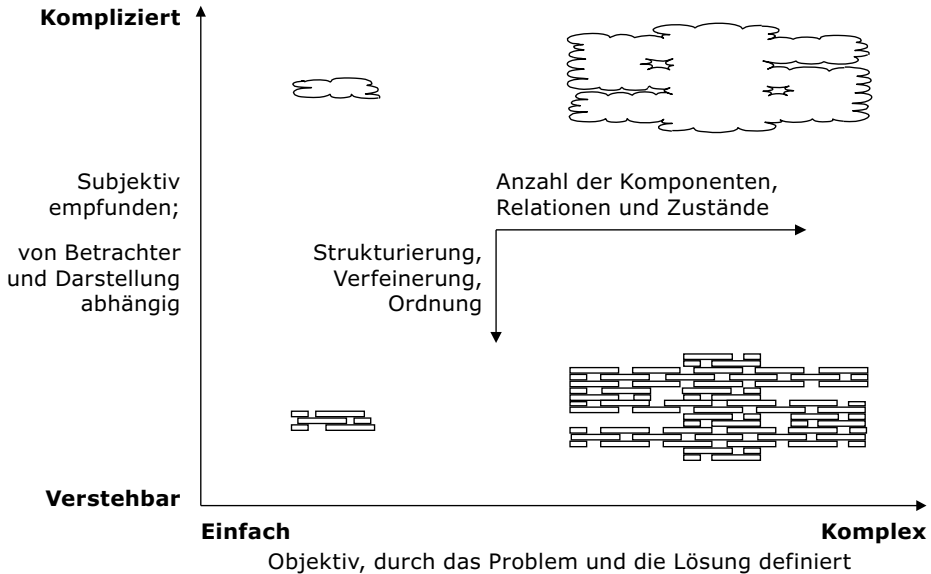


Abb. 9-5 Komplexität und Kompliziertheit

Die Beherrschung der Komplexität ist nur möglich, wenn die Kompliziertheit gezielt verringert wird. Das hatte bereits E. Dijkstra 1972 im Rahmen der Verleihung des Turing Award gefordert [Dijkstra1972]. Später hat Fred Brooks eindrucksvoll dargestellt, dass es dafür keine »Silver Bullets« gibt, sondern nur diszipliniertes Arbeiten [Brooks1987]. Die »lösungsspezifische Komplexität« entsteht während der Softwareentwicklung. Bestimmte Aspekte der problemspezifischen Komplexität sind untrennbar mit solchen der lösungsspezifischen Komplexität verbunden. Beispielsweise beeinflusst die Zahl der Sensoren eines Automatisierungssystems die Schnittstellenbeschreibung im Softwareentwurf. Die lösungsspezifische Komplexität beeinflusst die Projektkosten maßgeblich – und wächst häufig unkontrolliert [Ebert1995].

Anforderungen legen durch ihre Beschreibung und Struktur sowohl die problemspezifische Komplexität als auch Teile der lösungsspezifischen Komplexität fest. Hier folgen einige konkrete Vorgehensweisen, um die Komplexität zu kontrollieren und damit die Kompliziertheit zu reduzieren (siehe auch Kap. 4) [Ebert2007]:

- **Anforderungen klar strukturieren**
Überschriften und Vorlagen einsetzen, damit wesentliche Strukturmerkmale klar erkennbar bleiben.
- **Anforderungen kurz und prägnant halten**
Eine übliche Faustregel ist, eine einzige Anforderung auf ungefähr ein bis zwei Seiten zu beschränken. Eine einzige Seite hat den Vorteil, dass sie auf einen Blick erfassbar ist. Sobald sich eine Anforderung über mehrere Seiten erstreckt, ist das Risiko groß, dass Zusammenhänge übersehen werden.

■ Einfache und definierte Sprache verwenden

Häufig haben Anforderungen und Spezifikationen verschiedene Autoren, die ihrerseits eigene Sprachstile bevorzugen. Hier hilft ein Standardwörterbuch oder Glossar (»Data Dictionary«), das separat und verbindlich geführt wird. Das Wörterbuch muss während der Erfassung der Anforderungen bereits angelegt werden, denn es hilft auch, Widersprüche in Interviews zu erkennen.

