10 Prozessmodelle

Die Vorgehensmodelle (siehe Kap. 9) sind geeignet, dem Projektleiter und den Entwicklern Hinweise zu geben, welche Tätigkeit als nächste auszuführen ist. Sie machen aber keine Aussagen über

- die personelle Organisation,
- die Gliederung der Dokumentation,
- die Verantwortlichkeiten für Aktivitäten und Dokumente.

Nimmt man diese Punkte zur Vorgabe des Ablaufs hinzu, so entsteht aus dem Vorgehensmodell ein Prozessmodell. Wenn dieses für eine Firma oder eine Abteilung verbindlich vorgegeben, also standardisiert ist, wird es möglich,

- Planungen (halbfertig) »von der Stange« zu verwenden,
- Werkzeuge und Methoden gemeinsam einzuführen,
- Resultate zu vergleichen und auszutauschen,
- Schwachstellen zu erkennen und aus Erfahrungen zu lernen.

Die Literatur bietet eine Reihe sehr unterschiedlicher Prozessmodelle an. Bevor man eines davon übernimmt, sollte man aber in jedem Fall den Prozess, wie er bisher verstanden und gelebt wird, bestimmen und analysieren, um seine Stärken und Schwächen zu erkennen, damit man nicht am Ende den Spatzen in der Hand preisgegeben hat, um – mit unsicherem Ausgang – die Taube auf dem Dach zu jagen. Änderungen des Prozesses sind mühsam und riskant. Wenn man sich darauf einlässt, sollte man genau wissen, was man erreichen will und was man verlieren kann.

In der öffentlichen Diskussion ist das Thema der Prozesse – ähnlich wie früher, manchmal auch heute noch, das Thema der Programmiersprachen – durch Heilslehren, Glaubenskriege und Lagerdenken geprägt. Das macht eine rationale Auseinandersetzung mit den Konzepten schwierig und eine Synthese nahezu unmöglich. Natürlich sind auch die Verfasser dieses Buches nicht frei von Emotionen. Aber wir bemühen uns um eine rationale Wertung.

10.1 Begriffe und Definitionen

10.1.1 Das Prozessmodell und seine Ausprägung im Prozess

Ein (Software-)Prozessmodell ist die Beschreibung eines Software-Prozesses als präskriptives Modell für die Durchführung der Projekte. Da ein solches Modell nicht alle Details vorgeben kann, weil sie nach den konkreten Bedingungen der speziellen Projekte unterschiedlich ausgeprägt werden müssen, lässt es mehr oder minder große Spielräume für die konkrete Ausgestaltung. Entsprechend kann ein Prozessmodell sehr karg sein, also nur einige zentrale Punkte vorgeben, oder sehr detailliert, sodass die Projekte kaum Unterschiede aufweisen.

Überall, wo das Prozessmodell keine (genauen) Vorgaben macht, müssen sie für jedes Projekt oder für eine Gruppe sehr ähnlicher Projekte hinzugefügt werden. Diesen Vorgang bezeichnet man als *Prozessausprägung* (*Tailoring*). Durch die Prozessausprägung wird aus dem generischen Prozessmodell ein konkreter Prozess, der im Idealfall eins zu eins im Projekt umgesetzt wird. Natürlich muss dieser konkrete Prozess geprüft und abgenommen werden, bevor er zum Einsatz kommt.

Ob es sinnvoll ist, mit den Vorgaben sehr weit zu gehen, hängt von den Rahmenbedingungen ab: Ein Software-Haus, das für eine heterogene Kundschaft arbeitet, braucht große Freiheit bei der Ausprägung der Prozesse, weil der eine Kunde verlangt, was der andere auf keinen Fall haben will. Dagegen profitiert eine Bank, die ein riesiges Software-System einsetzt, von einer weit reichenden Standardisierung. Auch die Kultur der Firma spielt eine wichtige Rolle: Entwickler mit Cowboy-Mentalität lassen sich kaum auf einen rigiden Prozess ein, während viele disziplinierte Programmierer einen solchen Prozess begrüßen.

Es ist daher nicht möglich, eine allgemeingültige Festlegung zu treffen, was alles zu einem Prozessmodell gehört. In der Regel finden sich darin Aussagen zu folgenden Punkten:

- Organisation und Verantwortlichkeiten, Rollenverteilung
- Struktur und Merkmale der Dokumente
- Einzusetzende Verfahren, z. B. für die Erhebung der Anforderungen oder für die Prüfung der Zwischenergebnisse
- Die auszuführenden Schritte der Entwicklung, ihre Reihenfolge und ihre Abhängigkeiten (das Vorgehensmodell)
- Projektphasen und Meilensteine, Prüfkriterien
- Notationen und Sprachen
- Werkzeuge

Mindestens implizit ist durch das Prozessmodell auch eine bestimmte Terminologie vorgegeben, beispielsweise für die Rollen und die Dokumente im Projekt.

Das Feld der Prozessmodelle ist unübersichtlich und nicht abgegrenzt. Täglich werden neue erdacht, vielleicht ausprobiert und meist verworfen; nur gelegentlich schafft eines den Sprung in die Fachwelt, wird Gegenstand von Publikationen und Kursen und schließlich auch eingesetzt. Aber nur ganz selten sind die neuen Prozessmodelle wirklich neu. In der Regel werden bekannte Konzepte neu kombiniert, neu etikettiert – und als Wundermittel angeboten.

10.1.2 Leichte und schwere Prozesse

Im Zusammenhang mit den sogenannten *agilen Prozessen* ist die Unterscheidung zwischen *leichten* und *schweren* Prozessmodellen aufgekommen. Leider ist unklar, worauf sich diese Attribute beziehen. Ganz offensichtlich sind alle Prozessmodelle, die von ihren Verfassern als leicht eingestuft werden, durch einen geringen Aufwand für die Dokumentation gekennzeichnet. Das ist natürlich kein Wert an sich. Wer bei schönem Wetter durch Kalifornien fährt, fühlt sich in einem leichten, womöglich offenen Wagen sehr wohl; wer eine Expedition durch die Wüste Taklamakan unternimmt, braucht ein sehr stabiles Fahrzeug mit Allradantrieb, das große Mengen an Lebensmitteln, Zelten, Treibstoff, Werkzeug und Ersatzteilen befördern kann. Die Entscheidung, welche Dokumentation erforderlich ist, muss also unter dem Gesichtspunkt getroffen werden, welche später mutmaßlich benötigt wird.

Weit verbreitet ist auch die Vorstellung, dass »schwere Prozesse« eigentlich schwerfällige, wenig flexible Prozesse sind. Natürlich ist ein Prozessmodell, das eine vollständige Planung zu Beginn des Projekts vorsieht, hinderlich, wenn der Projektverlauf immer wieder Änderungen des Plans erfordert. Aber auch für die Flexibilität gilt, was oben zur Dokumentation gesagt wurde: Wenn es darum geht, einen wankelmütigen Kunden zu bedienen, ist Flexibilität notwendig; wenn aber ein komplexes System aus Hardware, Software und nicht zuletzt menschlicher Organisation entstehen soll, ist die Stabilität und Zuverlässigkeit der Arbeiten, die getrennt ausgeführt werden, vorrangig.

Eine weitere Interpretation des Wortes »leicht« bezieht sich auf die Menge der Regeln. Allerdings zeigt sich, dass die »leichten Prozesse« keineswegs besonders wenige Regeln haben, sie haben nur andere als die vermeintlich schweren Prozesse.

Schließlich gibt es – beim V-Modell, siehe Abschnitt 10.3 – die verwirrende Situation, dass ein nach landläufiger Einschätzung schweres Prozessmodell als eine von mehreren Möglichkeiten das Vorgehen nach dem Konzept der leichten Modelle anbietet. Die Situation ist also alles andere als klar. Wir machen darum keinen Versuch, die Prozesse in das Schema »leicht – schwer« einzuordnen.

10.2 Das Phasenmodell

In Abschnitt 8.3.2 haben wir bereits beschrieben, dass Phasen und Meilensteine wichtige Elemente der Projektplanung sind. Nachfolgend gehen wir auf diese für die Prozessmodelle wichtigen Konzepte genauer ein.

10.2.1 Phasen und Meilensteine

Eng mit dem Software-Lebenslauf verbunden ist die Vorstellung, dass er in Abschnitte, sogenannte *Phasen,* gegliedert ist, an deren Grenzen wesentliche Änderungen eintreten. Am auffälligsten ist eine solche Änderung, wenn sie als sichtbare Metamorphose verläuft wie bei einer Libelle, die sich aus einer Larve befreit. Phasen sind also Abschnitte des Lebenslaufs, die einen Anfang und ein Ende haben, nicht unterbrochen werden können und nicht überlappen.

Auch eine längere Wanderung ist in Etappen (= Phasen) gegliedert. Wenn ein *Meilenstein* (allgemeiner ein markanter Punkt mit bekannter Position) erreicht ist, endet die eine und beginnt die nächste Etappe. Bei der Planung der Wanderung schätzen wir ab, wie viel Zeit und wie viel Wegzehrung wir für jede Etappe brauchen werden. Sicher vorhersagen können wir aber weder das eine noch das andere. Der Weg kann weit schwieriger sein als erwartet, oder wir verlaufen uns. Dann erreichen wir den Meilenstein mit Verspätung und haben bereits die Vorräte für die nächste Etappe verzehrt. Der Meilenstein steht, wo er steht, er kommt uns nicht entgegen, wenn wir langsam gehen. Darum ist die Phase der Weg bis zu einem bestimmten Ziel. Wie viel Zeit dieser Weg benötigt, ist im Allgemeinen erst klar, wenn das Ziel erreicht ist.

Die Gliederung einer Wanderung in Etappen hat einige Vorteile:

- Wir stellen fest, ob unsere Zeitschätzung realistisch war. Bei großer Abweichung korrigieren wir die Planung für die folgenden Etappen.
- Wir stellen fest, dass wir noch auf dem richtigen Weg sind.
- Wir teilen uns die Wegzehrung sinnvoll ein.
- Wenn wir langsamer vorankommen, als geplant war, können wir eventuell eine Abkürzung nehmen oder einen Teil des Weges fahren statt gehen.
- Wenn wir schon bis zum ersten oder zweiten Meilenstein viel zu lange brauchen, brechen wir die Wanderung ab, bevor wir in große Schwierigkeiten geraten.

10.2.2 Phasen und Meilensteine im Software-Projekt

Das Bild der in Etappen gegliederten Wanderung können wir komplett in das Projektmanagement übernehmen. Wir legen Zwischenziele, *Meilensteine,* fest, und schätzen ab, wie viel Zeit und welche Mittel wir für jede Phase brauchen. Dann führen wir das Projekt entsprechend der Planung durch und verfolgen

dabei, ob wir planmäßig vorankommen. Bei kleinen Verzögerungen versuchen wir in der Regel, die Leistung zu erhöhen, sodass wir den Rückstand aufholen. Bei großen Abweichungen müssen wir mehr Zeit vorsehen (also die Termine verschieben) oder die Projektziele reduzieren oder das Projekt abbrechen.

Auch im Software-Projekt gilt für die Meilensteine, dass sie durch die Ziele, nicht durch den Termin definiert sind. Ein Meilenstein »Spezifikation fertig« mag für den 31. Oktober geplant sein; erreicht ist er nicht am 31. Oktober, sondern erst, wenn die Spezifikation fertig ist. Dazu sind klare Kriterien anzugeben; es reicht nicht aus, wenn die Entwickler versichern, dass alle Ziele erreicht sind. Erst wenn alle Teilresultate vorliegen und abgenommen sind, kann der Meilenstein gefeiert werden.

Wir folgen bei der Definition des Meilensteins Wallin et al. (2002):

A **milestone** is defined as a scheduled event that marks the completion of one or more important tasks, and it is used to measure achievements and development progress. At a milestone, a predefined set of deliverables should have reached a predefined state to enable a review.

Wir stellen aber klar, dass das Review oder allgemeiner die (bestandene) Prüfung eine Voraussetzung, keine Folge des Meilensteins ist. Für jeden Meilenstein wird definiert,

- welche Ergebnisse (Dokumente) vorliegen müssen,
- wer nach welchen Kriterien welche Prüfung vornimmt und
- wer schließlich entscheidet, ob der Meilenstein erreicht ist.

Die Definition der *Phase* ist unmittelbar an die des Meilensteins geknüpft: Eine Phase ist der Zeitraum zwischen zwei Meilensteinen. Zwischen Beginn und Ende der Phase werden die Arbeiten durchgeführt, deren Ergebnisse vor dem Meilenstein geprüft werden. Am Meilenstein »Spezifikation fertig« (das könnte der Meilenstein M1 in Abb. 10–1 sein) muss die Spezifikation geprüft sein. Also ist die Phase 1 die Spezifikationsphase.

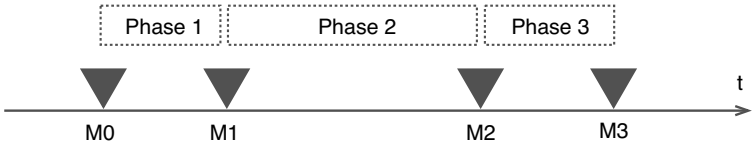

Abb. 10–1 *Das Konzept der Phasen und Meilensteine*

Das Phasenmodell kann damit wie folgt charakterisiert werden:

Die Software-Entwicklung wird vor Beginn in Phasen gegliedert, die streng sequenziell durchlaufen werden. Für jede Phase gibt es ein eigenes Budget; dieses Budget wird erst freigegeben, wenn der vorangehende Meilenstein erreicht ist. Dann kann die nächste Phase beginnen.

Wegen der strikten Verknüpfung der Phasen mit den Budgets wird das Phasenmodell auch *Kostenmodell* genannt. In der Planung werden die Phasen durch Arbeitspakete (siehe Abschnitt 8.3.1) verfeinert; das kann teilweise erst im Projektverlauf geschehen, wenn die Strukturen des Systems geklärt sind.

Mit dem Begriff des Phasenmodells ist keine Festlegung auf eine bestimmte Definition oder Reihenfolge der Phasen verbunden. Ausprägungen des Phasenmodells können also sehr unterschiedlich aussehen.

Die logische Sequenz Phasenanfang – Entwicklung – Prüfung – Phasenende suggeriert, dass die Prüfungen – wie eine Abiturprüfung am Ende einer langen Schulzeit – jeweils unmittelbar vor dem Meilenstein stattfinden. Das ist falsch. Auch innerhalb einer Phase gibt es Teilergebnisse, die sofort geprüft werden sollten. Am Ende müssen alle Prüfungen abgeschlossen sein, sie können aber bereits weit früher durchgeführt werden.

10.2.3 Abgrenzung zum Wasserfallmodell

Im Beispiel oben war von der Spezifikationsphase die Rede. Haben wir also einfach das Wasserfallmodell mit Meilensteinen garniert, aber im Übrigen unverändert übernommen? In Abschnitt 9.2.1 hatten wir bereits festgestellt, dass das nicht funktioniert. Tatsächlich gibt es zwei wesentliche Unterschiede zwischen dem Phasenmodell und dem Wasserfallmodell:

- Im Phasenmodell gibt es keine Zyklen. Eine abgeschlossene Phase kann nicht wieder geöffnet werden, sie ist Vergangenheit. Darum dokumentiert das Erreichen eines Meilensteins einen Fortschritt des Projekts.
- Da kein Prozessmodell der Welt verhindern kann, dass Fehler gemacht werden, die erst später auffallen, oder dass der Kunde neue Anforderungen vorträgt, obwohl die Analyse längst abgeschlossen ist, oder dass aufgrund neuer Einsichten frühere Entscheidungen revidiert werden müssen, kann das Phasenmodell die Rückkehr in frühere *Tätigkeiten* nicht verbieten. Nur finden diese Tätigkeiten im Budget und unter den Bedingungen der gerade laufenden Phase statt. Das bedeutet: Sie werden sehr strikt überwacht, der Entwickler arbeitet quasi unter Aufsicht.

Auf den ersten Blick erscheint das wie ein Etikettenschwindel: Wir dürfen zwar nicht die Zyklen des Wasserfallmodells durchlaufen, sondern müssen in der Phase bleiben, in der wir sind. Aber wir können trotzdem die früheren Tätigkeiten fortsetzen. Dieser Eindruck täuscht. Wenn ein Entwickler bei der Arbeit an der Spezifikation einen Fehler im entstehenden Dokument bemerkt, klärt er die Sache auf und beseitigt den Fehler. Ganz anders sieht es aus, wenn sich erst bei der Codierung ein Fehler der Spezifikation zeigt. Die Entwickler können die Spezifikation nicht einfach korrigieren, auch dann nicht, wenn sie selbst die Verfasser sind. Der Fehler wird dokumentiert, und der Verantwortliche muss entscheiden, ob er

behoben werden soll. (Manchmal ist es sinnvoller, eine bestimmte Funktion abzuschalten oder eine Warnung ins Handbuch zu drucken.) War die Entscheidung positiv, so wird die Spezifikation geändert. Dann wird sie nach den gleichen Kriterien geprüft, die am Meilenstein am Ende der Spezifikationsphase gültig waren. Alle, die auf der Basis der fehlerhaften Spezifikation gearbeitet hatten, z. B. die Codierer, die Tester, die Verfasser des Handbuchs, werden über die Änderung informiert, damit sie prüfen können, ob ihre Ergebnisse davon betroffen sind. Schließlich ersetzt die neue Version der Spezifikation die alte. Die Änderung eines bereits geprüften Dokuments ist also wesentlich aufwendiger. Der Mehraufwand ist durch die großen Risiken gerechtfertigt, die mit einer unkontrollierten Änderung verbunden sind.

