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].

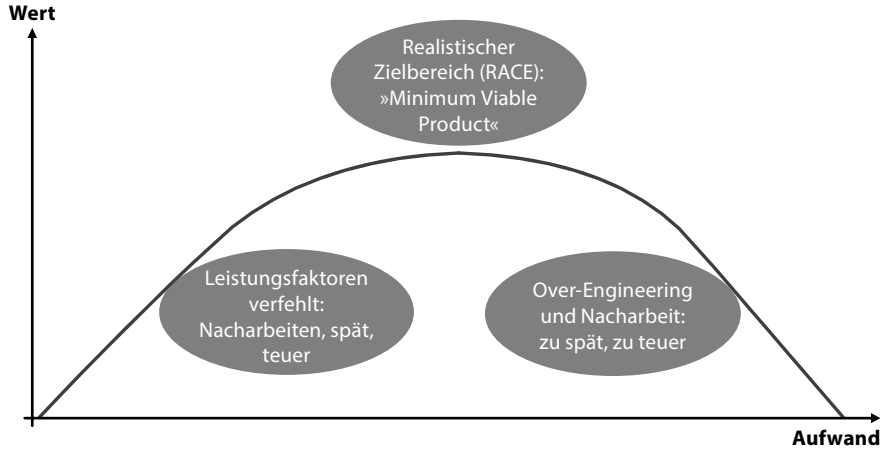

Abb. 9–1 *Lean und agil: Wert für den Kunden optimieren*

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).

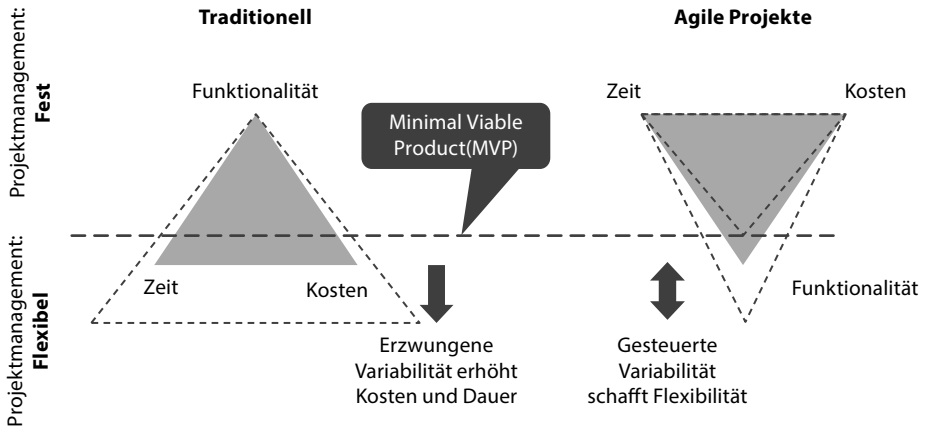

Abb. 9–2 *Agile Projektsteuerung: das magische Dreieck*

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.