■ Eine Standardgrammatik nutzen und die Lesbarkeit regelmäßig prüfen

Einfache Regeln dazu, welche Hilfsverben einzusetzen sind oder in welcher Person und Zeit Sätze zu schreiben sind, schaffen Konsistenz und Lesbarkeit. Die Satzlänge muss – gerade bei deutschen Autoren – beschränkt sein.

■ Bei Bildern auf Syntax und Semantik achten

Dies gilt nicht nur für die Symbolik, für die UML bereits eine gute Basis liefert, sondern auch für die Anordnung von grafischen Elementen und für Beschriftungen. Da UML keine Semantik bietet, sollte man im Unternehmen einfache Regeln zur Konsistenzsicherung und Verständlichkeit aufstellen. Beispielsweise sollten Bezeichner bereits ihren Standards aus der Programmierung folgen. Die Anzahl von Kanten und Elementen in einem Diagramm sollte sowohl nach oben als auch nach unten eingeschränkt werden. Aus der Psychologie kennt man die Anzahl von 7 ± 2 grafischen Blockelementen, die in einem Zusammenhang wahrgenommen und verstanden werden können [Miller1956]. Selbst Experten können nur bis zu 20 gleichartige Elemente in einer Abbildung erfassen.

»Gut genug« ist ein wichtiges Prinzip, um Komplexität zu beherrschen. Projekte scheitern, wenn aufgrund nicht beherrschter Komplexität zu viel oder zu wenig analysiert wird. Zu viel Analyse schafft die »Paralyse durch Analyse«. Man versucht zunehmend mehr Details auf einer immer unsichereren Basis herauszufinden. Zu wenig Analyse passiert meistens unter Termindruck und wenn sich Anforderungen plötzlich ändern. Wichtig ist es daher, bereits vor dem Projektstart die Kriterien des Projektmanagements anzuwenden und die Analysephase in ihrer Dauer an den gesamten Entwicklungszyklus anzupassen.

Was heißt »gut genug«? Es bedeutet, dass die Anforderungen eine hinreichend gute Qualität haben (z.B. Abschlusskriterien beim Review, Freigabekriterien, Restfehlerdichte) und die kritischen (d.h. hinsichtlich des Werts für den Kunden) und architekturentscheidenden Anforderungen so detailliert spezifiziert sind, dass die Entwicklung darauf aufsetzen kann. Eine gewisse Stabilität ist nötig. Andererseits kann man über Monate analysieren und wird merken, dass es ständig zu weiteren Änderungen kommt. Anforderungen gelten als stabil, wenn ihr Rauschpegel so gering ist, dass er in den folgenden Phasen nicht über die erlaubten Kosten- und Zeitlimits hinaus verstärkt wird [Ebert2014a].



Beispiel:

Praktisch alle Unternehmen leiden unter nicht beherrschter Komplexität. Nur Start-ups sind in der Lage, wirklich mit einem leeren Blatt Papier und ohne Altlasten und Varianz zu beginnen.

Ein Kunde hatte zwar agiles Arbeiten eingeführt, konnte aber trotzdem noch nicht seine Komplexität wirkungsvoll reduzieren. Seine Kunden waren weiter unzufrieden mit Fehlerbehebung und Wiederholungsfehlern. Vector wurde zur Optimierung hinzugezogen.

Unsere initiale Diagnose ergab ein typisches Phänomen: Viel Pseudo-Agilität ohne Effekt. Standup-Meetings mit zu vielen Teilnehmern; langwierig und wenig wirksam. Wenig Verständnis für Zusammenhänge, da nur verschlankt wurde, aber kein agiles Wissensmanagement aufgebaut wurde. Die Koordination der Arbeitsergebnisse mit Sprints lähmte Fortschritt. Die Synchronisation global verteilter Teams funktionierte nicht.

Wir steuerten unmittelbar dagegen, indem wir kleine autonome Feature-Teams einführten. Aufgaben wurden den Teams über Funktionen und Funktionsgruppen zugeordnet. Außerdem führten wir Burndown-Charts für Feature-Teams ein. Jedes Feature-Team bekam einen eigenen Leiter, die Synchronisation zwischen den Teams erfolgte über die Feature-Team-Leiter. Wir coachten die Feature-Team-Leiter und synchronisierten verteilte Teams mit Kanban und Sprints.