Natürlich spielt aus den genannten Gründen die Konfigurationsverwaltung (siehe Kap. 20) im Phasenmodell eine wichtige Rolle. Sie garantiert, dass das *Baselining* zuverlässig funktioniert, also das »Einfrieren« der Dokumente, die gerade geprüft werden oder bereits geprüft und abgenommen sind.

10.2.4 Vor- und Nachteile des Phasenmodells

Das Phasenmodell bringt erheblichen Aufwand für die Organisation und für die Prüfungen mit sich. Dafür gibt es eine Reihe von Vorteilen:

- Das Projekt ist präzise geplant und organisiert. Abweichungen von der Planung sind leicht erkennbar, weil sie sich spätestens dann zeigen, wenn ein Meilenstein nicht wie geplant erreicht wird.
- Die Dokumente, deren Erstellung geplant wurde, werden geprüft; entsprechen sie nicht den Anforderungen, ist der Meilenstein noch nicht erreicht. Damit ist sichergestellt, dass die Dokumente nach dem Meilenstein vorhanden sind und den Anforderungen genügen. Das ist in weniger systematisch geführten Projekten keineswegs selbstverständlich.
- Der Personalbedarf ist ebenfalls frühzeitig geklärt. Damit kann sichergestellt werden, dass die Entwickler zur richtigen Zeit verfügbar sind und später auch wieder rechtzeitig an andere Projekte abgegeben werden können.
- Die Durchführung der Prüfungen ist gewährleistet. Damit ist das Risiko, am Projektende zu erkennen, dass in die falsche Richtung gearbeitet wurde oder dass eine Komponente viel zu spät fertig wird, ganz erheblich reduziert.
- Das *90 %-fertig-Syndrom* ist ausgeschlossen. In Projekten ohne Meilensteine neigen die Entwickler dazu, den Entwicklungsstand als »zu 90 % fertig« zu charakterisieren. Rückblickend erkennt man, dass diese Aussage über lange Zeit unverändert bleibt (Abb. 10–2). Der Grund ist, dass zu Beginn die leichteren Aufgaben gelöst werden. Am Ende bleiben die schwierigen übrig; durch die Restriktionen, die alle bereits realisierten Teile schaffen, wird die Schwierigkeit erhöht. Dieses seit Langem bekannte Phänomen ist treffend durch den Satz beschrieben: *Die ersten 90 % des Projekts brauchen 90 % der Zeit. Die*

letzten 10 % des Projekts brauchen die anderen 90 % der Zeit. Meilensteine können diese Neigung der Entwickler nicht beseitigen, aber auf die einzelnen Phasen begrenzen.

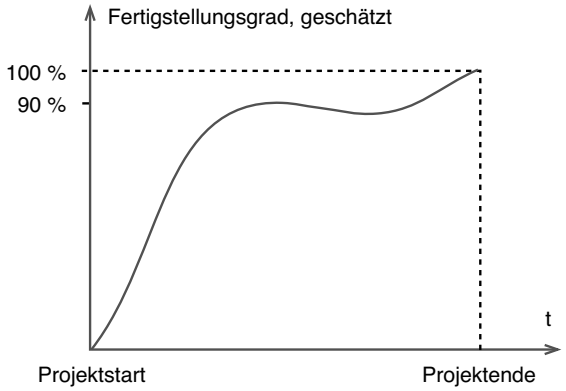

Abb. 10–2 *Das 90 %-fertig-Syndrom*

Die Vorteile sind so gewichtig, dass man eine Entwicklung ohne Phasenmodell als ausgesprochen riskant einstufen muss. Die Festlegung auf ein Phasenmodell klärt allerdings noch nicht viel; es schafft nur einen weiten Rahmen, innerhalb dessen noch sehr viele Festlegungen zu treffen sind. Ausgeschlossen sind damit nur die optimistischen Ansätze nach dem Beckenbauer-Prinzip (»*Schaun mer mal*«).

10.2.5 Überlappende Phasen

In einem Software-Projekt werden je nach Einteilung bis zu acht Phasen durchlaufen, von der Anforderungsanalyse bis zum Betrieb. Da die den Phasen zugeordneten Tätigkeiten aufeinander aufbauen, liegt es nahe zu verlangen, dass sie strikt sequenziell, eine nach der anderen, durchgeführt werden. Praktisch sprechen drei Gründe gegen diesen »strengen« Ansatz:

1. Oft stehen notwendige Informationen nicht zur richtigen Zeit zur Verfügung (z. B. werden Anforderungen spät im Projekt zugefügt oder geändert).
2. Vor einem Meilenstein sind die meisten Beteiligten bereits mit ihren Arbeiten fertig und müssen warten, bis alle anderen ebenfalls den Meilenstein erreicht haben, bevor sie weiterarbeiten können.
3. In vielen Fällen ist es vorteilhaft, möglichst rasch Teilergebnisse zur Verfügung zu stellen.

Darum wird das Phasenmodell – viele sprechen ungenau vom Wasserfallmodell – »aufgeweicht«. Dies kann in unterschiedlicher Weise geschehen:

a) Wenn sich das geplante System aufspalten lässt in unbedingt notwendige und nicht unbedingt notwendige Teile, dann kann inkrementell entwickelt werden (siehe Abschnitt 9.5.4).
b) Teile, die keine oder nur geringe Abhängigkeiten aufweisen, können asynchron entwickelt werden, also je nach verfügbarem Personal gleichzeitig oder in beliebiger Reihenfolge.
c) Dokumente werden bereits verwendet, obwohl sie noch nicht als vollständig einzustufen sind. Dies wird als *Überlappung* bezeichnet.
d) Die Arbeiten werden ohne Rücksicht auf logische Abhängigkeiten nach Verfügbarkeit der Bearbeiter in beliebiger Reihenfolge ausgeführt.

Natürlich ist Vorgehen d inakzeptabel, weil es jegliche Qualitätssicherung konterkariert. (Tatsächlich wird diese Strategie *offiziell* nirgends angewendet, aber praktisch ist sie nicht selten zu beobachten.)

Das Vorgehen a ist sehr sinnvoll, weil es zu kürzeren Entwicklungszyklen führt und die Möglichkeit bietet, die Erfahrungen der Klienten in den späteren Inkrementen zu berücksichtigen. Allerdings setzt das inkrementelle Vorgehen eine präzisere Planung voraus, denn die Schnittstellen zu den später entwickelten Teilen sollten bereits beim Kernsystem stabil sein.

Das Vorgehen b ist sinnvoll und wird in aller Regel angewendet. Natürlich gewinnt dadurch die Integration an Bedeutung: Missverständnisse und Fehler der Komponentenentwicklung verursachen Integrationsprobleme. Da eine seriöse Bearbeitung dieser Probleme nur durch die Entwickler erfolgen kann, entstehen zeitraubende Nacharbeitszyklen. Alternativ werden Inkonsistenzen durch Bastelei (*scheinbar*) kompensiert.

Beim Vorgehen c ist weiter zu differenzieren: Die Überlappung muss durch definierte Änderungsprozesse abgepolstert sein, damit sichergestellt ist, dass späte Änderungen der bereits verwendeten Dokumente nicht ignoriert werden, sondern ins Endprodukt eingehen. Wird beispielsweise die Spezifikation verändert, während bereits auf Basis einer früheren Version implementiert wurde, so muss der Prozess sicherstellen, dass

- die Spezifikation erneut validiert wird,
- alle Entwickler über die Änderung informiert werden,
- Auswirkungen auf alle bereits fertiggestellten Teile geprüft werden.

Eine Überlappung ist nicht an allen Phasengrenzen möglich und zulässig: Während beispielsweise Spezifikation und Implementierung überlappen können (auch wenn das keineswegs unproblematisch ist), ist der Eintritt in die Prüfung völlig unsinnig, solange die Spezifikation nicht vollständig vorliegt. Einer Prüfung, die sich nicht auf die gültige Spezifikation abstützt, fehlt einfach die Basis. Es können aber Teile geprüft werden, deren Spezifikation bereits stabil ist.

10.3 Das V-Modell

10.3.1 Hintergrund und Geschichte

Das V-Modell[1] ist Teil des Entwicklungsstandards für IT-Systeme des Bundes. Das »V« hat anscheinend eine doppelte Bedeutung: Einerseits steht es für »Vorgehen«; tatsächlich ist aber durch das V-Modell weit mehr als das Vorgehen geregelt, es handelt sich um ein ausgewachsenes Prozessmodell. Zum anderen hatte in der ersten Fassung des V-Modells die V-förmige Darstellung des Entwicklungsganges zentrale Bedeutung; sie ist noch immer zu erkennen (siehe Abb. 10–7; zum »V« vgl. die Fußnote zur Badewannenkurve in Abschnitt 4.4, S. 63).

Das Interesse öffentlicher Auftraggeber an Prozess-Standards ist leicht zu verstehen: Die Ministerien, allen voran das Verteidigungsministerium, geben Entwicklungen in Auftrag. Dabei müssen sie einerseits die Kosten im Auge behalten, streng genommen also stets den billigsten Anbieter wählen, andererseits sicherstellen, dass sie Waren und Dienstleistungen in angemessener Qualität erhalten. Schließlich müssen sie die Entscheidung für einen Anbieter nach objektiven Kriterien fällen, die notfalls auch vor einem Verwaltungsgericht Bestand haben. Darum suchen sie nach klaren Vorgaben und Kriterien für ihre Zulieferer. In den USA ist aus den gleichen Gründen das CMM entstanden (siehe Kap. 11).

Das V-Modell wurde im Auftrag des Bundesministeriums für Verteidigung entwickelt und ist dort seit 1992 verbindlich vorgeschrieben (Bröhl, Dröschel, 1995). Im Sommer 1996 wurde es auch für den Einsatz im zivilen Verwaltungsbereich der Bundesbehörden empfohlen. Damit sind auch die Lieferanten in der Regel an das V-Modell gebunden. Da es öffentlich zugänglich ist, haben auch andere Unternehmen das V-Modell erprobt und eingesetzt. Die damit gemachten Erfahrungen wurden systematisch gesammelt. Auf dieser Basis wurde es überarbeitet und unter der Bezeichnung V-Modell 97 publiziert (V-Modell, 1997). Zu den Verbesserungen dieser Version zählten die Unterstützung der inkrementellen Entwicklung, die koordinierte Entwicklung von Hard- und Software und die objektorientierte Entwicklung. Dröschel, Heuser und Midderhoff (1998) haben die wesentlichen neuen Elemente des V-Modells 97 beschrieben. 2004 wurde als Ergebnis einer weiteren Revision das V-Modell XT vorgestellt. XT steht hier für »extreme Tailoring«. Das drückt zum einen die Flexibilität beim Tailoring (Abschnitt 10.1.1) aus, die im neuen Modell angestrebt wurde; Rausch und Niebur (2005) nennen als erstes von vier Zielen »Verbesserung der Unterstützung von Anpassbarkeit, Anwendbarkeit, Skalierbarkeit und Änder- und Erweiterbarkeit des V-Modells«. Zum anderen war es wohl das Bestreben der gewiss nicht extremistischen Urheber, am modischen Trend zum »X« teilzuhaben. Die folgende Darstellung stützt sich überwiegend auf

1 Das V-Modell® XT ist eine geschützte Marke der Bundesrepublik Deutschland.

das vom Verein zur Weiterentwicklung des V-Modell XT e.V. (Weit e.V.) publizierte V-Modell-XT-Referenzdokument (V-Modell XT, o. J.).

Da es beim V-Modell auch um die Absicherung eines System- oder Software-Bestellers geht, sind darin die Beziehungen zwischen Auftraggeber (AG) und Auftragnehmer (AN) besonders wichtig.

10.3.2 Eigenschaften des Modells

Die folgenden Merkmale charakterisieren das V-Modell und das V-Modell XT:

- Das V-Modell ist in allen seinen Versionen als Weiterentwicklung des Standard-Phasenmodells ein aktivitätsorientiertes Modell. Ein logisch verknüpftes Netz von Aktivitäten und Produkten bildet den Kern des Modells.
- Das V-Modell XT teilt ein Projekt in Projektabschnitte, also in Phasen, ein. Am Ende jedes Projektabschnitts steht ein Meilenstein, der »Entscheidungspunkt« genannt wird.
- Das V-Modell kann für verschiedene Projektarten genutzt werden. Während sich das V-Modell in der Fassung von 1992 darauf beschränkte, die Software-Entwicklung zu beschreiben, schließt es seit der Version 97 auch die Hardware-Entwicklung mit ein.
- Das V-Modell deckt nicht nur die technische Systementwicklung ab. Die wesentliche Neuerung, die das V-Modell vom Wasserfallmodell unterscheidet, ist die Integration projektbegleitender Tätigkeiten, insbesondere der Qualitätssicherung, der Konfigurationsverwaltung und des Projektmanagements.
- Das V-Modell XT unterstützt neben der inkrementellen und komponentenbasierten auch die prototypische Entwicklung. Das traditionelle Wasserfallmodell wird als Sonderfall der inkrementellen Entwicklung betrachtet. Die Entwicklung selbst kann klassisch oder agil erfolgen.
- Das V-Modell lässt sich – besonders in seiner neuesten Fassung – in weitem Rahmen anpassen und erweitern.
- Das V-Modell XT trennt zwischen Auftraggeber- und Auftragnehmerprojekten. Es ist aber auch möglich (und natürlich sinnvoll), beide Rollen in einem Projekt zusammenzubringen.

Die Terminologie des V-Modells XT ist gewöhnungsbedürftig; das beginnt schon mit der »Empfehlung« zur Anwendung. (Zitat: »... *wird nunmehr die Anwendung des weiterentwickelten Entwicklungsstandards – V-Modell XT – in der vom IMKA*[2] *verabschiedeten Fassung für die Planung und Durchführung von IT-Verfahren in der Bundesverwaltung empfohlen.*« An anderer Stelle wird deutlich, dass es sich um eine bindende Vorschrift handelt.)

2 IMKA = Interministerieller Koordinierungsausschuss.

Viele bereits breit akzeptierte Begriffe wurden im V-Modell XT durch neue ersetzt (so wird der Begriff »Phase« nicht verwendet, stattdessen wird der Begriff »Projektabschnitt« eingeführt), andere Begriffe werden mit veränderter Semantik benutzt. So bedeutet zum Beispiel der Begriff »Projekttyp« nicht das, was man naiv dahinter vermutet.

10.3.3 Projekttypen

Das V-Modell XT soll für unterschiedliche Projektkonstellationen nutzbar sein. Dazu definiert es drei Projekttypen:

- Ein *Systementwicklungsprojekt (AG)* ist ein Entwicklungsprojekt, wie es sich aus der Sicht des Auftraggebers darstellt. Im Rahmen eines solchen Projekts erstellt der Auftraggeber eine Ausschreibung und wählt den Auftragnehmer aus den Angeboten aus. Ist das System fertig, nimmt es der Auftraggeber ab.
- Ein *Systementwicklungsprojekt (AN)* ist ein Entwicklungsprojekt aus der Perspektive des Auftragnehmers. Nachdem auf der Basis einer Ausschreibung ein Angebot erstellt wurde und ein Auftrag erteilt ist, entwickelt der Auftragnehmer das System gemäß festgelegten Projektdurchführungsstrategien und übergibt es an den Auftraggeber.
- Ein *Systementwicklungsprojekt (AG/AN)* liegt dann vor, wenn keine separaten Projekte auf der Auftraggeber- und Auftragnehmerseite notwendig sind. Dies kann beispielsweise der Fall sein, wenn Auftraggeber und Auftragnehmer aus derselben Organisation kommen.

Der Gegenstand von Projekten kann ein Hardware-System, ein Software-System, ein eingebettetes System, ein komplexes System, das Hardware-, Software- und eingebettete Anteile hat, oder eine Systemintegration sein. Für jeden Projekttyp bietet das V-Modell XT angepasste Varianten an. So gibt es beispielsweise für den Projekttyp »Systementwicklungsprojekt (AG/AN)« zwei Varianten, eine für die Neu- bzw. Weiterentwicklung (»AG-AN-Projekt mit Entwicklung, Weiterentwicklung oder Migration) und eine für ein Wartungsprojekt (»AG-AN-Projekt mit Wartung und Pflege«). Die Einordnung eines Projekts in diese Klassifikation bildet die Grundlage für das projektspezifische Tailoring des V-Modells XT (Abschnitt 10.3.6).

10.3.4 Elemente des Modells

Aktivitäten, Produkte und Rollen

Als *Produkte* werden die Ergebnisse und Zwischenergebnisse eines Projekts bezeichnet. Produkte werden in *Aktivitäten* erstellt oder bearbeitet; eine Aktivität kann in *Arbeitsschritte* gegliedert sein. Aktivitäten, die nicht weiter aufgeteilt sind, und Arbeitsschritte werden typischerweise en bloc durchgeführt.

Produkte können ebenfalls gegliedert sein, das V-Modell XT spricht dann von einem *Thema*. Inhaltlich zusammengehörende Produkte und die Aktivitäten, die sie erstellen, werden gruppiert und als *Disziplin* bezeichnet. Das V-Modell-XT definiert 19 Disziplinen und gruppiert diese in die Bereiche Management, Entwicklung, AG/AN-Schnittstelle.

Betrachten wir beispielsweise die Disziplin »Problem- und Änderungsmanagement« aus dem Bereich Entwicklung. Abbildung 10–3 zeigt diesen Prozess, die Produkte und Rollen dieser Disziplin in der Notation des V-Modells XT. Die V-Modell-Dokumentation (V-Modell XT Bund, 2019) beschreibt diese Disziplin (verkürzt) folgendermaßen: *Der Startpunkt ist eine Problemmeldung oder ein Änderungsantrag; beide können im Projektumfeld (1a) oder vom Projektteam erstellt werden (1b). Ein Änderungsverantwortlicher erfasst die Problemmeldung oder den Änderungsantrag in der Änderungsstatusliste und organisiert die Bewertung nach Relevanz, Dringlichkeit und sonstiger Kriterien (2). Die Änderungssteuerungsgruppe (Change Control Board, CCB) trifft auf dieser Basis eine Änderungsentscheidung (3), beispielsweise die Annahme des Änderungsantrags. Durch die Änderungsentscheidung können Konsequenzen wie Planänderungen entstehen.*

Die Aktivitäten, die durch die Rollen auszuführen sind, werden (leider) nicht explizit dargestellt. Sie werden im entsprechenden Vorgehensbaustein beschrieben.