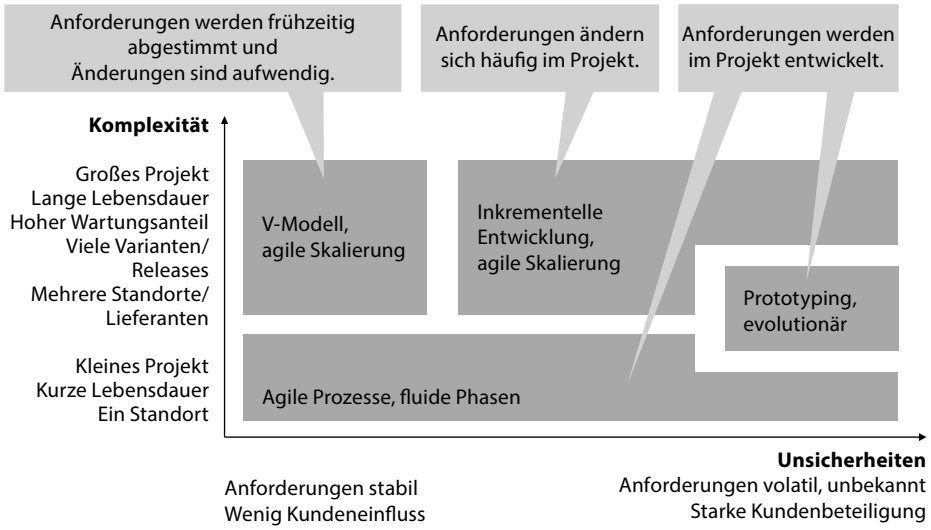

Abb. 9–4 *Anforderungen bestimmen die Methodik*

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.

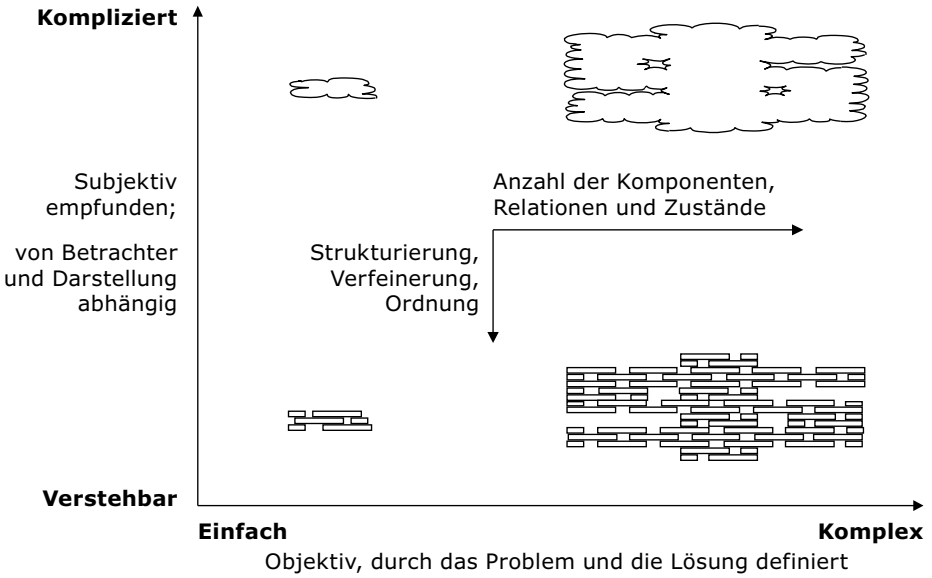

Abb. 9–5 *Komplexität und Kompliziertheit*

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).

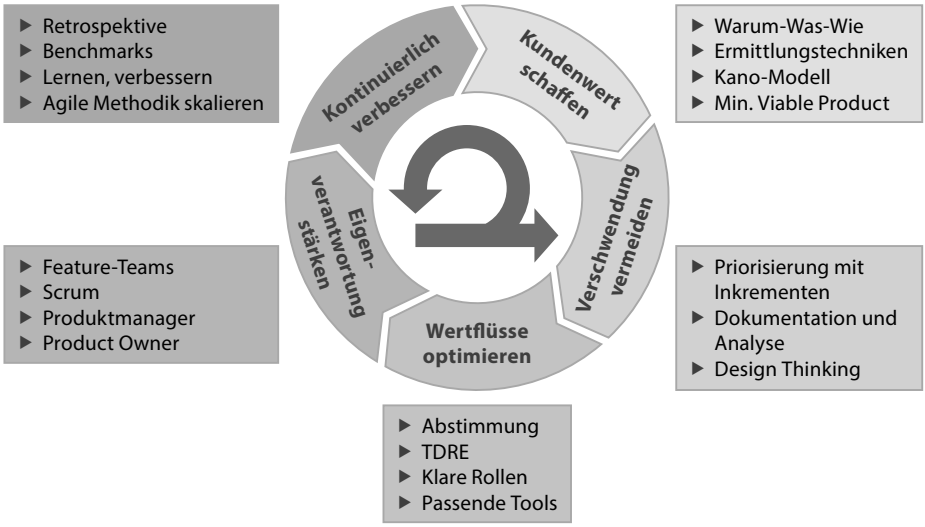

Abb. 9–6 *Agile Techniken für das Requirements Engineering*

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.