Die Verbesserungen waren schnell greifbar: Kleine motivierte Teams mit hoher Dynamik bezüglich Implementierung von Funktionen und Funktionsgruppen. Transparenz über Fortschritt je Feature-Team. Identifikation der Teams mit ihren Funktionen. Verdoppelung der Korrekturrate innerhalb von 8 Wochen nach Einführung der Feature-Teams (680 → 1300). Rückgang der nicht korrekt gelösten Fehler um 25%. Erhöhung der Kundenzufriedenheit durch bessere Qualität ohne ständiges Nachfassen.

9.3 Praxis des agilen RE

Agile Entwicklung braucht agiles Requirements Engineering. Agile Projekte und Transformationen scheitern, wenn das RE nicht dazu passt. Agilität lässt sich nicht umsetzen, wenn die Anforderungsbasis zu starr ist, wenn häufige Änderungen prozessual nicht machbar sind oder wenn Teams nicht hinreichend breit aufgestellt sind, um Entscheidungen direkt im Team zu treffen und umzusetzen [Davis2005, Cao2008, Bergsmann2018, Ebert2021b].

Für jedes der fünf agilen Grundprinzipien gibt es passende Methoden im agilen Requirements Engineering (Abb. 9–6). Die Methoden kommen im gesamten Buch zum Einsatz, weshalb wir sie nicht komplett in diesem Abschnitt erklären. Unsere Erfahrung zeigt, dass vor allem Inkremente und Priorisierung Schwierigkeiten machen. Daher wollen wir darauf intensiver eingehen, verweisen aber gleichzeitig auf die Grundlagen, wie das **Kano-Modell** (siehe Abschnitt 3.2), die **agile Planung** und Schätzung (beispielsweise mit dem **Planungspoker**) und die Umsetzung in der Validierung mit **Test-Driven RE** (TDRE, siehe Abschnitt 6.4).

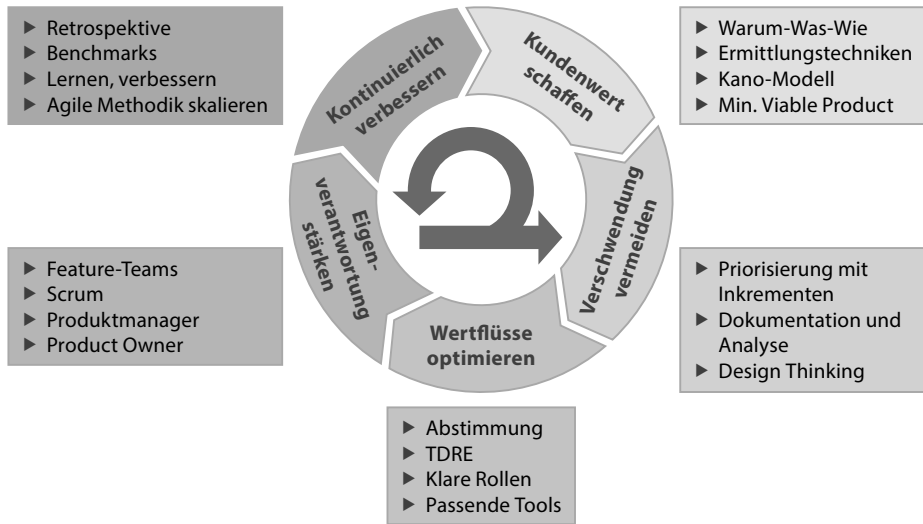


Abb. 9-6 Agile Techniken für das Requirements Engineering

Was zeichnet das agile Requirements Engineering aus, und worauf ist zu achten? Die Anforderungen sind der wesentliche Hebel, um die Werterzeugung zu optimieren. Oftmals habe ich gerade im RE eine Scheinwelt aus schwerfälligen Prozessen und Werkzeugen bemerkt, während die Produktmanager oder Entwickler mit einem agilen Spreadsheet gearbeitet haben. Den Kunden werden Zusicherungen gemacht, ohne dass diese vorher mit der Entwicklung und den zur Verfügung stehenden Ressourcen abgeglichen wurden. Das führt zu Fehlern durch Inkonsistenz, zu Redundanzen und Nacharbeit und demotiviert. RE-Prozesse müssen unterstützen, ohne einzuengen. Sie müssen aktiv vorgelebt werden, um zur Disziplin zu erziehen. Agilität heißt daher nicht, dass Prozesse unkontrolliert wegfallen, sondern dass schlanke Prozesse auf die wesentlichen Bedürfnisse abgestimmt sind.