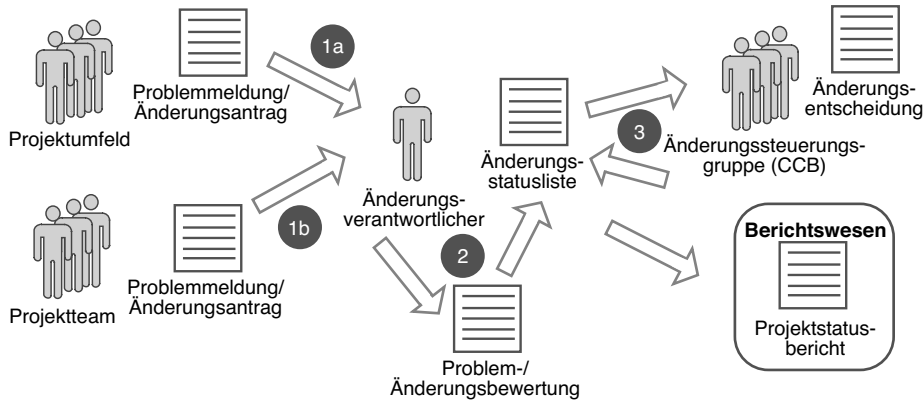

Abb. 10–3 *Die Disziplin »Problem- und Änderungsmanagement«*

Produkte können voneinander abhängen. So hängt das Produkt »Änderungsentscheidung« unter anderem von den Produkten »Problemmeldung/Änderungsantrag« und »Problem-/Änderungsbewertung« ab.

Rollen beschreiben zusammengehörende Aufgaben und Verantwortlichkeiten sowie die dazu notwendigen Fähigkeiten. Jedem Produkt ist genau eine dafür verantwortliche Rolle zugeordnet. Zusätzlich können Produkten weitere Rollen zugeordnet sein, die am Produkt mitwirken. So ist beispielsweise die Rolle

»Change Control Board« verantwortlich für das Produkt »Änderungsentscheidung«; die Rollen »KM-Verantwortlicher« und »QS-Verantwortlicher« wirken mit. Insgesamt sind im V-Modell XT mehr als 30 verschiedene Rollen definiert.

Vorgehensbausteine

Vorgehensbausteine sind die zentralen Einheiten des V-Modells XT. Ein Vorgehensbaustein umfasst eine Disziplin oder fasst Disziplinen zusammen, die inhaltlich zusammengehören und für eine Aufgabe, z. B. für das Projektmanagement, relevant sind. Das V-Modell XT definiert für alle Projekte vier obligatorische Vorgehensbausteine (der sogenannte V-Modell Kern):

1. Projektmanagement
2. Qualitätssicherung
3. Problem- und Änderungsmanagement
4. Konfigurationsmanagement

Je nach Ausprägung spezieller Projektmerkmale kommen im Tailoring (Abschnitt 10.3.6) weitere Vorgehensbausteine hinzu. Vorgehensbausteine bauen aufeinander auf; diese Abhängigkeiten sind im Tailoring zu berücksichtigen.

Als Beispiel betrachten wir den Vorgehensbaustein »Problem- und Änderungsmanagement«. Abbildung 10–4 zeigt die Produkte, Tätigkeiten und Rollen, die das V-Modell XT dazu vorsieht.

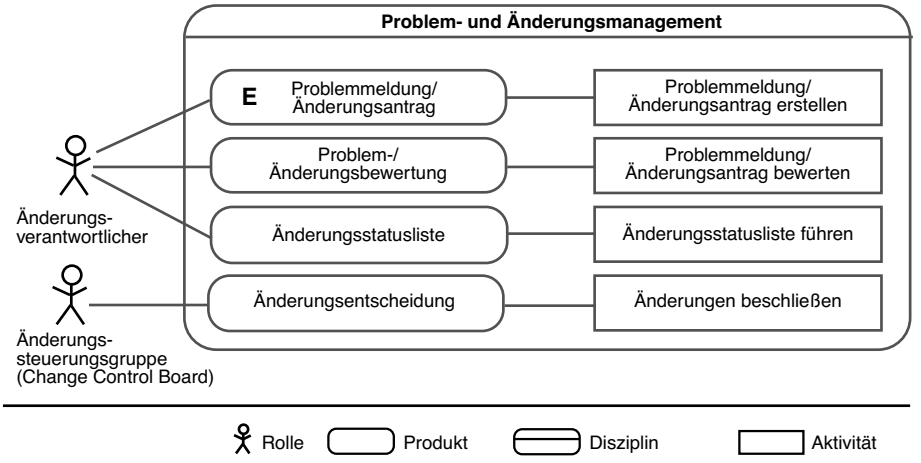

Abb. 10–4 *Vorgehensbaustein »Problem- und Änderungsmanagement«*

Wir sehen, dass die Rolle »Änderungsverantwortlicher« für die Produkte »Problemmeldung/Änderungsantrag«, »Problem-/Änderungsbewertung« und »Änderungsstatusliste« verantwortlich ist. Das mit einem »E« gekennzeichnete Produkt »Problemmeldung/Änderungsantrag« kann auch ein externes Produkt sein, dann wird es nicht im Projekt erstellt, sondern kommt von außen.

Entscheidungspunkte und Projektdurchführungsstrategien

Das V-Modell XT führt den Begriff *Entscheidungspunkt* ein. Entscheidungspunkte entsprechen etwa dem Konzept des Meilensteins (Abschnitt 10.2.1). Sie teilen das Projekt in Projektabschnitte (Phasen) ein. Ein Entscheidungspunkt ist ein Zeitpunkt im Projekt, an dem entschieden wird, ob der nächste Projektabschnitt begonnen werden kann. Dazu definiert jeder Entscheidungspunkt die Produkte, die am Entscheidungspunkt erstellt sein müssen und deren Bewertung die Grundlage der Entscheidung ist. Das V-Modell XT definiert 21 Entscheidungspunkte.

Eine *Projektdurchführungsstrategie* ordnet eine Menge von zusammengehörenden Entscheidungspunkten und gibt deren zeitliche Reihenfolge vor. Sie schafft den Rahmen, um ein Projekt geordnet und nachvollziehbar durchzuführen. Die gewählte Projektdurchführungsstrategie liefert die Grundlage für die Projektplanung. Der Projekttyp (inklusive der gewählten Variante) legt die Projektdurchführungsstrategie fest. Abbildung 10–5 zeigt am Beispiel des Projekttyps »AG-AN-Projekt mit Entwicklung, Weiterentwicklung oder Migration« die zeitliche Anordnung der dafür definierten Entscheidungspunkte.

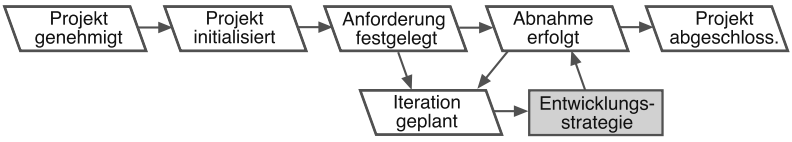

Abb. 10–5 *Entscheidungspunkte und Ablauf der Strategie für die Projekttypvariante: AG-AN-Projekt mit Entwicklung, Weiterentwicklung oder Migration*

Wie zu erkennen ist, werden Entwicklungsprojekte immer in Iterationen durchgeführt. Für jede Iteration muss entschieden werden, welche *Entwicklungsstrategie* angewendet werden soll. Eine Entwicklungsstrategie ordnet ebenfalls eine Menge von Entscheidungspunkten, die erreicht werden müssen.

Das V-Modell XT unterstützt die folgenden drei Entwicklungsstrategien:

1. Inkrementelle Entwicklung
2. Komponentenbasierte Entwicklung
3. Prototypische Entwicklung

Wir betrachten diese in Abschnitt 10.3.5 genauer.

Zusammenfassung

Abbildung 10–6 zeigt die beschriebenen zentralen Elemente des V-Modells XT und deren Zusammenhänge in der Notation der UML-Klassendiagramme.

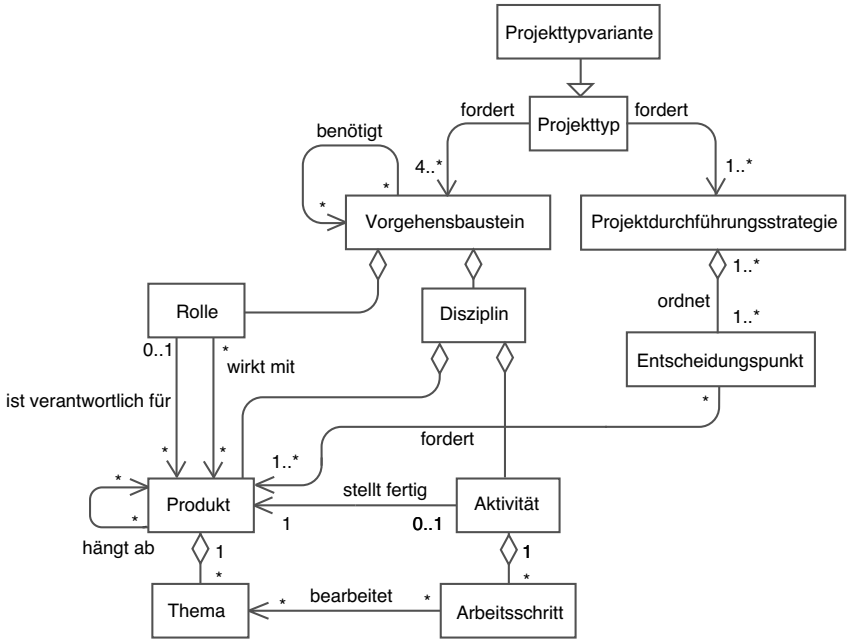

Abb. 10–6 *Struktur des V-Modells XT (vereinfacht)*

Die Struktur des V-Modells XT kann folgendermaßen (vereinfacht) beschrieben werden:

- Für jeden Projekttyp bzw. für jede Projekttypvariante sind Vorgehensbausteine und Projektdurchführungsstrategien festgelegt. Die vier Kern-Vorgehensbausteine sind obligatorisch, andere sind optional.
- Eine Projektdurchführungsstrategie gibt Entscheidungspunkte vor, die erreicht werden müssen. An jedem Entscheidungspunkt müssen definierte Produkte fertiggestellt sein.
- Ein Vorgehensbaustein fasst alle Rollen, Produkte und Aktivitäten (die Disziplinen) zusammen, die notwendig sind, um eine zentrale Projektaufgabe zu lösen. Vorgehensbausteine bauen aufeinander auf.
- Produkte und Aktivitäten sind in Disziplinen gruppiert und können gegliedert sein. Jedes Produkt wird durch genau eine Aktivität fertiggestellt. Produkte hängen voneinander ab.
- Rollen sind für Produkte verantwortlich und wirken an der Erstellung von Produkten mit.

10.3.5 Entwicklungsstrategien

Das V-Modell XT unterstützt drei Strategien, um ein System zu entwickeln (inkrementell, komponentenbasiert, prototypisch). Die zentralen Entscheidungspunkte dieser Strategien sind in Abbildung 10–7 in der typischen V-Anordnung dargestellt.

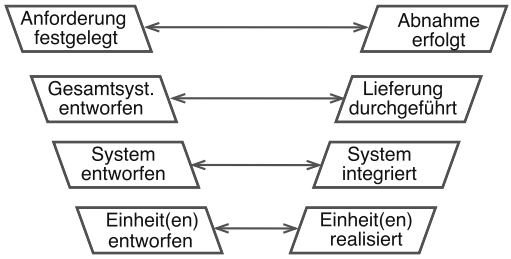

Abb. 10–7 *Zentrale Entscheidungspunkte der Systementwicklung*

Die Sequenz zwischen den Entscheidungspunkten und damit die Reihenfolge, in der diese erreicht werden müssen, variiert zwischen den unterschiedlichen Strategien (Abb. 10–8).

Bei der inkrementellen Entwicklung erkennen wir den typischen Zyklus: In jeder Iteration wird ein Inkrement entwickelt, ausgeliefert und abgenommen.

Die komponentenbasierte Entwicklung wird ebenfalls in Iterationen durchgeführt, die je ein Ergebnis abliefern. Zusätzlich können jedoch interne Iterationen durchlaufen werden. Diese Strategie sieht vor, dass auf Basis der Spezifikation und der vorhandenen Komponenten gleichzeitig top-down und bottom-up entworfen wird, um die Komponenten in die Architektur einzubinden.

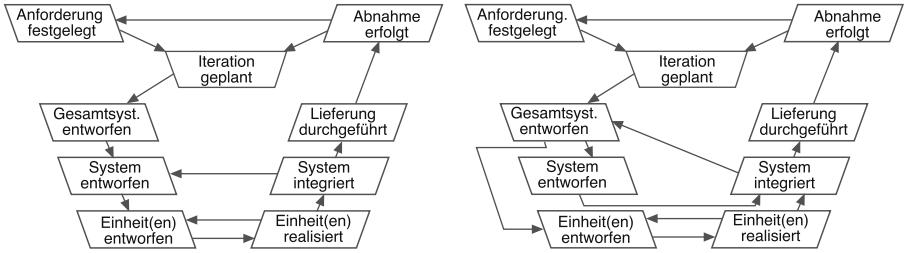

Abb. 10–8 *Ablauf der inkrementellen und der komponentenbasierten Entwicklung (vereinfacht)*

In Abschnitt 9.4 hatten wir Prototyping als eine wichtige Vorgehensweise eingeführt, um die Anforderungen zu klären und damit eine lange und teure Entwicklung zu vermeiden, die ein unbrauchbares System liefert.

Das V-Modell XT sieht dafür die Entwicklungsstrategie »prototypische Entwicklung« vor. Dabei wird das V praktisch aufgebrochen und zerfleddert: Die

Sequenz der zentralen Entscheidungspunkte ist hier (1) Anforderungen festgelegt, (2) Einheit(en) realisiert, (3) Einheit(en) entworfen, (4) System integriert, (5) System entworfen, (6) Gesamtsystem entworfen, (7) Lieferung durchgeführt, (8) Abnahme erfolgt (Abb. 10–9).

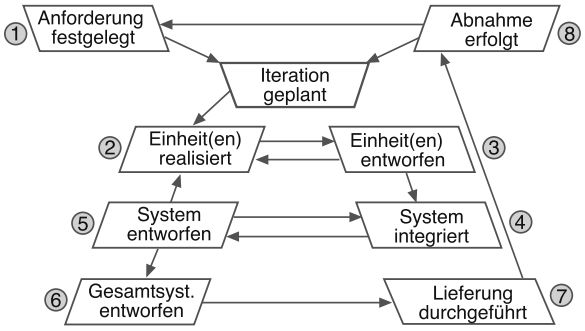

Abb. 10–9 *Ablauf der prototypischen Entwicklung (vereinfacht)*

Prototypen sollten immer dann entwickelt werden, wenn Risiken und Unklarheiten dadurch beseitigt werden können. Das V-Modell XT erlaubt aber (wie das folgende Zitat zeigt), dass die gesamte Systementwicklung auf Prototypen aufgebaut wird. Das erscheint uns zweifelhaft, ja, fahrlässig, da Prototypen nicht die Qualitätsanforderungen erfüllen (sollen), die an das Zielsystem gestellt werden.

> *Die prototypische Entwicklungsstrategie basiert auf der Erkenntnis, dass es oft nicht möglich ist, die Anforderungen an ein System vorab zu definieren. Außerdem stellt sie sicher, dass nichts spezifiziert wird, was sich als nicht realisierbar herausstellt. Somit wird diese Strategie insbesondere verwendet, wenn Realisierungsrisiken im Projekt vorhanden sind. Änderungen an den Anforderungen werden über das Problem- und Änderungsmanagement verwaltet. Typisch für diese Entwicklungsstrategie ist darüber hinaus die Präsenz des Auftraggebers auf der Auftragnehmerseite während der Entwicklung. Dadurch kann der Auftraggeber Änderungswünsche sehr direkt übermitteln. Der Auftragnehmer entwirft, realisiert und liefert das System dann ähnlich wie bei der Entwicklungsstrategie inkrementelle Entwicklung in einzelnen Stufen. Diese Stufen werden jede für sich vom Auftraggeber abgenommen. Für den Auftraggeber hat diese Vorgehensweise den Vorteil, dass er bereits frühzeitig in den Besitz eines lauffähigen Systems gelangt, das die wichtigsten Grundfunktionalitäten realisiert. Ferner ermöglicht sie eine frühzeitige Rückmeldung durch den Auftraggeber, die die Entwicklungsrisiken des Auftragnehmers minimiert.*
>
> V-Modell XT (o. J.), S. 292

Das liest sich überraschend naiv. Die Aussagen schlachten nicht nur ein paar heilige Kühe des Software Engineerings (allen voran die Spezifikation als wichtigstes Dokument), sondern gehen auch von einer Bilderbuchwelt aus, in der der Auftraggeber (beim V-Modell meist die öffentliche Hand!) im Projekt präsent ist und sowohl die fachliche als auch die juristische Kompetenz hat, um die Resultate kontinuierlich abzunehmen.

Insgesamt ist nicht nachvollziehbar, warum es sinnvoll sein soll, die zahlreichen Optionen der Entwickler gerade auf die drei Modelle inkrementell, komponentenbasiert und prototypisch zu reduzieren.