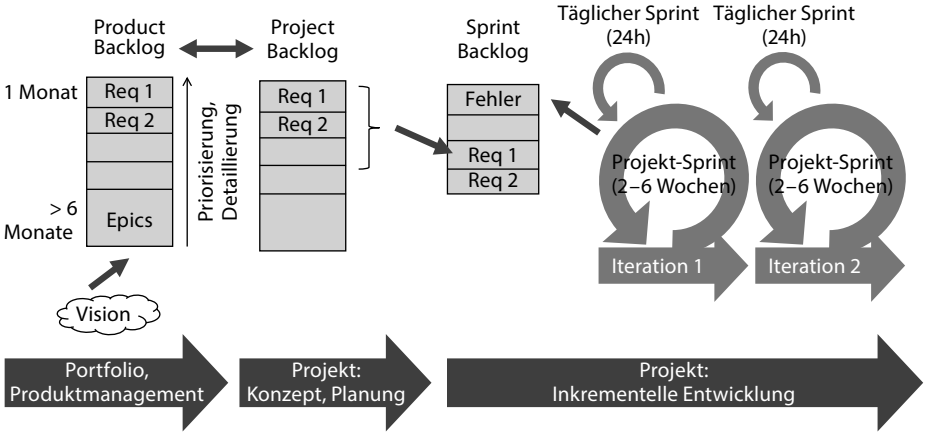

Abb. 9–7 *Kaskadierte Scrums für Projekte und Produkte*

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

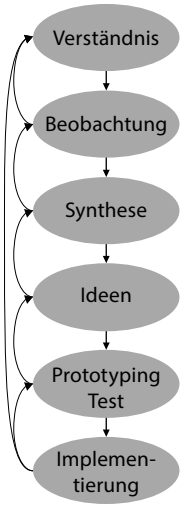

Abb. 9–8 *Design-Thinking-Methodik*

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 Kundenversteher, 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).

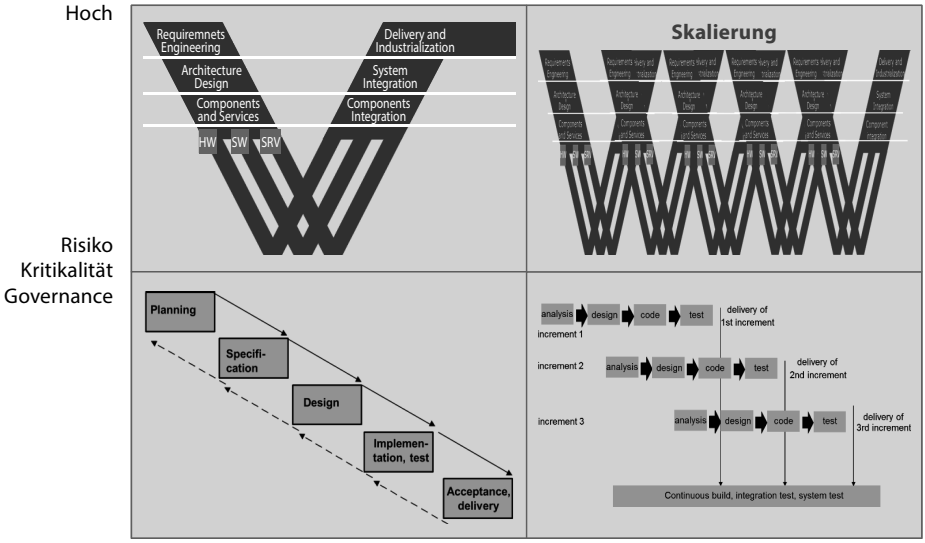

Abb. 9–9 *Skalierbare Agilität im Lebenszyklus*