Agiles RE unterstützt die schrittweise Entwicklung und Umsetzung von Anforderungen. Funktionen werden anhand ihrer Abhängigkeiten in Inkrementen schrittweise umgesetzt. Das Kano-Modell hilft bei der Priorisierung, damit die wesentlichen Inhalte, vor allem die wesentlichen Leistungsfaktoren, auf jeden Fall geliefert werden. Die Produktentwicklung und damit die Releaseroadmap setzt sich aus Projekten zusammen, die schrittweise Inhalte liefern. Inhalt ist der einzige Puffer, während Liefertreue und Budgeteinhaltung durch »Timeboxing« erreicht werden.

Zum Projektstart müssen die kritischen Anforderungen mit Einfluss auf Architektur, Performance und Projektdefinition ermittelt und analysiert werden. Dann wird festgelegt, wie viele Inkremente und Iterationen entwickelt werden und wie Anforderungen auf diese Schritte verteilt werden. Im iterativen Vorgehen werden danach Teilprojekte gebildet. Im inkrementellen Vorgehen werden die Inkremente aufeinander aufbauend entwickelt und integriert. Man spricht dann auch von kontinuierlicher Integration, da das Gesamtsystem kontinuierlich mit den Inkrementen wächst. Entscheidend bei den iterativen Vorgehensmodellen ist, dass Fortschritt an implementierten und getesteten Anforderungen festgemacht wird.

Sind die Anforderungen teilweise unbekannt und entwickeln sich erst im Laufe der Systementwicklung, wählt man ein iteratives oder evolutionäres Vorgehen. Zahl und Abhängigkeiten innerhalb dieser Entwicklungsschritte werden zum Projektstart definiert.

Inkrementelle Entwicklung braucht ein sauber aufgestelltes Requirements Engineering. Hier einige Tipps aus der Praxis:

- Analysieren Sie die Anforderungen von Beginn an immer unter der Perspektive, wie sie zusammengehören und welche Nutzen durch eine Gruppierung erreicht werden können.
- Analysieren Sie die Anforderungen vor Projektbeginn mit multifunktionalen Expertenteams.
- Analysieren Sie den Kontexteinfluss von Funktionen und Inkrementen vor dem Entwicklungsstart.
- Stellen Sie einen Projektplan auf, der ausschließlich auf diesen gruppierten Funktionen beruht.
- Evaluieren Sie den vorgeschlagenen Projektplan mit Bezug auf die Ingenieure, die zur Verfügung stehen.
- Weisen Sie den Teams Verantwortung für konkrete und einzeln identifizierte Anforderungen sowie für Inkremente und Meilensteine zu. Nur klare Verantwortungen schaffen klare Liefertermine.
- Verfolgen Sie den Projektstatus auf Basis des erreichten Nutzens – also anhand des Status von Marktanforderungen – und nicht nach Pseudofortschritt, wie beispielsweise nach Dokumenten oder geschriebenen Testfällen. Nur das, was der Kunde sieht und bezahlt, stellt einen Wert dar.
- Testen Sie die Inkremente unabhängig von den Entwicklungsteams.

Scrum unterstützt eine verlässlich abgestimmte Inkrementplanung. Abbildung 9–7 zeigt das agile Requirements Engineering in einem iterativen Kontext, wie er bei Scrum eingesetzt wird. Die Anforderungen werden parallel zum Projekt entwickelt und umgesetzt. Begonnen wird mit wenigen Anforderungen, die einen großen Einfluss auf Architektur und grundlegende Entwurfsentscheidungen haben. Wesentlich ist die gute Verzahnung vom Produktmanagement (Abb. 9–7 links) zur Projektplanung und schließlich zu den Teams mit ihren Iterationen und Sprints (Abb. 9–7 rechts). Anforderungen werden anhand von Prioritäten von rechts nach links gesteuert, sodass auch kurzfristige Änderungen direkt umgesetzt werden können. Entdeckte Fehler sind Anforderungen für den nächsten Sprint.