10.3.6 Tailoring

Ging man beim V-Modell 97 von der Gesamtmenge der Aktivitäten aus, die durch Streichungen eingeschränkt wurde, so beginnt man im V-Modell XT umgekehrt. Die Anpassung des V-Modells XT für ein konkretes Projekt geschieht dadurch, dass auf Basis des gewählten Projekttyps und der gewählten Projekttypvariante die Vorgehensbausteine bestimmt werden, die minimal erforderlich sind.

Weitere Vorgehensbausteine können hinzukommen, indem die Projektmerkmale bewertet werden. Dazu zählt neben dem Projektgegenstand (z. B. Hard- oder Software-System) beispielsweise der Einsatz von Fertigprodukten oder das Vorhandensein einer Bedienoberfläche. Einzelne Aktivitäten oder Produkte werden beim Tailoring nicht betrachtet, da sie immer Bestandteil eines Vorgehensbausteins sind.

Die Tabelle 10–1 zeigt die Zuordnung von Vorgehensbausteinen zu den Systementwicklungsprojekttypen AG, AN und AG/AN (X = verpflichtend, O = optional). Die obligatorischen Vorgehensbausteine, die jedes Projekt haben muss, sind nicht aufgeführt.

Für das V-Modell XT gibt es den sogenannten *Projektassistenten*, ein Werkzeug, das das Tailoring unterstützt. Der Projektassistent erhält als Eingaben den Projekttyp und die Werte für die Projektmerkmale, und bestimmt damit die Vorgehensbausteine. Mit dem Projektassistenten können nun die Projektdurchführungsstrategie (also die Reihenfolge der Entscheidungspunkte) geplant und die projektspezifische V-Modell-Dokumentation sowie die Vorlagen (Templates) für die zu erstellenden Produkte erzeugt werden.

Projekttyp	AG	AN	AG/AN
Projektcontrolling	X		X
Wirtschaftlichkeitsbetrachtung	X		X
Vertragsschluss (AG / externer Dienstleister)	X		O
Agile Software-Entwicklung	O	O	O
Lieferung und Abnahme	X		X
Anforderungsfestlegung			X
Evaluierung von Fertigprodukten	O	O	O
Systemerstellung			X
Software-Entwicklung	O	O	O
Benutzbarkeit und Ergonomie	O	O	O
Informationssicherheit und Datenschutz	O		O
Betriebsübergabe	O		O
Weiterentwicklung und Migration von Altsystemen	O	O	O
Multi-Projektmanagement	O		

Tab. 10–1 *Zuordnung von Vorgehensbausteinen zu Projekttypen*

10.3.7 Projektdurchführung

Durch das projektspezifische Tailoring werden die Vorgehensbausteine festgelegt und die Projektdurchführungsstrategie, die alle Entscheidungspunkte und deren Reihenfolge definiert.

Weil jeder Entscheidungspunkt die Produkte definiert, die fertiggestellt sein müssen, bevor der nächste Projektabschnitt beginnen kann, können nun die Tätigkeiten geplant werden, um die Produkte für die Entscheidungspunkte zu erstellen. Dabei sind natürlich die im V-Modell XT festgelegten Produktabhängigkeiten zu berücksichtigen. Die Entscheidungspunkte dienen als zentrale Bezugspunkte für die Projektfortschrittskontrolle. An jedem Entscheidungspunkt wird entschieden, ob das Projekt fortgesetzt werden kann (ggf., nachdem Nacharbeiten durchgeführt wurden) oder abgebrochen werden muss. Die getroffene Entscheidung wird in einer sogenannten *Projektfortschrittsentscheidung* dokumentiert.

Mit dem Projektassistenten kann auf Basis der gewählten Projektdurchführungsstrategien eine erste Planung erstellt werden, die die Termine aller Entscheidungspunkte festlegt. Das Werkzeug erzeugt mit dieser Information einen Vorschlag für die Planung der Aktivitäten, die aufgrund der Vorgehensbausteine und Entscheidungspunkte notwendig sind. Diese Information kann dann in ein Projektmanagement-Werkzeug übernommen und dort angepasst werden.

10.3.8 Bewertung des V-Modells

Für die Anwendung des V-Modells XT liegen einige Erfahrungsberichte vor (siehe z.B. Kuhrmann, Lange, Schnackenburg, 2011), bei denen der Einsatz dieses Modells untersucht wurde. Folgende Vorteile sehen wir in diesem Prozessmodell:

- Dadurch, dass das Modell im Kern vier Vorgehensbausteine definiert, die Bestandteil jedes Projekts sind, erhalten diese im Bewusstsein des Managements und der Entwickler denselben Stellenwert wie sogenannte konstruktive Tätigkeiten (also Spezifizieren oder Codieren).
- Das Modell ist öffentlich zugänglich und kann ohne Lizenzkosten benutzt werden.
- Das Modell ist generisch und bietet Unterstützung, um es unternehmens- und projektspezifisch anzupassen. Das V-Modell XT ist im Sinne eines Baukastens immer ein sinnvoller Ausgangspunkt, wenn ein unternehmensspezifisches Prozessmodell erstellt werden soll. Es senkt das Risiko, dass Wichtiges vergessen wird.

Als problematisch sehen wir die folgenden Aspekte:

- Mit dem V-Modell wird angestrebt, die Tätigkeiten, Arbeitsergebnisse und Abläufe vollständig festzulegen. Daraus ergibt sich ein großer Umfang, erkennbar an der enormen Menge von Produkten, Aktivitäten und Rollen, die das Modell definiert. Damit scheint es für kleinere und mittlere Projekte nicht ohne Weiteres geeignet zu sein. Notwendige Anpassungen werden zwar durch das Modell unterstützt, die Erfahrung zeigt aber, dass der Anpassungsprozess viel Zeit und Aufwand erfordert. Die (nachvollziehbare) Tendenz, lieber zu viel als zu wenig zu fordern, hat zur Folge, dass auch nach dem Tailoring noch ein sehr umfangreiches Gerüst übrig bleibt.
- Wird das V-Modell ohne erhebliche Anpassungen verwendet (das kann nur bei sehr großen Projekten sinnvoll sein), dann ist es aufgrund seiner Größe schwerfällig in der Anwendung. Die Menge der zu erstellenden Produkte (Dokumente) ist auch nach einem Tailoring für ein kleines Entwicklungsprojekt erheblich; die Wirtschaftlichkeit vieler Dokumente muss bezweifelt werden.
- Die Beschreibungen und die in den Projektdurchführungsstrategien modellierten Abläufe der Entwicklungsstrategien sind zum Teil mehr als diskutabel (z. B. die prototypische Systementwicklung).

Das V-Modell XT wird durch den Verein zur Weiterentwicklung des V-Modell XT e. V. stetig weiterentwickelt. Aktuell (2022) liegt es in der Version 2.3 vor.

10.4 Der Unified Process

Die objektorientierte Programmierung verbreitete sich ab 1980 mit der Sprache SMALLTALK. Nur wenige Jahre später standen weitere Programmiersprachen zur Verfügung, die nicht wie SMALLTALK genuin objektorientiert sind, sondern durch Erweiterung älterer Sprachen um die Konzepte der Objektorientierung geschaffen wurden. Insbesondere C++ spielte wegen der Kompatibilität mit der bereits weitverbreiteten Sprache C eine wichtige Rolle.

Die objektorientierte Programmierung wurde gegen Ende der Achtzigerjahre rasch populär, denn sie hatte neben dem Reiz des Neuen offensichtliche Vorteile: schnelle Entwicklung komplexer Systeme, vor allem ansprechender Bedienoberflächen, leichte (feingranulare) Wiederverwendung des Codes. Bald zeigten sich aber auch neue Probleme, am schnellsten bei den SMALLTALK-Projekten: Für die neuen Konzepte taugten die alten, auf rein imperative Sprachen zugeschnittenen Methoden für Analyse und Entwurf, Qualitätssicherung und Test nicht. Auch die konventionellen Prozessmodelle erwiesen sich als wenig geeignet für die objektorientierte Software-Entwicklung.

Ivar Jacobson entwickelte und publizierte zwischen 1987 und 1992 einen Ansatz, der zunächst *Objectory* (Object Factory for Software Development) genannt wurde. Hierbei handelte es sich um eine auf die Objektorientierung abgestimmte Entwurfsmethode. Jacobson erkannte aber, dass auch neue Prozesskomponenten benötigt wurden. So entstand der *Objectory-Prozess*. Als neues Element führte Jacobson den sogenannten *Anwendungsfall* (*Use Case*) ein und machte ihn zur Grundlage des Objectory-Prozesses. In Jacobson et al. (1992) werden die Konzepte und die Grundelemente des Objectory-Prozesses detailliert beschrieben.

1995 schlossen sich die Firmen Objectory AB von Ivar Jacobson und die Rational Software Corporation zusammen. Die Konzepte des Objectory-Prozesses in der Version 3.8 und die bei Rational entwickelten Ansätze zur Software-Entwicklung, die maßgeblich durch G. Booch, J. Rumbaugh und P. Kruchten geprägt wurden, verschmolzen. Sie wurden zuerst als *Rational Objectory Process*, ab 1998 als *Rational Unified Process* (RUP) bezeichnet.

1999 wurde der »Unified Software Development Process« (kurz: Unified Process) veröffentlicht (Jacobson, Booch, Rumbaugh, 1999), der von den Spezifika des RUP abstrahiert. Der RUP ist eine konkrete Ausprägung des Unified Process. Nachfolgend stellen wir die zentralen Konzepte und Elemente des Unified Process vor, anschließend betrachten wir den RUP.

10.4.1 Eigenschaften des Unified Process

Der Unified Process (UP) soll die Vorteile von Phasenmodellen und nichtlinearen Prozessmodellen vereinen: Phasen erleichtern die Planung und das Management von Entwicklungsprojekten, Iterationen und inkrementelles Entwickeln helfen, Risiken frühzeitig zu erkennen und zu beseitigen. Der Unified Process kann durch die folgenden Merkmale charakterisiert werden:

- *Der UP ist ein Phasenmodell.*
  Er gibt vier Phasen vor, an deren Ende jeweils ein Meilenstein passiert werden muss. In jeder einzelnen Phase werden alle definierten Arbeitsabläufe durchgeführt, jedoch in unterschiedlicher Intensität. Der Durchlauf durch die vier vorgegebenen Phasen wird als Zyklus bezeichnet. Das Ergebnis eines Zyklus ist immer ein Release, das entweder intern oder an den Kunden ausgeliefert wird.
- *Der UP ist iterativ, ein Release entsteht inkrementell.*
  Während Phasen die Managementsicht eines Projekts bilden, läuft die technische Realisierung in Iterationen ab. Jede Phase kann mehrere Iterationen enthalten. In jeder Iteration werden alle im Prozess definierten Arbeitsabläufe durchgeführt und die zu erstellenden Arbeitsergebnisse inkrementell weiterentwickelt. Das Ergebnis einer Iteration ist immer eine ausführbare Systemkonfiguration. Die Schwerpunkte der Iterationen sind unterschiedlich. In den Iterationen der frühen Phasen entstehen oft Prototypen. Der Test (hier im Sinne von Prüfung) verteilt sich auf die Iterationen aller Phasen, da in jeder Iteration geprüft werden muss (siehe dazu auch Abb. 10–11).
- *Die Produktentwicklung kann inkrementell erfolgen.*
  Soll ein Produkt in Ausbaustufen entwickelt werden, dann werden mehrere Zyklen durchlaufen, in denen das Produkt inkrementell entwickelt wird. Am Ende des letzten Zyklus ist das Produkt fertig.
- *Der UP basiert auf der Verwendung von Anwendungsfällen (Use Cases).*
  Die Identifikation und Modellierung von Anwendungsfällen ist ein zentraler Bestandteil des Ansatzes. Anwendungsfälle dienen dazu, die funktionalen Anforderungen zu beschreiben. Die dabei erstellten Modelle bilden die Grundlage für viele andere Tätigkeiten (z. B. Entwerfen der Architektur, Auswahl von Testfällen, Planung von Ausbaustufen) und deren Ergebnisse.
- *Der UP ist architekturzentriert.*
  Die Architektur bildet neben Anwendungsfällen die Basis für die Entwicklung. Die Architektur wird frühzeitig, parallel zu den Anwendungsfällen, entworfen und stetig weiterentwickelt.

10.4.2 Die Struktur des Unified Process

Rollen, Aktivitäten und Artefakte

Der UP benutzt nicht, wie sonst üblich, den Begriff »*Rolle*«, um zusammengehörende Aufgaben, Verantwortlichkeiten und Fähigkeiten zu bezeichnen, sondern führt dazu den Begriff »*Worker*« ein. Dies hat zwei Gründe: Zum einen ist der Begriff »Rolle« im Kontext von UML definiert, hier wollte man keine Verwirrung stiften, zum anderen nehmen »Worker« in den vorgesehenen Arbeitsabläufen unterschiedliche Rollen ein. Wir nutzen in diesem Abschnitt jedoch weiterhin den Begriff »Rolle«.

Der UP versteht unter einer *Aktivität* eine zusammenhängende Tätigkeit, die von einer Rolle in einem Arbeitsablauf ausgeführt wird. Eine Aktivität liefert ein definiertes Ergebnis (eine Menge von Artefakten, die erstellt oder modifiziert werden) und hat eine Menge von Artefakten als Eingabe. Die Rolle »System Analyst« führt beispielsweise die Aktivität »Find Actors and Use Cases« durch.

Mit dem Begriff »*Artefakt*« bezeichnet der UP jede explizit formulierte Information; dabei wird zwischen Dokumenten und Modellen (z. B. einem Use-Case-Modell) unterschieden. Artefakte werden im Rahmen von Aktivitäten durch Rollen erstellt, benutzt oder modifiziert. So nutzt die Rolle »System Analyst« in der Tätigkeit »Find Actors and Use Cases« z. B. das Artefakt »Business Model« und produziert oder erweitert die Artefakte »Use Case Model« und »Glossary«. Arbeitsergebnisse unterliegen der Konfigurations- und Änderungsverwaltung.

Der UP beschreibt lediglich die zentralen Rollen, Aktivitäten und Artefakte. Diese können (und müssen) für ein konkretes Prozessmodell angepasst und erweitert werden.

Phasen

Als Phasenmodell definiert der UP die folgenden vier Phasen:

- *Inception*
  Das Ziel dieser Phase ist, die Produktidee und die Anforderungen an das zukünftige System so weit zu verstehen, dass auf der Basis von wirtschaftlichen Betrachtungen ein Entwicklungsprojekt gestartet werden kann. Dazu werden die zentralen Anwendungsfälle identifiziert und modelliert, die Risikosituation des Projekts wird bewertet, erste Fassungen der Architektur und des Projektplans werden erstellt. Wenn notwendig, werden erste Prototypen gebaut, um risikoreiche Aspekte zu untersuchen.
- *Elaboration*
  In dieser Phase werden alle noch fehlenden Anforderungen identifiziert. Ferner werden die grundlegenden Architekturentscheidungen getroffen und in der Systemarchitektur formuliert. Der Bau eines ersten Prototyps, der den Architekturkern und dessen Funktionalitäten zeigt, findet in dieser Phase

statt. Aus Managementsicht gilt es, die größten Risiken zu identifizieren, zu bewerten und geeignete Gegenmaßnahmen zu planen sowie die Planung für die folgenden Phasen zu erstellen.

- *Construction*
  In dieser Phase wird das System auf der Basis der vorhandenen Systemarchitektur implementiert, integriert und getestet. Noch unvollständige Dokumente, wie die Benutzerdokumentation, werden fertiggestellt. Am Ende dieser Phase steht ein einsetzbares System in der Qualität einer Beta-Version zur Verfügung.
- *Transition*
  Das bereits einsetzbare System wird in dieser Phase so lange verbessert, bis sein Zustand stabil ist. Dabei spielen die Erfahrungen und die Rückmeldungen der Anwender eine zentrale Rolle. Das übergeordnete Ziel dieser Phase ist, das System mit allen dazugehörenden Dokumenten so weit zu vervollständigen, dass es Produktqualität erreicht, ausgeliefert und ohne Einschränkung eingesetzt werden kann. Diese Phase endet, wenn der Kunde mit dem gelieferten Ergebnis zufrieden ist. Das Projekt wird abgeschlossen und die Ergebnisse werden archiviert. Gegebenenfalls muss entschieden werden, ob ein weiterer Zyklus, d. h. eine weitere Ausbaustufe, geplant und realisiert werden soll.

Arbeitsabläufe

Ein Arbeitsablauf (Workflow) fasst zusammengehörende Aktivitäten, die ausführenden Rollen sowie die benötigten und produzierten Artefakte zusammen. Der UP definiert fünf Kern-Arbeitsabläufe (Core Workflows): »Requirements«, »Analysis«, »Design«, »Implementation« und »Test«. Diese werden in allen Projektphasen ausgeführt. Abbildung 10–10 zeigt in der Notation des UP die Rollen und Aktivitäten des Workflows »Requirements«.

Iterationen

Wie bereits erwähnt, ist der UP ein iterativer Prozess. Jede Phase kann in mehrere Iterationen aufgeteilt werden. In einer Iteration werden die für die Phase relevanten Artefakte inkrementell weiterentwickelt. Jede Iteration wird durch einen (internen) Meilenstein abgeschlossen. Da alle Kern-Arbeitsabläufe in jeder Phase durchgeführt werden, sind sie immer auch Teil jeder Iteration. Der UP nennt diese dann »Iteration Workflows«. Abbildung 10–11 zeigt die Zusammenhänge zwischen Phasen, Arbeitsabläufen und Iterationen.

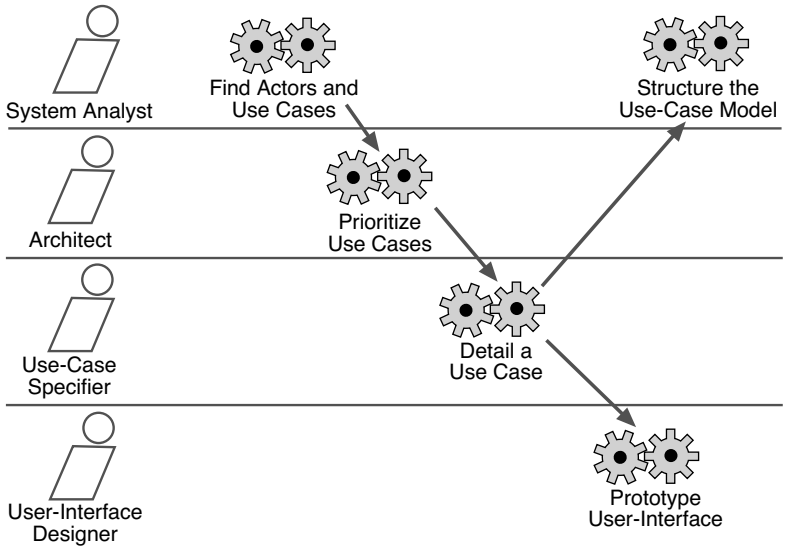

Abb. 10–10 *Der Workflow »Requirements« (Jacobson, Booch, Rumbaugh, 1999, S. 143)*

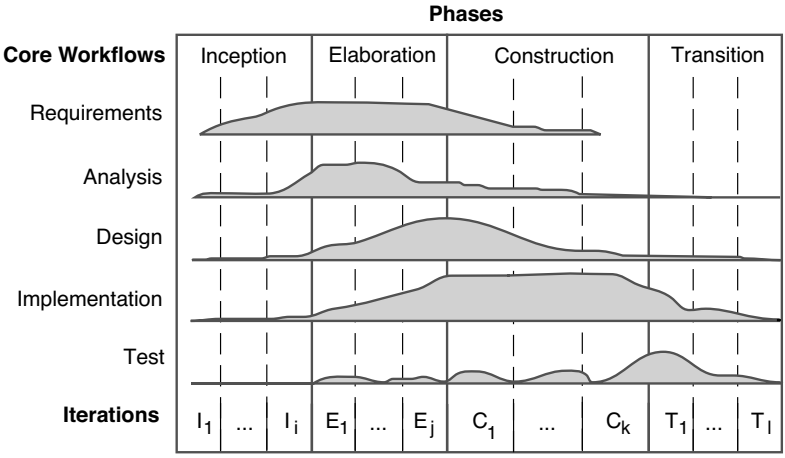

Abb. 10–11 *Phasen, Iterationen und Arbeitsabläufe (Jacobson, Booch, Rumbaugh, 1999)*

In jeder Phase werden die Arbeitsabläufe in unterschiedlicher Intensität durchgeführt. Während in den ersten Phasen die Tätigkeiten zur Anforderungsermittlung und zum Systementwurf dominieren, überwiegen in den späteren Phasen Codier- und Testaktivitäten.

10.4.3 Der Rational Unified Process

Der Rational Unified Process (RUP) prägt den generischen Unified Process zu einem konkreten Prozessmodell aus. Dabei bleiben natürlich die Grundideen und Konzepte des UP erhalten. Die zentralen Elemente des UP – also Rollen, Aktivitäten und Artefakte – werden jedoch erweitert und konkretisiert. Der RUP wird von der Firma IBM/Rational vermarktet; seine Anwendung erfordert eine Lizenz. Er wurde bereits in mehreren Entwicklungsstadien publiziert und wird kontinuierlich weiterentwickelt. Der RUP ist gut dokumentiert; mit der Lizenz erhält man eine vollständige HTML-basierte Dokumentation. Aber auch die öffentlich zugängliche Literatur zum RUP ist umfangreich, eine gute Einführung gibt beispielsweise Kruchten (2003).

Leider verändert der RUP die Terminologie des UP an einigen Stellen, so wird der Begriff »Workflow« durch »Discipline« ersetzt. Als Workflow bezeichnet RUP die Anordnung und den Ablauf der Aktivitäten einer Disziplin. Wir benutzen weiterhin den Begriff »Arbeitsablauf«; ein Arbeitsablauf wird durch einen Workflow beschrieben. Neben neuen Rollen, Aktivitäten und Artefakten modifiziert und erweitert der RUP die fünf Kern-Arbeitsabläufe des UP. Insgesamt gibt es die in der folgenden Tabelle aufgeführten neun Arbeitsabläufe.

Arbeitsablauf	Zweck des Arbeitsablaufs
Business Modelling	Die Strukturen und die Abläufe in der Auftraggeber-Organisation verstehen, eine gemeinsame Sprache etablieren
Requirements	Die Anforderungen an das zu entwickelnde System erheben und bearbeiten
Analysis and Design	Aus den Anforderungen ein Entwurfsmodell und die Systemarchitektur gewinnen
Implementation	Die Teile der Architektur codieren und integrieren
Test	Prüfungen verschiedener Arten durchführen, prüfen, ob das entwickelte System die Anforderungen erfüllt
Deployment	Das System zusammenstellen, geordnet an den Auftraggeber übergeben und dort in Betrieb nehmen
Configuration and Change Management	Die erstellten Arbeitsergebnisse systematisch verwalten und Änderungen daran geordnet durchführen
Project Management	Ein Projekt und seine Iterationen planen und kontrollieren (einschließlich Risikomanagement)
Environment	Das Entwicklungsprojekt unterstützen (z. B. durch die Auswahl von Entwicklungswerkzeugen, die Administration von Rechnern oder das Erstellen von Backups)

Tab. 10–2 *Arbeitsabläufe (Disciplines) des RUP*

Jeder Arbeitsablauf wird auf der oberen Ebene durch einen Workflow in Form eines UML-Aktivitätsdiagramms beschrieben. Damit wird versucht, die Reihen-

folge und die Abhängigkeiten der einzelnen Aktivitäten eines Workflows zweidimensional zu visualisieren. Jeder Workflow hat einen Ein- und einen Austrittspunkt. Neben Aktivitätssymbolen kann ein solches Diagramm auch Entscheidungsaktivitäten enthalten, die mit Bedingungen versehen sind. An diesen Punkten verzweigt sich der Arbeitsablauf. Außerdem kann durch sogenannte Synchronisationsbalken Parallelität modelliert werden. Wenn die auf einen Synchronisationsbalken führenden Aktivitäten abgeschlossen sind, können alle Aktivitäten, die vom Synchronisationsbalken ausgehen, parallel ausgeführt werden. Abbildung 10–12 zeigt den Workflow des Arbeitsablaufs »Requirements«. Man erkennt, dass der Detaillierungsgrad im Vergleich zum generischen Ablauf im UP (Abb. 10–10 auf S. 214) gestiegen ist.

Die Aktivitätssymbole fassen in der Regel eine Gruppe von Aktivitäten zusammen. Auf der darunterliegenden Ebene, die »Workflow Detail« genannt wird, werden die Gruppen ebenfalls in einer grafischen Notation aufgelöst. Abbildung 10–13 zeigt exemplarisch die im Workflow »Requirements« (Abb. 10–12) definierte Aktivität »Analyze the Problem«.

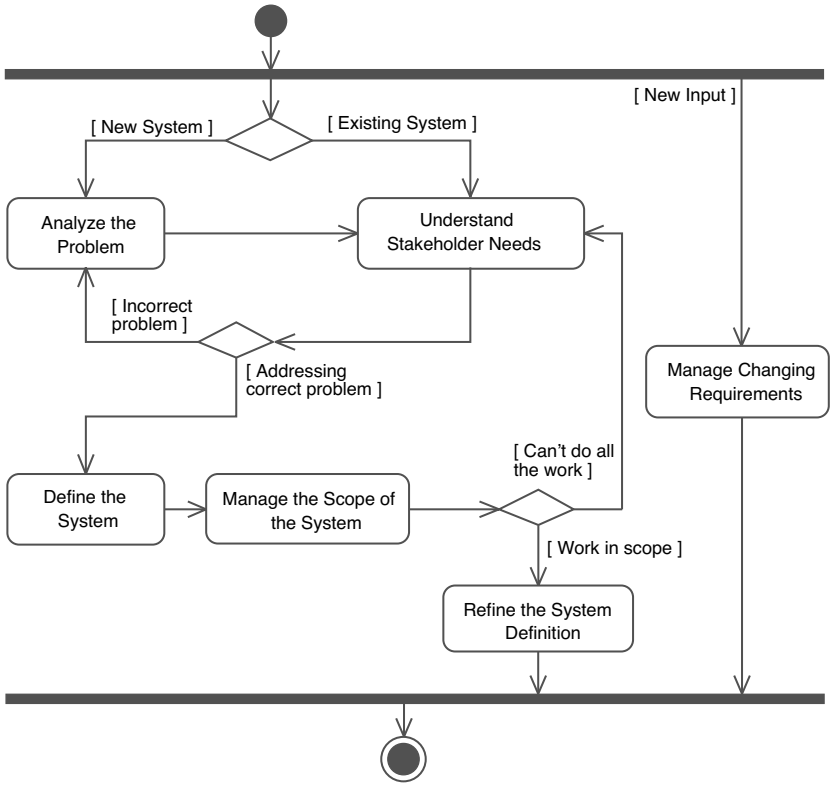

Abb. 10–12 *Workflow des Arbeitsablaufs »Requirements«, © IBM Corp.*

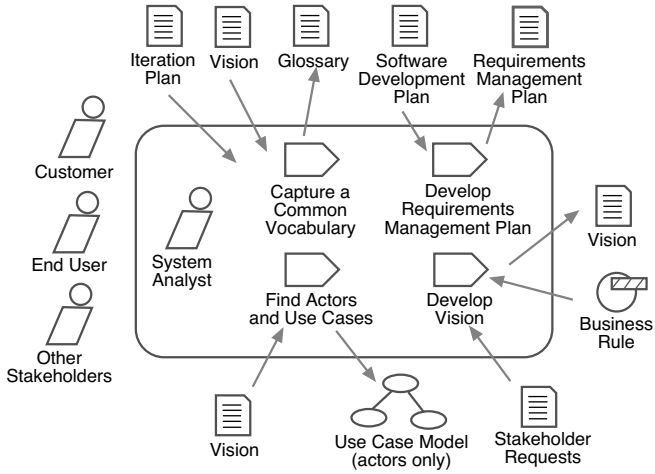

Abb. 10–13 *Darstellung der Aktivitätsgruppe »Analyze the Problem«, © IBM Corp.*

Auf dieser Ebene werden die Beziehungen zwischen Aktivitäten, Rollen und den wichtigsten Artefakten sichtbar. So führt die Rolle »System Analyst« die vier gezeigten Aktivitäten durch. Die am Rand platzierten Rollen sind an den gezeigten Aktivitäten beteiligt, indem sie notwendige Informationen beisteuern. Die Aktivitäten produzieren oder verändern Artefakte; diese sind ebenfalls dargestellt.

Der RUP beschreibt jede atomare Aktivität detailliert durch einen Text, der die auszuführenden Schritte angibt. Für die Artefakte wird die Gliederung vorgegeben, der Inhalt wird umrissen.

10.4.4 Bewertung des Rational Unified Process

Damit der Rational Unified Process eingesetzt werden kann, muss die Entwicklungsorganisation folgende Voraussetzungen erfüllen:

- *Ausgezeichnete Konfigurations- und Änderungsverwaltung*
  Die iterative und inkrementelle Entwicklung führt dazu, dass sich die Arbeitsergebnisse in jeder Iteration verändern. Damit dies nicht ins Chaos führt, muss die Konfigurations- und Änderungsverwaltung ausgereift und mit geeigneten Werkzeugen ausgerüstet sein.
- *Gute Projektmanagement-Fertigkeiten*
  Auch wenn die definierten Phasen auf der oberen Ebene die Struktur eines Projekts klar vorgeben, bleibt bei diesem Prozessmodell doch die Schwierigkeit bestehen, die Anzahl und die Dauer der einzelnen Iterationen zu planen, die in den Phasen durchgeführt werden müssen. Dies ist nicht einfach und erfordert entsprechende Erfahrung.

- *Kenntnis objektorientierter Konzepte und Notationen*
  Der RUP ist für die objektorientierte Systementwicklung konzipiert. Daher muss das dazu notwendige Know-how bereits vorhanden sein, es entsteht nicht durch Einführung des Prozesses. Insbesondere muss das für diesen Prozess essenzielle Konzept der Anwendungsfälle verstanden sein und beherrscht werden.

Das Prozessmodell hat unseres Erachtens seine Stärken in den folgenden Bereichen:

- *Gute Darstellung des Prozesses*
  Da der RUP in einer HTML-Fassung zur Verfügung steht, können seine Anwender einfach durch den Prozess navigieren und Informationen finden. Die Prozessbeschreibung ist jedem Mitarbeiter am Arbeitsplatz direkt zugänglich und erhöht damit die Bereitschaft, darin zu lesen und in Zweifelsfällen in der Prozessbeschreibung nachzuschauen.
- *Hoher Detaillierungsgrad*
  Der Prozess ist sehr detailliert beschrieben. So werden zu den Aktivitäten nicht nur die einzelnen Arbeitsschritte aufgelistet, sondern auch Arbeitsanleitungen mitgeliefert. Die Arbeitsergebnisse werden inhaltlich beschrieben, und zu jedem Arbeitsergebnis gibt es ein Musterdokument, das die Struktur des zu erstellenden Dokuments aufweist. Dieser feine Detaillierungsgrad ist ein Vorteil, weil die Prozessbeschreibung meist ausreichend Information enthält, damit der Ablauf und die definierten Tätigkeiten zu verstehen sind.

Daneben sehen wir die folgenden Problembereiche:

- *Schwierige Anpassung*
  Der RUP muss in der Regel an die speziellen Gegebenheiten einer Entwicklungsorganisation angepasst werden. Dies betrifft mindestens die Tätigkeiten der notwendigen Rollen und die geforderten Arbeitsergebnisse. Da der Prozess sehr detailliert beschrieben ist und dadurch die Prozesselemente sehr stark miteinander vernetzt sind, ist die Anpassung nicht einfach und nur mit erheblichem Aufwand möglich. Der RUP selbst gibt nur wenig Hilfestellung für das projektspezifische Tailoring.
- *Instabile Prozessdefinition*
  Der RUP ist ein Produkt und wird entsprechend vermarktet und weiterentwickelt. Dies führt unweigerlich dazu, dass immer wieder geänderte Fassungen des Prozesses entstehen. Damit wird die Definition des eigenen organisationsspezifischen Entwicklungsprozesses abhängig von der Weiterentwicklung dieses Produkts. Werden in kurzer Folge neue Fassungen des Prozesses erstellt, dann muss der eigene Prozess laufend angepasst werden. Dies ist schwierig, teuer und nicht immer gewünscht. Die Alternative ist, sich von der Weiterentwicklung des RUP abzukoppeln. Das kann aber andere Nachteile haben

(Kompatibilität mit Werkzeugen, erschwerter Austausch mit Leuten, die an die neue Fassung gewöhnt sind).

- *Der Prozess täuscht vor, dass die Software-Entwicklung algorithmisch durchgeführt werden kann*
  Die mithilfe von Aktivitätsdiagrammen modellierten Arbeitsabläufe können den Eindruck erwecken, dass eine Durchführung der Tätigkeiten in der vorgeschriebenen Reihenfolge garantiert zu einem guten Ergebnis führt. Diese Sicherheit besteht tatsächlich nicht. Es genügt nicht, nur formal nach der Prozessvorgabe zu arbeiten. Viel wichtiger ist, dass die in den Aktivitäten erzielten Arbeitsergebnisse brauchbar und von ausreichender Qualität sind.

Offenkundig werden durch den Rational Unified Process viele Aspekte der Entwicklung sinnvoll identifiziert und geregelt. Besonders wichtig ist, dass man die Vorteile der phasenorientierten Entwicklung erhält, ohne auf die iterative Entwicklung verzichten zu müssen.

Um den Rational Unified Process einfacher anpassen zu können, wurde von IBM der sogenannte RATIONAL METHOD COMPOSER entwickelt. Dieses Werkzeug enthält die komplette Dokumentation des Rational Unified Process.

10.5 Cleanroom Development

Unter den in diesem Kapitel vorgestellten Prozessmodellen sticht eines heraus, das in verschiedener Hinsicht ganz anders ist als die übrigen: das Cleanroom Development. Es ist etwa 1987 entstanden, gelegentlich, aber nicht sehr oft, angewendet worden und nach über 30 Jahren praktisch vergessen. In seiner Reinform ist es nicht praxistauglich, vermutlich, weil es sehr hohe Anforderungen an alle beteiligten Personen stellt. Wir behandeln es hier trotzdem, weil darin sehr sinnvolle Ideen stecken, die jedem Software-Ingenieur bekannt sein sollten, sodass er sie dort, wo sie positive Impulse geben, in seine Projekte einbringen kann. »Cleanroom« könnte man auch übersetzen mit »Quality first«.

10.5.1 Der Hintergrund des Cleanroom Development Process

In der traditionellen industriellen Fertigung entstehen unvermeidlich auch fehlerhafte Produkte. Darum werden die Resultate geprüft. Wo ein defektes Produkt entdeckt wird, kann man es wegwerfen oder, soweit dies möglich und rentabel ist, reparieren. In jedem Fall versucht man sicherzustellen, dass nur ein kleiner Teil der Produkte betroffen ist. Beispielsweise sind einige der produzierten Leuchtmittel Ausschuss, und bei Autos sind nicht selten Nachbesserungen nötig.

Bei der Fertigung hochintegrierter Schaltungen (Prozessor- und Speicherchips) ließ sich dieser Ansatz nicht ohne Weiteres übernehmen: Nur ein kleiner Bruchteil der Chips war in Ordnung; die übrigen konnten aber nicht repariert

werden, denn die Strukturen lassen sich selbst im Mikroskop nicht mehr betrachten, geschweige denn reparieren. Die Ausbeute war also extrem gering.