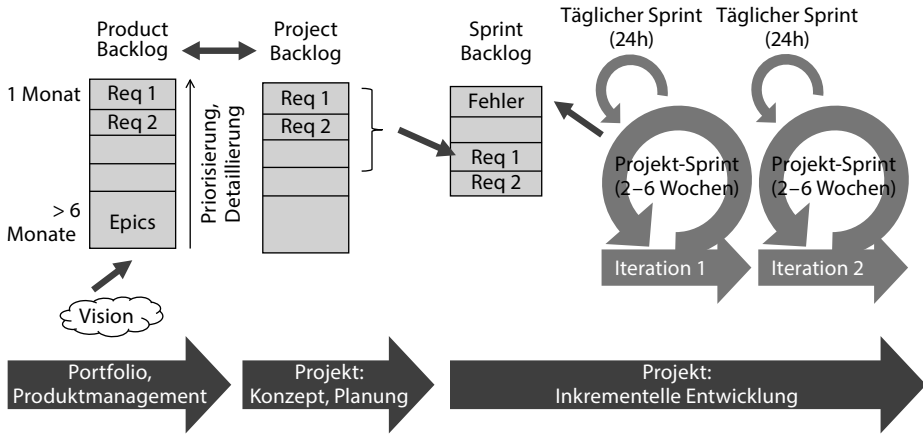


Abb. 9-7 Kaskadierte Scrums für Projekte und Produkte

Im agilen RE ist der Aufwand für RE über das gesamte Projekt relativ gleichmäßig verteilt. Damit hat das agile RE den Vorteil einer hohen Flexibilität, selbst wenn sich im Verlauf des Projekts neue Randbedingungen und Bedürfnisse ergeben, die zu beträchtlichen Änderungen führen. Unklare Bedürfnisse oder Anwendungsfälle können während der Entwicklung mit dem Kunden geklärt werden, sodass die fertige Lösung diejenigen Funktionen umfasst, die auch genutzt werden. Es wird weniger Ballast erzeugt, und die typischen Fallen des eher frontgeladenen traditionellen RE (nämlich eine hohe Änderungsrate der Anforderungen sowie sehr viele nicht genutzte Funktionen) werden vermieden. Da das agile RE anfangs keine geschlossene Basis des zu entwickelnden Systems hat, können Inkonsistenzen in der Architektur und Realisierung entstehen. Diese werden in weiteren Iterationsschritten entdeckt und behoben. Refactoring hilft beim Säubern und Aufräumen dieser Änderungen. Qualitätsanforderungen allerdings müssen bereits von Anfang an klar beschrieben und priorisiert sein, denn sie lassen sich nicht später erst in ein System hineinentwickeln.



Beispiel:

Agile Entwicklung braucht systematisches Requirements Engineering. Flexibilität und zu starke Kundenbeteiligung führen ins Chaos. Betrachten wir das »Virtual Case File System« des FBI, das zwischen 2001 und 2005 entwickelt wurde. Das IT-System sollte das bisher organisch gewachsene Dokumentenmanagement ersetzen und mit einem E-Workflow automatisieren, damit beispielsweise zusammenhängende Akten schneller gefunden werden: Standardanforderungen an ein IT-System, so sollte man meinen. Namhafte Hersteller wie Oracle und SAIC waren beteiligt.





Zudem wurden agile Techniken eingesetzt, beispielsweise wurden anfangs nur wesentliche Anforderungen spezifiziert, der Kunde wirkte direkt im Projekt mit, man versuchte sich an Inkrementen und Teilsystemen etc. Doch die Anforderungen änderten sich ständig, die Kosten stiegen, das System war fehlerträchtig, und 2005 wurde es nach 170 Millionen US\$ Investment abgebrochen. Die Gründe wurden detailliert analysiert [SEBOK2018]: Die Spezifikation änderte sich ständig, ohne einen Freeze zu definieren; fehlende Architekturmodellierung zu Beginn führte zu schlechten Architekturentscheidungen; um möglichst schnell und agil zu arbeiten, wurde Code im Voraus entwickelt, ohne dass die Anforderungen abgestimmt waren. Fazit: Agile Prinzipien wie »Kunde an Bord« und »inkrementelle Entwicklung« sind kein Ersatz für systematisches Requirements Engineering.