Ein beträchtlicher Teil der Defekte entsteht durch Verunreinigungen. Jedes noch so kleine Staubkorn, das sich auf den Wafer setzt, bedeutet einen Defekt. Darum wurden die Räume, in denen Chips produziert werden, mit großem Aufwand vor Staub geschützt. Durch Filter, Schleusentüren usw. wurde die Wahrscheinlichkeit einer Verunreinigung drastisch gesenkt und die Ausbeute entsprechend erhöht. Im *Reinstraum* (*Cleanroom*) wird also dafür gesorgt, dass nicht Fehler gemacht und dann beseitigt werden, sondern sie von Beginn an vermieden werden. Der erforderliche Aufwand ist groß, aber die Einsparung ist größer.

Software lässt sich – anders als die Chips – reparieren. Aber der Aufwand dafür ist sehr hoch, und der Erfolg ist höchst ungewiss, zumal bei einer Reparatur mit beträchtlicher Wahrscheinlichkeit neue Fehler entstehen. Forscher in der IBM Federal Systems Division hatten darum die Idee, das Konzept des Cleanrooms zu übernehmen, also Software so zu entwickeln, dass sie *gleich richtig* ist. Dieses Vorgehen führte zum *Cleanroom Development Process* (CDP).

Der CDP ist in der Literatur gut dokumentiert (beispielsweise in Mills, Dyer, Linger, 1987; Baker, Basili, Selby, 1987; Dyer, 1992; Linger, Trammell, 1996; Linger, 1993; Poore, Trammell, 1996; Prowell et al., 1999). Trotzdem ist nicht klar, welche Elemente genau den CDP definieren. Viele Nachahmer verwenden die Bezeichnung, obwohl sie kaum mehr als eine normale Qualitätssicherung zu bieten haben. Die Urheber selbst bieten drei verschiedene Stufen der Cleanroom-Implementierung an (Hausler, Linger, Trammell, 1994), nämlich die Basisimplementierung (introductory), die vollständige (full) und die fortgeschrittene Implementierung (advanced). Am klarsten sind die Aussagen zur mittleren Stufe: Daran orientieren sich die folgenden Aussagen. Auf der ersten Stufe sind vor allem die Grundideen verwirklicht, nicht die speziellen Techniken. Die dritte Stufe erlaubt und fordert die Adaption strenger, formaler Verfahren, die über die zweite Stufe hinausgehen.

10.5.2 Merkmale und Eigenschaften des Cleanroom Development Process

Die Situation vor dem Cleanroom-Ansatz

Nach den Erfahrungen und Untersuchungen in den Siebzigerjahren können folgende Feststellungen als gesichert gelten:

- Projekte begrenzter Größe und Laufzeit haben weit bessere Chancen als langlaufende Großprojekte.
- Eine gründliche Analyse und Spezifikation (siehe Kap. 15) ist die wichtigste technische Voraussetzung für den Erfolg.
- Inspektionen sind geeignet, Fehler frühzeitig, womöglich vor der Implementierung, zu entdecken. Das gilt sowohl für die Reviews (siehe Abschnitt 13.5)

als auch – mit den entsprechenden Einschränkungen – für das gründliche Lesen durch den Verfasser selbst (vgl. Abschnitt 6.5). Aber auch beim Code ist die Inspektion wirksamer als der Test.

- Hinter syntaktischen Fehlern verbergen sich oft semantische Probleme. Ein Bezeichner, zu dem es keine Deklaration gibt, kann durch einen Tippfehler entstanden sein, aber ebenso durch das Versäumnis, eine weitere Variable einzuführen. Die Unterscheidung zwischen syntaktischen und semantischen Fehlern ist oft spekulativ.
- Nachbesserungen sind, wie oben festgestellt wurde, mit hohen Kosten und beträchtlichen Risiken behaftet.
- Testdaten, die einzeln »von Hand geschnitzt« werden, lassen sich wegen des großen Aufwands nur in geringer Zahl bereitstellen, sie liefern keine Aussagen über die Zuverlässigkeit des Systems.

Cleanroom-Konzepte

Als Reaktion auf die durchweg negativen Erfahrungen mit den Software-Engineering-Ansätzen der Siebzigerjahre postulierten die Urheber mit dem CDP einen neuen und rigorosen Ansatz für die Software-Entwicklung, der folgende auch heute noch sehr ambitionierte Veränderungen anstrebte (Hausler, Linger, Trammell, 1994).

From		To
Individual operations	→	Team operations
Waterfall development	→	Incremental development
Informal specification	→	Black box specification
Informal design	→	Box structure refinement
Defect correction	→	Defect prevention
Individual unit testing	→	Team correctness verification
Path-based inspection	→	Function-based verification
Coverage testing	→	Statistical usage testing
Indeterminate reliability	→	Certified reliability

Abb. 10–14 *Angestrebte Veränderungen durch den CDP*

Um diese Veränderungen umzusetzen, basiert der CDP auf den folgenden Prinzipien:

- Ein großes Projekt wird durch ein inkrementelles Vorgehen (Abschnitt 9.5.4) aufgeteilt, also durch eine Serie kleiner Projekte ersetzt. Jedes dieser kleinen Projekte kann mit einigen (etwa fünf bis acht) Entwicklern in einigen Monaten durchgeführt und abgeschlossen werden.

- Der Aufwand für Analyse und Spezifikation ist weit höher als üblich. Der CDP schreibt keine spezielle Methode oder Notation vor, je nach Problem verwendet man die besten Verfahren und Darstellungen, die infrage kommen, also auch formale Techniken. Die Spezifikationen werden sehr gründlich inspiziert.
- Den Entwicklern wird keine Möglichkeit gegeben, den Code zu compilieren und zu testen. Von ihnen wird erwartet, dass sie Programme liefern, die nicht nur semantisch, sondern auch syntaktisch fehlerfrei sind.
- Auf einen Test der einzelnen Komponenten wird ganz verzichtet. Stattdessen werden sie integriert und insgesamt getestet (siehe Abschnitt *Der statistische Test*).
- Fehler werden nur in sehr geringem Umfang akzeptiert und nicht von den Testern, sondern von den Entwicklern behoben. Zeigen sich in einer Komponente zu viele Fehler, wird sie nicht geflickt, sondern verworfen, also neu implementiert.

Linger (1993) gibt für den Verzicht auf den Komponententest eine interessante Begründung: Testet man die Komponenten einzeln und entdeckt dabei Abweichungen von der Spezifikation, so erhält man oft kein vollständiges Bild des Problems. Denn viele Fehler entstehen durch subtile Inkompatibilitäten zwischen den Modulen. Die Korrektur beseitigt dann nicht den Fehler, sondern nur seine Symptome. Auf diese Weise wird die spätere Entdeckung und Beseitigung des Fehlers im Systemtest weniger wahrscheinlich, der Fehler mogelt sich durch die Prüfungen und bleibt im System. Wird der Fehler dagegen erst im Systemtest angezeigt, sind seine Wirkungen offensichtlich, der Fehler kann »enttarnt« und behoben werden.

Linger nennt beim Eintritt in den ersten Test als Fehlerrate 3 bis 4 Fehler pro KLOC (Kilo Lines of Code) gegenüber ca. 25/KLOC bei traditioneller Codierung. Oft wurde beobachtet, dass bereits die allererste Übersetzung des Codes keine Fehler meldete. Die Publikationen sprechen von außerordentlich erfolgreichen Projekten verschiedenster Arten (Software-Komponente für einen Hubschrauber, COBOL Structuring Facility) und drastisch gesenkten Kosten für spätere Korrekturen.

Spezifikation und Entwurf mit Black, State und Clear Boxes

Eine Black-Box-Spezifikation beschreibt eine Software-Einheit (unabhängig von ihrer Größe) durch den Zusammenhang zwischen Ein- und Ausgaben. Da die Systeme zustandsbehaftet sind, genügt es nicht, nur die aktuelle Eingabe zu betrachten, es muss jeweils auch die Eingabe-Historie, die Sequenz der früheren Eingaben, berücksichtigt werden. Wir sind damit auf der Ebene der Abstrakten Datentypen, wie sie in reiner Form mit der algebraischen Spezifikation definiert werden. Im CDP wird aber keine strikt mathematische Formalisierung verlangt, man verwendet Tabellen und präzise (natürliche) Sprache. Beispielsweise kann

man die Reaktion eines Editors auf den Befehl »speichern« etwa wie folgt beschreiben:

- Wenn keine Datei geöffnet ist, wird eine Fehlermeldung erzeugt.
- Wenn eine Datei geöffnet ist, seit dem Öffnen aber keine Veränderung daran vorgenommen wurde, hat der Befehl keinen Effekt.
- Wenn eine Datei geöffnet ist und verändert wurde, wird die Datei in der aktuellen Form abgespeichert.

Aus der Black-Box-Darstellung wird die State Box entwickelt: An die Stelle der Vorgeschichte tritt der Systemzustand. Im Beispiel oben bedeutet das: Der Editor ist in einem von drei Zuständen (plus dem Zustand, bevor er gestartet wird): Diese Zustände können als »keine Datei«, »unveränderte Datei« und »veränderte Datei« bezeichnet werden. Durch den Befehl »speichern« wird im Zustand »keine Datei« eine Fehlermeldung hervorgerufen, der Zustand bleibt gleich; im Zustand »unveränderte Datei« hat »speichern« gar keine Wirkung. Im Zustand »veränderte Datei« wird gespeichert, und der Zustand wird auf »unveränderte Datei« gesetzt.

Die Clear-Box-Darstellung schließlich liegt auf der Ebene des Programmcodes, sie implementiert die State Box durch Datenstrukturen und den Ablauf durch Ablaufstrukturen. Damit liegt ein Feinentwurf vor, die Implementierung des Systems ergibt sich durch Komposition der Clear Boxes.

Die Clear Box wird gegen die State Box verifiziert, die State Box gegen die Black Box. Da die Darstellungen nicht streng formal sind, handelt es sich um eine Inspektion. Jeder im Projekt muss zustimmen, dass die Umsetzung korrekt ist.

Im Zuge der Entwicklung können gleichzeitig Boxes in verschiedenen Stadien vorliegen; typischerweise verwendet eine State Box oder eine Clear Box Verfeinerungen, die noch im Zustand der Black Box sind.

Der statistische Test

Testen dient dazu, Fehler zu entdecken; das wird in Kapitel 18 ausführlich dargestellt. Natürlich gilt das grundsätzlich auch für den Test im CDP. Aber ein Test hat auch stets eine andere Seite: Wir schließen – in aller Regel ohne große Überlegung – aus dem im Test beobachteten Verhalten auf den Einsatz: Hat ein Programm im Test völlig zuverlässig funktioniert, dann rechnen wir auch mit einem problemlosen Einsatz.

Im CDP wird dieser zweite Aspekt in den Vordergrund gerückt. Der Test ist so angelegt, dass man nicht nur »aus dem Bauch«, sondern mit guten Gründen daraus Prognosen der Zuverlässigkeit ableiten kann. Überspitzt könnte man sagen: Da die Fehler durch die Entwicklung vermieden werden, brauchen sie im Test nicht gesucht zu werden. Vielmehr zeigt der Test, ob die geforderte Zuverlässigkeit erreicht ist. Der statistische Test gehört zu den umstrittenen Ideen des Cleanroom-Ansatzes (vgl. Abschnitt 18.7).

10.5.3 Bewertung des Cleanroom Development Process

Cleanroom Development wurde in mehreren industriellen Projekten erfolgreich eingesetzt (Hausler, Linger, Trammell, 1994, S. 92; Linger, 1994). Die publizierten Ergebnisse, insbesondere die geringe Zahl der Fehler, die erst im Betrieb entdeckt wurden, sind beeindruckend. Viele der publizierten und nach dem Cleanroom-Ansatz durchgeführten Projekte haben eingebettete Systeme realisiert, deren Zuverlässigkeit besonders kritisch ist.

Die Vorteile lassen sich wie folgt zusammenfassen:

- Durch den systematischen Einsatz rigider, auch mathematischer Modellierungs- und Verifikationstechniken und durch den Test mit Zufallsdaten, die dem Benutzungsprofil entsprechen, entstehen Systeme, die ungewöhnlich fehlerarm sind und deren Zuverlässigkeit im Einsatz prognostiziert werden kann.
- Da die (Teil-)Ergebnisse einer statistischen Kontrolle unterliegen, ist jederzeit klar, ob der Entwicklungsprozess den Vorgaben entspricht. Damit ist die Prozessverbesserung ein Nebeneffekt des Prozesses.

Eigentliche Nachteile sehen wir nicht, wohl aber Voraussetzungen, die in der Praxis oft nicht erfüllbar sind:

- Die Anforderungen an die Entwickler sind hoch. Sie müssen in der Lage und bereit sein, nach den strengen Prinzipien des Prozesses zu arbeiten. Insbesondere müssen sie mit den formalen Techniken vertraut sein und sich auf die ungewohnte Herausforderung einstellen, hochwertige Programme abzuliefern, ohne sie getestet zu haben.
- Die Organisation muss bereits an das methodische Arbeiten und an die iterative Entwicklung gewöhnt sein. Da die Entwickler ohnehin mit vielen neuen Anforderungen umgehen müssen, können nicht zugleich auch noch neue Vorgehensweisen vermittelt werden.
- Dem Cleanroom-Ansatz liegt – wenigstens für jede Iteration – eine streng sequenzielle Vorgehensweise zugrunde. Was nicht spezifiziert ist, kann nicht entwickelt werden. Bei vielen eingebetteten Systemen ist diese Voraussetzung gegeben; die technischen Randbedingungen sind relativ stabil. Dagegen entsteht Software mit komplexer Bedienoberfläche und viel Interaktion in der Regel nicht auf der Basis einer stabilen Spezifikation, sondern explorativ nach dem Trial-and-Error-Prinzip. Darum sind solche Projekte für den Cleanroom-Ansatz nicht geeignet; sie lassen sich dagegen erfolgreich nach dem in vieler Hinsicht komplementären Konzept der sogenannten agilen Prozesse entwickeln (siehe Abschnitt 10.6).

Ein echtes Problem, das den Cleanroom-Ansatz mit den agilen Prozessen verbindet, ist der Mangel an neutralen Bewertungen: Hier wie dort (und auf vielen anderen Baustellen des Software Engineerings) gibt es reichlich viele positive Dar-

stellungen und Kommentare, aber kaum distanzierte, für alle Ergebnisse offene Untersuchungen. Darum bleibt es schwierig, ohne eigene Erfahrungen Aussagen über den wahren Wert der Konzepte zu machen.

10.6 Agile Prozesse

10.6.1 Die agile Bewegung

Was der Kunde haben und der Hersteller liefern will, ist funktionierende Software. Das war schon so, bevor der Begriff »Software Engineering« entstand, und es ist auch heute noch so. Aber wir haben gelernt, dass – wie in jeder Disziplin – Umwege nötig sind, um das Ziel sicher und insgesamt zu vertretbaren Kosten zu erreichen. Wenn man sich im Auto anschnallt oder gelegentlich Wartungsarbeiten am Auto machen lässt, dient das nicht direkt dem schnellen Transport von A nach B; die meisten Menschen haben aber verstanden, dass das Anschnallen trotzdem sinnvoll ist und regelmäßige Ölwechsel billiger sind als der Verzicht darauf.

Entsprechend ist auch das »Drauflos-Programmieren« (Code and Fix, siehe Abschnitt 9.1.1) weder sicher noch rentabel. Planung, Spezifikation, Architekturentwurf, Qualitätssicherung und Projektverfolgung sind die Schlüsselwörter der systematischen Software-Bearbeitung.

Allerdings tun sich viele Menschen mit diesen Umwegen schwer. Darum wurden Prozesse definiert, die die Entwickler führen. Diese Prozesse können vom Entwickler als Gängelung erlebt werden, besonders dann, wenn sie bürokratisch formuliert sind und durchgesetzt werden. Ein Prozess, der sich selbst dient und nicht dem übergeordneten Ziel, mit minimalem Aufwand qualitativ ausreichende Software bereitzustellen, hat mehr mit Kafka als mit den traditionellen Ingenieurprinzipien zu tun.

Die Neigung, solche bürokratischen Prozesse zu installieren, scheint in den USA besonders ausgeprägt zu sein. Sie provozierte eine Gegenbewegung: Entwickler, die ihre Situation erlebten, als sei sie die Vorlage zu den Dilbert-Cartoons, beschlossen gegen Ende der Neunzigerjahre, den Ballast der »schweren Prozesse« (siehe Abschnitt 10.1.2) abzuwerfen und sich auf die ursprüngliche Zielsetzung zu konzentrieren. Diese Tendenz wurde unterstützt durch die rasch wachsende Bedeutung, die zwei neue Anwendungsgebiete der Informatik gewannen, das Internet und die mobile Kommunikation. Auf beiden Gebieten hat die Fähigkeit, rasch auf die Wünsche der Kunden oder des Marktes zu reagieren, ein deutlich größeres Gewicht als die Existenz einer Dokumentation, die eine lange Lebensdauer möglich macht. Und auch wenn ein Fehler in der Software eines Mobiltelefons für den Hersteller schmerzhaft sein mag, so sind die Risiken doch nicht mit denen eines fehlerhaften Airbags vergleichbar.

Es entstand eine Reihe von »Anti-Prozessen« (Extreme Programming, Adaptive Software Development, Crystal, Scrum), die unter der Sammelbezeichnung

»agile Prozesse« bekannt wurden. Abrahamsson et al. (2002) haben die Ziele im Abstract griffig formuliert:

> *Agile – denoting »the quality of being agile; readiness for motion; nimbleness, activity, dexterity in motion« – software development methods are attempting to offer an answer to the eager business community asking for lighter weight along with faster and nimbler software development processes. This is especially the case with the rapidly growing and volatile Internet software industry as well as for the emerging mobile application environment.*

10.6.2 Das »Agile Manifesto«

2001 haben sich die Protagonisten der agilen Prozesse in Utah getroffen und ein *Agile Manifesto* verfasst (Abb. 10–15).

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

that is, while there is value in the items on the right, we value the items on the left more.

Abb. 10–15 *The Agile Manifesto (2001)*

Manifest: Wem fällt bei diesem Wort nicht das kommunistische Manifest von 1848 ein? Mancher kennt auch das GNU-Manifest, das Dadaistische Manifest oder das Bauhaus-Manifest. Im »object-oriented database manifesto« (Atkinson et al., 1989) wurde den traditionellen Datenbanken die objektorientierte Alternative gegenübergestellt.

Alle diese Manifeste sind Streitschriften, in denen eine Minderheit, die Helden eines kleinen Dorfes in Gallien, der feindlichen Übermacht mit moralischen oder ästhetischen Argumenten den Krieg erklärt: So auch das »Manifest für die agile Software-Entwicklung«.

Seine Verfasser haben sich auf die folgenden Prinzipien geeinigt:

- *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
- *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
- *Business people and developers must work together daily throughout the project.*
- *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
- *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
- *Working software is the primary measure of progress.*
- *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
- *Continuous attention to technical excellence and good design enhances agility.*
- *Simplicity – the art of maximizing the amount of work not done – is essential.*
- *The best architectures, requirements, and designs emerge from self-organizing teams.*
- *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

10.6.3 Gemeinsamkeiten der agilen Prozesse

Das Manifest drückt die Ablehnung bestimmter Positionen und die Präferenz für andere aus. Viel weiter reichen die Gemeinsamkeiten nicht. Wenn wenigstens klar wäre, was ein typischer nicht agiler Prozess ist, dann wären die agilen Ansätze durch seine Negation definiert. In Deutschland würde sich dafür das V-Modell anbieten, ein Prozessmodell, das die Eleganz und Wendigkeit eines Armee-Lastwagens hat. Aber die Fronten sind verwischt: Das V-Modell bietet die agile Entwicklung als eine von mehreren Möglichkeiten an (siehe Abschnitt 10.3.5).

Schon ein zutreffender Sammelbegriff fehlt: Wir verwenden durchgehend das Wort »Prozessmodell«, auch wenn in manchen Fällen eher von Arbeitsprinzipien oder Organisationsgrundsätzen zu sprechen wäre. Mit diesen Einschränkungen kann man sagen, dass alle agilen Prozesse

- iterativ angelegt sind; die Zyklen sind meist einige Wochen lang, höchstens drei Monate,
- die Arbeit in einer kleinen Gruppe verlangen, meist aus sechs bis acht Leuten, die gemeinsam an einem Ort arbeiten (inzwischen gibt es auch Modelle für größere Gruppen, siehe Abschnitt 10.6.6),
- die Idee einer großen, reichlichen Dokumentation ablehnen, radikal oder mit Einschränkungen,
- den Kunden sehr wichtig nehmen und seine Präsenz im Projekt empfehlen oder verlangen,

- dogmatische Regelungen ablehnen (auch wenn das beim Extreme Programming, siehe Abschnitt 10.6.4, manchmal anders klingt).

Die Modelle haben unterschiedliche Schwerpunkte, sodass sie kombiniert werden können. Etwa seit 2010 hat sich Scrum praktisch überall, wo Software bearbeitet wird, durchgesetzt. Extreme Programming, einst eine Revolution, ist als Entwicklungsprozess praktisch verschwunden; viele der eingeführten Praktiken beispielsweise »Kontinuierliche Integration« oder »Strukturverbesserung« werden jedoch in agilen Projekten angewendet.

Nachfolgend betrachten wir kurz Extreme Programming und dann ausführlicher Scrum. Zu allen anderen agilen Prozessmodellen (und natürlich auch zu den hier behandelten) empfehlen wir die Übersicht von Abrahamsson et al. (2002).

10.6.4 Extreme Programming

Von allen agilen Prozessen hat das *Extreme Programming* (XP) die mit Abstand größte Aufmerksamkeit erreicht. Dies liegt auch daran, dass Kent Beck, der Urheber, keine Scheu vor plakativen Aussagen hat, die gut ankommen.

XP ist gut dokumentiert, viele Quellen sind im Internet verfügbar (z. B. Wells, o. J.). Neben dem Buch von Kent Beck (1999) gibt das Buch von Wolf, Roock und Lippert (2005) einen guten Einstieg in XP und beschreibt Erfahrungen beim Einsatz von XP in Entwicklungsprojekten.

Grundlagen

Die zentralen Elemente von XP sind die sogenannten XP-Werte (*values*), die XP-Prinzipien (*basic principles*) und die XP-Praktiken (*practices*). Kent Beck hat diese zentralen Elemente in der zweiten Auflage seines Buches (Beck, Andres, 2004) von Grund auf überarbeitet und erweitert. Wir beziehen uns hier jedoch auf die erste Fassung, die auch Grundlage der meisten Arbeiten über die Anwendung und die Erfahrungen mit XP war.

XP stellt als agiles Prozessmodell die Menschen, d. h. alle Projektbeteiligten, in den Mittelpunkt, nicht Dokumente, Werkzeuge oder Prozesse. Dies spiegelt sich in den vier Werten, auf denen XP basiert:

- *Einfachheit*
  Es sollen möglichst einfache Lösungen erstellt und möglichst einfache Prozesse genutzt werden. Einfache Lösungen sind leichter herzustellen und schneller zu verstehen als komplexe.
- *Feedback*
  Das Projektteam soll so schnell und so oft wie möglich Feedback über die erzielten Ergebnisse erhalten. Rückmeldungen sind Teil der Qualitätssicherung; sie kommen von den Anwendern, aber auch von den Kollegen im Team.

- *Kommunikation*
  Hier steht die persönliche und direkte Kommunikation zwischen den Projektbeteiligten im Vordergrund. Dies verlangt, dass das Projektteam nicht räumlich getrennt arbeitet und dass Kunde und Anwender in hohem Maße verfügbar sind. Dokumente als Mittel zur Kommunikation sind zweitrangig; sie ergänzen die direkte Kommunikation, wenn dies nötig ist.
- *Mut*
  XP betrachtet Mut als eine wichtige Voraussetzung, um die oben beschriebenen Werte im Projekt auch zu leben.

XP leitet aus diesen Werten Prinzipien ab. Diese sind, wie beispielsweise die Prinzipien »Ehrliches Messen« oder »Verantwortung übernehmen«, offensichtlich vernünftig.

XP-Praktiken

Wir stellen nachfolgend die XP-Praktiken vor, da diese sich auf die konkrete Projektarbeit auswirken und helfen, die Werte und Prinzipien im Projekt umzusetzen. XP definiert 14 Praktiken, die in Management-, Team- und Programmier-Praktiken gruppiert werden können (Wolf, Roock, Lippert, 2005). Abbildung 10–16 zeigt diese Gruppierungen von außen nach innen. Die Praktiken und die Regeln dazu erscheinen auf den ersten Blick einfach.

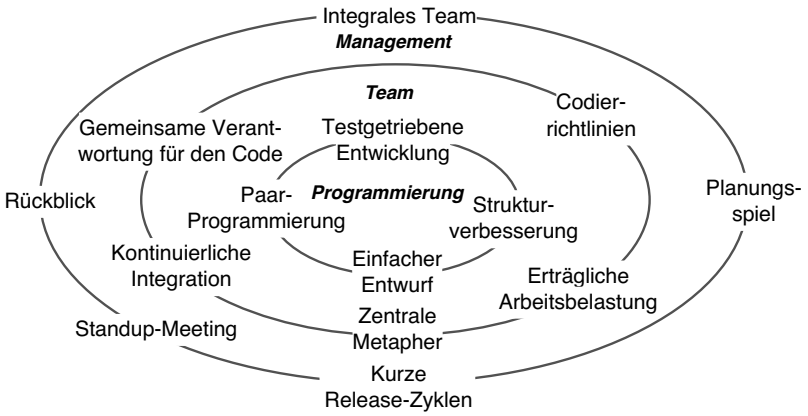

Abb. 10–16 *XP-Praktiken (nach Wolf, Roock, Lippert, 2005)*

- **Management-Praktiken**
  - *Integrales Team*
    Ein XP-Projekt verlangt zwingend den Kunden im Projekt, weil nur dieser die fachlichen Anforderungen kennt und nach seinen Prioritäten ordnen

kann. Der Kunde muss jederzeit für Fragen und Diskussionen zur Verfügung stehen (on-site customer).

- *Planungsspiel* (oder *Planungssitzung*)
  XP ist ein iterativer und inkrementeller Prozess. Jedes Inkrement wird gemeinsam durch das Projektteam im sogenannten Planungsspiel festgelegt. Der Kunde gibt die zu realisierenden Anforderungen vor, die Entwickler schätzen die dafür benötigte Zeit sowie den Aufwand. Natürlich können hier die Vorstellungen beider Seiten weit auseinander liegen. Dann ist es Aufgabe der Entwickler, die Planungssitzung so lange zu moderieren, bis eine gemeinsame Basis für das nächste Inkrement gefunden ist.
- *Kurze Release-Zyklen*
  Damit die Anwender möglichst schnell und oft Feedback geben können, darf die Zeit zwischen den Releases nicht zu groß sein (im Bereich weniger Wochen).
- *Standup-Meeting*
  Die »Sitzungen« des (einzigen) Teams, das keine ausgeprägte Hierarchie aufweist, finden täglich im Stehen statt und sind kurz (ca. 15 min). Sie dienen dazu, die Aufgaben zu verteilen und sich über den Fortgang des Projekts, über Probleme usw. auszutauschen.
- *Rückblick*
  In längeren Abständen werden spezielle Sitzungen mit ausgewählten Personen durchgeführt, in denen das Projekt rückblickend bewertet wird. Ziel ist es, Fehler und Schwächen zu identifizieren, um daraus für kommende Projektabschnitte oder neue Projekte zu lernen.

Team-Praktiken

- *Gemeinsame Verantwortung für den Code*
  Der Code gehört allen, und alle haben das Recht und die Pflicht, ihn weiterzuentwickeln und zu verbessern (Refactoring).
- *Codierrichtlinien*
  Alle schreiben Code gemäß gemeinsam vereinbarten Richtlinien.
- *Erträgliche Arbeitsbelastung*
  Überstunden gibt es höchstens ausnahmsweise, besser gar nicht. Damit soll vermieden werden, dass die Teilnehmer im Projekt verschlissen werden.
- *Zentrale Metapher*
  Metaphern (vgl. Abschnitt 2.5) in der Software-Entwicklung helfen, den Anwendungsbereich und die Architektur zu strukturieren. Jedes XP-Projekt soll sich an einer Metapher orientieren, die vor allem dem fachlichen Entwurf zugrunde liegt. Bekannte und erprobte Metaphern sind beispielsweise »Desktop« oder »Werkzeug-Automat-Material« (Züllighoven, 2005).

- *Kontinuierliche Integration*
  Die Ergebnisse werden laufend in einer speziellen Umgebung (im Integrationsrechner) integriert. So werden Änderungen am System sofort allen Entwicklern zur Verfügung gestellt. Nach jeder Integration müssen alle für das Gesamtsystem vorhandenen Tests ohne Fehler ablaufen. Nur dann verbleibt der neu integrierte Code im Gesamtsystem.

Programmier-Praktiken

- *Testgetriebene Entwicklung*
  Testen geht vor Implementieren, d. h., jede Implementierung einer neuen Funktion beginnt mit der Implementierung der entsprechenden Tests (Unit Tests), die automatisch ausgeführt werden. Akzeptanztests dienen dazu, implementierte fachliche Anforderungen abzunehmen.
- *Strukturverbesserung (Refactoring)*
  Werden Schwächen im Code oder im Entwurf entdeckt, dann werden sie beseitigt, bevor neue Funktionen realisiert werden (zum Refactoring siehe Abschnitt 22.3).
- *Einfacher Entwurf*
  Die gewählten Entwürfe sollen die zu realisierenden Anforderungen möglichst einfach umsetzen. Im Wechsel mit den Strukturverbesserungen wird die Architektur inkrementell erweitert.
- *Paar-Programmierung (Pair Programming)*
  Die Codierung wird grundsätzlich durch Programmierer im Doppelpack geleistet; einer codiert, der andere schaut zu, prüft und hinterfragt den erstellten Code. Diese Rollen wechseln ständig. Dadurch wird die Qualität des Codes erhöht, und das Wissen ist über mehrere Personen verteilt. Die Paare sind nicht stabil, sie wechseln ebenfalls.

Viele dieser Praktiken können nicht einzeln umgesetzt werden, da sie vielfache Bezüge untereinander haben; so bedingen sich beispielsweise die Praktiken »Planungsspiel« und »Integrales Team«. Trotzdem ist es möglich, die XP-Praktiken in sinnvoll zugeschnittenen Gruppen schrittweise einzuführen.

Bewertung des Extreme Programming

Die XP-Praktiken befreien die Entwickler von einigen Mühen und Lasten, die traditionell auf ihren Schultern liegen, vor allem von der Dokumentation, auch von einem mühsamen Prozess der Anforderungserhebung, von Review-Sitzungen, vom Ausfüllen der Stundenzettel und vom Kampf mit einer Planung, die am grünen Tisch gemacht wurde. Die Befreiung ist aber keineswegs total; tatsächlich ist XP weder chaotisch noch anarchisch, es erfordert im Gegenteil große Disziplin. Wer beispielsweise die gemeinsame Verantwortung für den gesamten Code so interpretiert, dass er nach Belieben darin herumflicken darf, wird sehr rasch

zurückgepfiffen; aber das wird schon daran scheitern, dass natürlich auch das Refactoring in Zweiergruppen geschieht. (Zitat: *Any pair of programmers can improve any code at any time*.) Die Einfachheit der Praktiken ist teilweise vordergründig. Eine zentrale Metapher, das klingt plausibel und ist für die Kommunikation mit dem Kunden vorteilhaft. Beck spricht aber von einer Metapher, die nicht nur für den Kunden (zur Erklärung), sondern auch für die Entwickler (als Architekturvorgabe) verbindlich ist. Das kann im Allgemeinen nicht funktionieren und funktioniert auch nicht. Die Metapher taugt als Leitbild, nicht als Entwurf. Beck selbst hat nach viel Kritik die Bedeutung der Metapher in der zweiten Auflage seines Buches erheblich reduziert.

Wer aus Erfahrung skeptisch ist und Kent Beck fragt, wie er die Praktiken im Detail umsetzt, stellt überrascht fest, dass die Realität nicht ganz den klaren Regeln entspricht. *Natürlich* denkt er vor Beginn des eigentlichen XP-Projekts über die Konzeption nach, und zwar weder im Zweierteam noch in Anwesenheit des Kunden. *Natürlich* setzt man sich bei den Besprechungen (»ist bequemer«). Und *natürlich* ist keineswegs immer ein Kunde verfügbar, der anwesend ist, testen kann und Entscheidungsbefugnis hat. Ein XP-Projekt wird auch nicht abgebrochen, wenn sich eine ungerade Anzahl verfügbarer Mitarbeiter ergibt, die sich nicht mehr ohne Rest auf Entwickler-Paare verteilen lässt. Kurz: Der lautstarke Protest gegen die »schweren« Prozessmodelle ist ein wesentlicher Teil des Programms.

10.6.5 Scrum

Scrum ist ein agiler Ansatz zum Management von Software-Entwicklungsprojekten. Er wurde von Jeff Sutherland, Ken Schwaber und Mike Beedle Mitte der Neunzigerjahre entwickelt und erstmals 1995 publiziert (Schwaber, 1995). Scrum basiert auf Ideen, die Takeuchi und Nonaka (1986) im Kontext der *Lean Production* entwickelt haben, um industrielle Produktionsverfahren den sich immer schneller ändernden Anforderungen anzupassen. Im Kern ihrer Ideen steht das sich selbst organisierende Team, bestehend aus verschiedenen Spezialisten, die nicht streng arbeitsteilig, sondern kooperativ die Entwicklung vorantreiben und auch den Erfolg oder Misserfolg kollektiv verantworten. Takeuchi und Nonaka vergleichen diesen Organisationsansatz mit dem Verhalten einer Rugby-Mannschaft, die nach jeder Unterbrechung ein Gedränge bildet (engl. *Scrum*), das den Ball zu erobern versucht. Ist das gelungen, so folgt ein schneller *Sprint* mit dem Ball, der möglichst viel Raumgewinn bringen soll.

Scrum definiert einen rollenbasierten iterativen und inkrementellen Prozess zur Software-Entwicklung. Scrum beschränkt sich im Gegensatz zu XP auf das Management agil durchgeführter Projekte, es werden keine Techniken vorgeschlagen, die zu nutzen sind. Dementsprechend kann Scrum selbst schnell erlernt und auch schnell eingeführt werden. Scrum ist gut dokumentiert – wir stützen

uns auf die Bücher von Schwaber und Beedle (2001), Pichler (2008) sowie Wolf und Roock (2021) – und wird erfolgreich in vielen Unternehmen eingesetzt.