Agile Methoden fordern ein neues Verständnis der Zusammenarbeit von Kunde und Lieferant. Transparenz bei der Planung und später im fortschreitenden Projekt schaffen Vertrauen. Kein Kunde wird blind zustimmen, dass die Hälfte seiner Anforderungen unnötig ist. Aber er wird in gemeinsamen Workshops merken, dass die Funktionen verschiedene Prioritäten haben. So sind auch Festpreisprojekte möglich. Beide Partner schätzen zu Beginn mit kaufmännischer Sorgfalt gemeinsam das Projekt auf funktionaler Ebene. Auf dieser Basis wird ein Projektplan vereinbart und das Festpreisprojekt abgestimmt.

Natürlich können auch im agilen RE spät im Projektlebenszyklus neu auftretende Anforderungen nicht mehr einfach übernommen werden. Daher kommt dem Produktmanager in allen Vorgehensweisen die Schlüsselrolle als »Wächter« zu, der aufgrund der Kosten-Nutzen-Funktion über die Anforderungen entscheidet.

Lean und agil helfen dabei, die eigenen Strukturen, Prozesse und Werkzeuge in der Entwicklung schlank zu gestalten. Unsere Erfahrungen in der Umsetzung von lean und agil zeigen, dass in Entwicklungsprozessen 20–30 % der Kapazität durch Verschwendung gebunden sind. Das Ziel von agilem Arbeiten ist es, diese neu gewonnene Kapazität so in wertschöpfende Tätigkeiten zu investieren, dass beispielsweise mehr Projekte mit gleicher Mannschaft möglich werden, Durchlaufzeiten verkürzt werden, Produktionsabläufe abgesichert werden und eine bessere Produktqualität frühzeitig in der Entwicklung erreicht wird.



Beispiel:

Vector hat bereits viele agile Projekte mit messbaren Erfolgen durchgeführt. Das machen wir einerseits in unserer eigenen Entwicklung, wo knapp 3000 Softwareentwickler aktiv sind. Aber wir unterstützen vor allem als Berater auch andere Unternehmen bei agilen Transformationen. Diese Kombination hat Vorteile, denn wir sind eines der wenigen Beratungsunternehmen, die das tun, was sie predigen. »Eat your own dog-food«, sagt der Amerikaner dazu. Was wir umsetzen, das funktioniert.



Beispiel Lieferantenmanagement: Aufgrund der großen Hebelwirkung sind Kostenreduzierungen von 10–30% möglich. Schlechtes Lieferantenmanagement schafft ein massives Risiko für die eigene Entwicklung, denn Termine werden nicht gehalten und zugelieferte Komponenten erfüllen nicht die Anforderungen. Mit einem OEM verbesserte *Vector Consulting Services* die Termineinhaltung der Lieferungen auf über 90% und reduzierte die Fehler auf die Hälfte. Im Produktmanagement lässt sich durch eine bessere Abstimmung von Produktstrategie, Roadmaps und wiederverwendbaren Komponenten eine Verschlankung um 20–40% erreichen. Frühzeitige Fehlerentdeckung schafft 10–20% Potenzial zum Abspecken.

9.4 Design Thinking

Design Thinking ist ein Innovationsmodell, das iterativ Bedarf und Lösung entwickelt. Das Ziel dabei ist es, den Bedarf aus verschiedenen Perspektiven zu verstehen und in Iterationen Ideen zu entwickeln, zu bewerten und auf der Bewertung aufbauend neue Ideen zu entwickeln, die den Benutzer überzeugen. Durch den zwingend ergebnisorientierten prototypischen Ansatz werden typische Denkfallen vermieden, die aus Selbstzufriedenheit entstehen [Christensen2016].

Abbildung 9–8 zeigt die Vorgehensweise beim Design Thinking. Verschiedene Methoden kommen situativ zum Einsatz:

- **Businessmodell-Generierung** und **Business Model Canvas**, d.h. schlanke und auf Bedarf fokussierte Darstellung eines Geschäftsmodells anhand angenommener Marktprinzipien
- **Benchmarking**, d.h. ständiges und konsequentes Lernen von den Besten, insbesondere produkt- und branchenübergreifend
- **Prototyping**, d.h. die systematische Entwicklung von möglichen Lösungen für den initialen Bedarf mit dem Ziel, den Bedarf besser zu verstehen und bei der Nutzung sowie im Geschäftsmodell frühzeitig und ergebnisoffen zu experimentieren
- **Persona**, d.h. die angenommenen typischen Rollenmuster späterer Kunden oder Benutzer, vor allem wenn in B2C oder B2B die tatsächlichen Kunden nicht an Bord sind
- **Fokusgruppen**, d.h. aktive Einbeziehung späterer Benutzer in frühe Prototypen zur Bewertung von Ideen und Entwicklung neuer Ideen
- **Reframing** und **5 Why**, d.h. systematisches Hinterfragen von Annahmen, Randbedingungen, Zielen und dem vermeintlichen Bedarf in fünf Schritten mit der Frage »Warum ist das so?«
- **Pecha Kucha** und **Elevator Pitch**, d.h. kurze Zusammenfassung einer Idee und deren Wert in kurzer Zeit mit dem Ziel, einen Entscheider oder eine Anwendergruppe zu überzeugen

- ▶ **Empathize**
In Problem hineinversetzen und Verständnis für dieses entwickeln
- ▶ **Define**
Team formuliert Problem neu, um ein gemeinsames Verständnis der Details zu erhalten
- ▶ **Ideate**
Team generiert möglichst viele Ideen und wählt die aussichtsreichste(n) aus
- ▶ **Prototype**
Team erstellt einfache, realistische und kostengünstige Prototypen
- ▶ **Test**
Team testet den Prototyp mit Kunden, um Feedback einzuholen

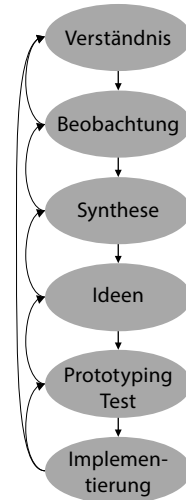


Abb. 9–8 Design-Thinking-Methodik

Offensichtlich bedient sich Design Thinking aus dem klassischen Prototyping, aber die gezielte Wiederverwendung bekannter guter Praktiken gilt ja für die agile Entwicklung im Allgemeinen. Klar ist, dass wie im klassischen Prototyping die Ergebnisse zunächst mal »Wegwerf-Prototypen« darstellen und in der Regel untauglich als späteres Produkt sind. Planen Sie im Design Thinking daher genug Zeit ein, um nochmals von vorne zu beginnen – jetzt auch unter Berücksichtigung der vorher ignorierten Randbedingungen und vor allem der Qualitätsanforderungen.



Beispiel:

Ein Kunde hatte am Markt das Image eines Spezialisten, der hochgradig kundenspezifische Lösungen liefern kann, die von den großen OEMs als zu teuer erkannt wurden. Sein Problem: Die Produkte waren auch für ihn zu teuer. Alles wurde spezifisch angepasst und produziert. Die Komplexität musste schnellstens reduziert werden, bevor es zum Kollaps kam. Allein die Kosten für Operations und Supply Chain zeigten, dass das Geschäftsmodell nicht mehr trug. Die Märkte weltweit liebten seine Lösungen als Kundenversther, aber wollten sie nicht mehr bezahlen. Wir unterstützten zunächst mit einem Benchmark und sprachen mit Kunden weltweit, um alternative Geschäftsmodelle zu bewerten. Schnell war klar, dass es modulare Lösungsbausteine brauchte, die wiederverwendet werden können. Doch wie sollten die Lösungselemente aussehen? Wir nutzten Design Thinking mit Prototypen möglicher Lösungselemente und spielten typische Szenarien durch, die wir anhand der jeweiligen Volumina und Kosten bewerteten. Das Senior-Management spielte noch nicht ganz mit und beklagte, dass mit »Lego-Lösungen« das Image ruiniert würde. Wir hinterfragten in gezielten Managementseminaren funktionsübergreifend die Alternativen mit den 5 Why. Die Überzeugungsarbeit war schnell erledigt, und wir konnten das nötige Change-Projekt zur Umsetzung starten.

9.5 Skalierbare Agilität

Agilität muss an die Randbedingungen der Zielumgebung angepasst werden. Das braucht Praxiserfahrung. Da helfen weder Lehrbücher noch geschlossene Methoden, wie XP oder SAFe. Vorgaben wie »ein Team an einem Ort« oder »Kunde an Bord« sind gut gemeint, aber nicht realistisch in Zeiten verteilter Entwicklung und Crowdsourcing. Egal, um welchen Prozess, er muss manuell und mit viel Erfahrung an Ihre Umgebung angepasst werden [Ebert2012, Ebert2020].

Prozesse scheitern nicht wegen der zugrunde liegenden Prinzipien, sondern wegen der falschen Anwendung. Konkret: Agil ist definitiv die Lösung für Effizienz, Motivation und Flexibilität – aber nicht so, wie es viele Unternehmen machen. Beispielsweise benötigen sicherheitskritische Systeme in Bahn, Automotive, Medizintechnik und Luftfahrt eine gründliche Dokumentation, die in agilen Rezeptsammlungen in der Regel verworfen wird. Governance und Nachvollziehbarkeit benötigen schlanke Vorgehensweisen mit guter Nachvollziehbarkeit. Skalierbarkeit lässt sich entlang von zwei Dimensionen erzeugen (Abb. 9–9).

Für hohe Flexibilität werden kontinuierliche Prozesse für Entwicklung und Lieferung umgesetzt (horizontale Achse in Abb. 9–9). Damit sind auch leichtgewichtige agile Methoden anwendbar, beispielsweise ein einfaches Scrum mit inkrementellen Lieferungen. Wachsende Risiken (beispielsweise aus der Produkthaftung) fordern Nachvollziehbarkeit und Governance. In kritischen Branchen, wie Medizin, Automotive oder Transport, genügen die einfachen agilen Methoden aus der Anwendungsentwicklung nicht. Definierte Lebenszyklen werden mit agilen Methoden angereichert (vertikale Achse in Abb. 9–9).

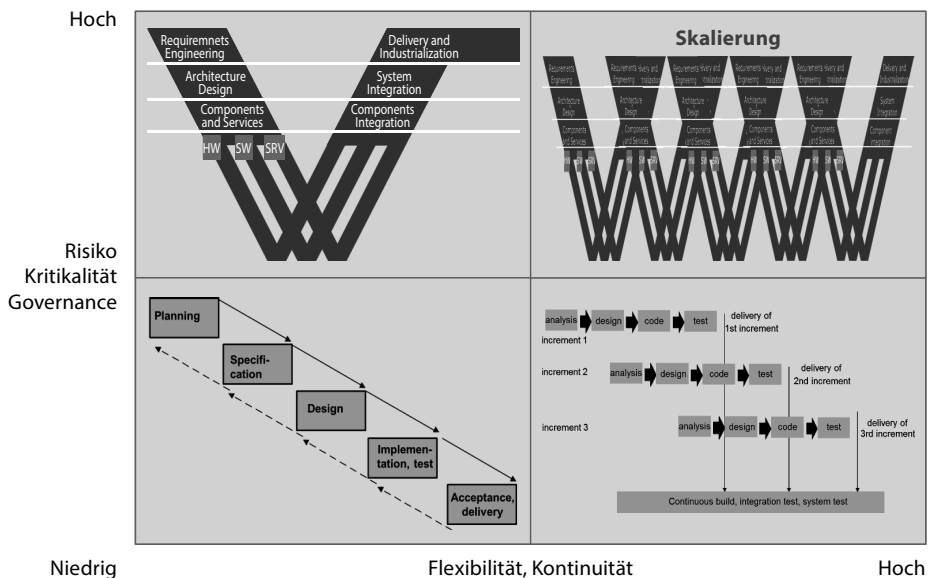



Abb. 9–9 Skalierbare Agilität im Lebenszyklus

Diese Leseprobe haben Sie beim
 **edv-buchversand.de** heruntergeladen.
Das Buch können Sie online in unserem
Shop bestellen.

[Hier zum Shop](#)