Rollen in Scrum

Scrum definiert drei Rollen, die in jedem Scrum-Projekt wahrgenommen werden müssen:

- Der *Product Owner* (Produktverantwortlicher) vertritt die Interessengruppen außerhalb des Projektteams, insbesondere die des Auftraggebers. Er kennt die fachlichen und technischen Anforderungen an das zu entwickelnde System und sammelt und pflegt diese im *Product Backlog*. Der Product Owner plant und entscheidet, welche Anforderungen in der nächsten Iteration (dem sogenannten *Sprint*) realisiert werden. Dadurch hat er maßgeblichen Einfluss auf das Projektergebnis. Der Product Owner arbeitet eng mit dem Projektteam zusammen, um die Anforderungen zu klären. Er sollte, wenn möglich, (passiv) an den täglichen Projekttreffen (*Daily Scrum*) teilnehmen, damit er stets über Fortschritt und Probleme informiert ist. Der Product Owner muss in der Lage sein, am Ende jedes Sprints die Umsetzung der Anforderungen zu beurteilen.
- Ein *Scrum-Team* muss so zusammengestellt sein, dass seine Mitglieder in der Lage sind, alle Entwicklungstätigkeiten autonom durchzuführen. Das Team entscheidet auf Basis der Priorisierung der Anforderungen durch den Product Owner, wie viele Anforderungen im nächsten Sprint realisiert werden und welche Arbeitsschritte dafür benötigt werden. Ein Scrum-Team organisiert sich selbst; es gibt keinen Teamleiter, der die Aufgaben verteilt. Das Team entscheidet, welche Aufgaben durchzuführen sind und wer wann welche Aufgaben übernimmt. Dies ist insbesondere bei neuen Teams nicht leicht. Hilfe kann das Scrum-Team vom Scrum Master bekommen (siehe unten). Zusätzlich stehen Dokumente zur Verfügung, die das Team bei der Planung und Fortschrittskontrolle des Projekts unterstützen. Weil die Mitglieder eines Scrum-Teams sehr intensiv kommunizieren und miteinander arbeiten, müssen ihre Arbeitsplätze nahe beieinander liegen (am besten im selben Raum).
- Scrum führt mit dem *Scrum Master* eine völlig neue Rolle für Entwicklungsprojekte ein. Die übergeordnete Aufgabe des Scrum Masters besteht darin, dem Team und der Organisation zu helfen, Scrum-Projekte richtig durchzuführen. Er sorgt dafür, dass der Prozess und die Regeln eingehalten und die definierten Verantwortlichkeiten auch wahrgenommen werden. Der Scrum Master unterstützt das Team und den Product Owner und kümmert sich darum, dass Hindernisse beseitigt werden, die den erfolgreichen Einsatz von Scrum gefährden. Weiterhin stellt er sicher, dass das Team direkt mit dem Product Owner zusammenarbeiten kann und dass es während eines Sprints nicht durch ungerechtfertigte Eingriffe gestört wird. Der Scrum Master moderiert

die Daily Scrums und sorgt dafür, dass die zentralen Scrum-Dokumente (wie das Product Backlog) aktuell sind. Neben diesen eher organisatorischen Aufgaben sollte ein kompetenter Scrum Master auch in der Lage sein, die Arbeitstechniken und Vorgehensweisen zu bewerten und auch zu verbessern. Scrum Master ist also die wichtigste Rolle im Scrum-Projekt. Ob die Einführung von Scrum gelingt, hängt vor allem vom Scrum Master ab.

Dokumente

Scrum definiert als agiler Prozess nur wenige, dafür aber für den Erfolg wichtige Dokumente, die in jedem Scrum-Projekt zu erstellen und zu pflegen sind. Diese Dokumente unterstützen vor allem die Planung von Scrum-Projekten und dienen der Fortschrittskontrolle.

Das wichtigste Dokument eines Scrum-Projekts ist das *Product Backlog*. Es enthält alle Anforderungen, die im Projekt umgesetzt werden sollen. Zusätzlich werden im Product Backlog alle Aufgaben aufgeführt, die erledigt werden müssen. Der Product Owner erstellt und aktualisiert das Product Backlog. Ausgangsbasis für die erste Version des Product Backlogs ist das Produktkonzept oder die Sammlung der Anforderungen, die in einem Anforderungsworkshop (siehe Abschnitt 15.2.8) erhoben werden.

Die Einträge im Product Backlog sind priorisiert, den erforderlichen Aufwand hat das Team geschätzt. Als Schätzgröße werden in der Regel nicht Personentage, sondern Punkte (*Story Points*) verwendet. Diese werden bei der Planung eines Sprints in Personentage umgerechnet. Die Aufwandsbewertung der Anforderungen findet in Schätzklausuren statt, an denen das Team, der Product Owner und in der Regel auch der Scrum Master teilnehmen.

Alle Einträge, die im nächsten Sprint umgesetzt werden sollen, werden aus dem Product Backlog in das *Sprint Backlog* übernommen. Dieses Dokument beschreibt somit die Ziele, die ein Sprint erreichen soll, und dient dem Team zur Organisation der Arbeit während eines Sprints. Die Einträge im Sprint Backlog sind möglichst präzise beschrieben und in Personenstunden oder Personentagen abgeschätzt. Welche Anforderungen in einem Sprint umgesetzt werden, wird in der Sprint-Planungssitzung festgelegt. Das Sprint Backlog wird täglich aktualisiert, d. h., es werden umgesetzte Anforderungen als erledigt markiert, die Aufwände dafür werden dokumentiert, und eventuelle Restaufwände und neue Aktivitäten und deren Aufwände werden eingetragen.

Um den Arbeitsfortschritt während eines Sprints zu dokumentieren, kennt Scrum den sogenannten *Sprint-Burndown-Bericht*. Dieser zeigt, typischerweise grafisch, über die Laufzeit des Sprints die Höhe der Aufwände, die im Sprint Backlog enthalten oder noch zu erbringen sind. Im Idealfall nehmen die Aufwände stetig und etwa linear ab. Natürlich können aber auch neue nicht vorhergesehene Tätigkeiten dazukommen, geplante Aufwände können sich als ungenügend erweisen. Durch den Sprint-Burndown-Bericht wird frühzeitig sichtbar, ob

die im Sprint zur Implementierung vorgesehenen Anforderungen mit den geplanten Aufwänden realisiert werden können; andernfalls müssen Gegenmaßnahmen ergriffen werden. Beispielsweise kann der Product Owner Anforderungen aus dem Sprint Backlog streichen. Der Sprint-Burndown-Bericht wird täglich aktualisiert, meist ist der Scrum Master dafür verantwortlich.

Am Ende jedes Sprints wird ein Sprint-Endebericht erstellt, der insbesondere dokumentiert, welche Anforderungen umgesetzt wurden und welche nicht realisiert werden konnten. Zudem werden etwaige Hindernisse, die im Sprint aufgetreten sind und zu Problemen geführt haben, beschrieben.

Natürlich muss bei einem größeren System geplant werden, in welchen Schritten (Ausbaustufen) das System realisiert (und wenn möglich auch eingesetzt) werden soll. Dazu wird auf Basis der initialen Version des Product Backlogs ein *Release-Plan* erstellt. Dieser legt fest, wie viele Sprints durchzuführen sind, um das Produkt zu erstellen, und welche zentralen Anforderungen in den einzelnen Sprints umgesetzt werden. Nach jedem Sprint wird der Release-Plan auf Basis der neuen Erfahrungen und Erkenntnisse aktualisiert. Analog zur Sprint-Fortschrittskontrolle wird auf Basis des Release-Plans ein *Release-Burndown-Bericht* erstellt und gepflegt.

Der Scrum-Prozess

Scrum verlangt einen iterativen Prozess, das Produkt entsteht inkrementell. Abbildung 10–17 zeigt die zentralen Elemente des Scrum-Prozesses, erweitert um Elemente der Release-Planung. Am Anfang müssen wie üblich Anforderungen gesammelt, priorisiert und bewertet werden. Dazu kann ein Anforderungsworkshop durchgeführt werden. Die gesammelten Anforderungen werden im Product Backlog festgehalten, das als Basis dient, um die Releases und die einzelnen Inkremententwicklungen, die Sprints, zu planen. Die Anforderungen, die in einem Sprint umgesetzt werden sollen, werden aus dem Product Backlog in das Sprint Backlog übernommen. Im Sprint werden alle Entwicklungstätigkeiten durchgeführt, die notwendig sind, um das Ziel des Sprints zu erreichen.

Ein wesentliches Element des Sprints ist das sogenannte Daily Scrum, ein tägliches Treffen, nur eine Viertelstunde lang. Alle Teammitglieder berichten über ihre Fortschritte und stimmen die Planung des Tages ab. Neue Hindernisse werden besprochen und dokumentiert. So kennen alle den Stand des Projekts. Damit 15 Minuten ausreichen, muss das Treffen vorbereitet sein, das Sprint Backlog und der Sprint-Burndown-Bericht müssen aktualisiert vorliegen. Am Daily Scrum nehmen das Team, der Scrum Master und, wenn möglich, der Product Owner teil.

Ein Sprint sollte nicht länger als 30 Tage dauern, kürzere Sprints, oft sind es zwei Wochen, sind die Regel. Am Ende jedes Sprints muss ein ausführbares und auslieferbares Produktinkrement vorliegen. Das ist jedoch, insbesondere bei den ersten Sprints, nicht immer sinnvoll. Pichler (2008) unterscheidet darum zwi-

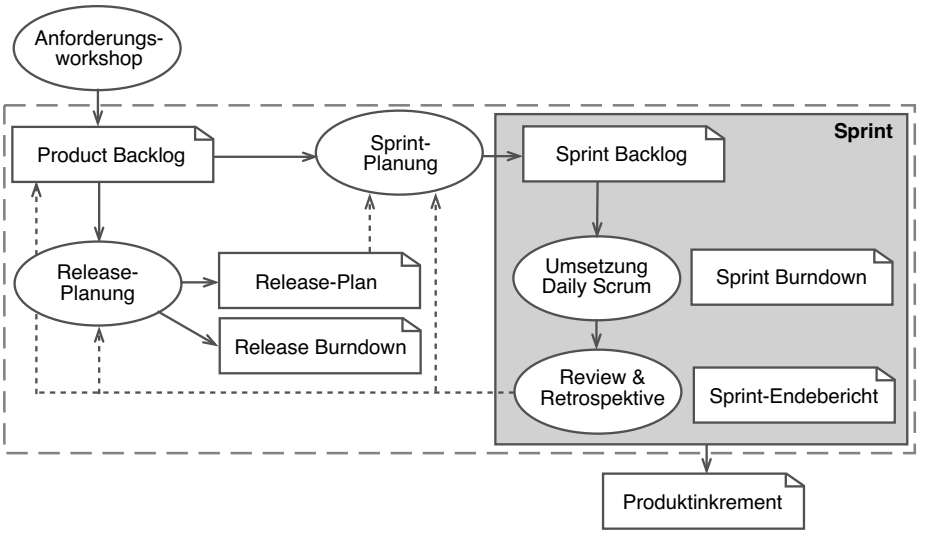

Abb. 10–17 *Elemente des Scrum-Prozesses*

schen Explorations-, Standard- und Release-Sprints. Explorations-Sprints dienen dazu, problematische und riskante Bereiche zu untersuchen und Lösungsalternativen zu erproben; das Ergebnis ist in der Regel ein Prototyp. Explorations-Sprints ähneln somit dem explorativen Prototyping (Abschnitt 9.4.4) oder einem Zyklus im Spiralmodell (Abschnitt 9.6). Während das Ergebnis eines Standard-Sprints ein Produktinkrement ist, das nicht unbedingt freigegeben und ausgeliefert wird, entsteht bei einem Release-Sprint immer ein auslieferbares Produktinkrement.

Am Ende jedes Sprints finden zwei Bewertungssitzungen statt. Im *Sprint-Review* wird festgestellt, ob das Team alle Anforderungen vollständig und in der geforderten Qualität umgesetzt hat. Dazu präsentiert das Team die Ergebnisse und diskutiert sie mit dem Product Owner. Der Product Owner nimmt die Implementierung ab, die erledigten wie auch die unerledigten Anforderungen werden im Sprint-Endebericht dokumentiert. In der *Sprint-Retrospektive* wird bewertet, wie gut der Scrum-Prozess im betrachteten Sprint umgesetzt werden konnte. Falls möglich, werden Verbesserungsmaßnahmen für kommende Sprints identifiziert. So wird der Scrum-Prozess stetig verbessert.

Die Ergebnisse des Sprints, des Sprint-Reviews und der Retrospektive fließen in die weitere Projektplanung ein, d. h., Product Backlog und Release-Plan müssen entsprechend aktualisiert werden, bevor ein neuer Sprint geplant werden kann.

10.6.6 Scrum im Großen

Scrum ist immer dann geeignet, wenn ein Team ein gesamtes Projekt bearbeiten kann. Das trifft aber eher selten zu, da der Umfang und die Komplexität von Software-Produkten typischerweise so groß sind, dass mehrere Teams dafür benötigt werden. In diesem Fall müssen zusätzliche Organisationselemente eingeführt werden, da gleichzeitig mehrere Scrum-Teams koordiniert werden müssen.

Aus diesem Grund wurden Organisationsmodelle entwickelt und vorgestellt, die es erlauben, die zentralen Konzepte und Elemente von Scrum auch in großen Projekten zu nutzen. Nachfolgend stellen wir exemplarisch die Modelle Scrum of Scrums und Nexus vor.

Scrum of Scrums

Jeff Sutherland hat bereits 2001 mit *Scrum of Scrums* ein einfaches Organisationselement eingeführt, um mehrere Teams zu koordinieren (Sutherland, 2001). Ein Scrum of Scrums ist eine spezielle Form eines Daily Scrums, in dem jedes Team durch ein Mitglied des Teams vertreten ist. Es dient dazu, dass sich die Teams gegenseitig über den aktuellen Stand informieren, die Arbeit der verschiedenen Teams synchronisiert wird und Abhängigkeiten und Hindernisse identifiziert werden, die andere Teams bei der Umsetzung der Backlog-Einträge beeinflussen. Scrum of Scrums finden nicht täglich statt, sondern etwa alle zwei oder drei Tage. Im Meeting berichtet jeder Teamvertreter zu folgenden Fragen (Cohn, 2007):

- Was hat das Team seit dem letzten Scrum of Scrums geschafft?
- Was wird das Team bis zum nächsten Scrum of Scrums erledigt haben?
- Gibt es Hindernisse, die die Arbeit des Teams erschweren?
- Könnte irgendetwas, das das Team tut, die Arbeit eines anderen Teams behindern?

Dieser Koordinationsmechanismus ist jedoch nur dann effektiv, wenn es nur wenige und einfache Abhängigkeiten zwischen den Teams gibt, die abgestimmt werden müssen.

Weiterhin darf die Zahl der zu koordinierenden Teams nicht zu groß sein. Paasivaara, Lassenius und Heikkilä (2012) berichten, dass in diesem Fall das Scrum-of-Scrums-Meeting die angestrebten Ziele nicht mehr erreichen kann. Prinzipiell kann man Scrums of Scrums über mehrere Ebenen durchführen, also beispielsweise ein Scrum of Scrums of Scrums nutzen, um mehrere Teilprojekte zu koordinieren, die sich intern über ein Scrum of Scrums organisieren. In der Literatur gibt es aber keine Belege, dass das skaliert.

Das Nexus™ Framework[3]

Nexus (lat. Verknüpfung) ist ein Organisationsmodell für Scrum-Projekte, bei denen mehrere Teams, in der Regel zwischen zwei und neun Teams, gemeinsam ein Produkt entwickeln oder weiterentwickeln. Ken Schwaber, einer der Scrum-Väter, hat Nexus entworfen; es wird detailliert in Bittner, Kong und West (2018) vorgestellt. Wir beziehen uns im Folgenden auf dieses Buch.

Alle Teams, die zusammen als Nexus bezeichnet werden, bearbeiten in ihren Sprints die Einträge aus einem gemeinsamen Product Backlog. Es wird angestrebt, dass alle Teams am Ende jedes Sprints ein gemeinsames integriertes Produktinkrement herstellen. Um die Teams zu koordinieren, erweitert Nexus die Elemente von Scrum so, dass sie auf der Ebene aller Teams genutzt werden können.

Neben dem Product Backlog und den teamspezifischen Sprint Backlogs enthält ein *Nexus Sprint Backlog* alle aus dem Product Backlog entnommenen Einträge, die die Teams im nächsten Sprint gemeinsam umsetzen. Wichtig ist, dass die Abhängigkeiten, die während des Sprints zwischen den Arbeiten der Teams bestehen, identifiziert und im Nexus Sprint Backlog dokumentiert sind.

Um die Zusammenarbeit der Teams zu koordinieren und deren Arbeit zu unterstützen, führt Nexus eine zusätzliche, sehr wichtige Rolle ein, das *Nexus-Integrationsteam*. Es besteht aus dem Product Owner, dem Scrum Master und aus Mitgliedern der Teams. Zusätzlich können weitere Personen, beispielsweise aus den Bereichen Betrieb oder Sicherheit, im Nexus-Integrationsteam temporär mitwirken, wenn ihr fachliches Wissen benötigt wird, um einen Sprint oder mehrere Sprints erfolgreich durchzuführen.

Das Nexus-Integrationsteam ist nicht, wie der Name suggeriert, für die Integration zuständig, sondern es unterstützt die Teams dabei, am Ende jedes Sprints ein gemeinsames *integriertes Produktinkrement* zu liefern. Das Nexus-Integrationsteam ist insbesondere dafür verantwortlich, Abhängigkeiten zwischen Einträgen im Product Backlog zu erkennen und diese möglichst zu reduzieren. Abhängigkeiten können architektonische Gründe haben. Aber auch organisatorische oder personelle Abhängigkeiten müssen betrachtet werden. Da der Projekterfolg direkt von der erfolgreichen Arbeit des Nexus-Integrationsteams abhängt, müssen alle benötigten Kompetenzen in diesem Team vorhanden sein. Auch sollten die Mitglieder ihre Aufgaben im Nexus-Integrationsteam in Vollzeit ausführen. Deshalb sollten die in das Nexus-Integrationsteam entsandten Teammitglieder von ihren Aufgaben im Team befreit sein.

Folgende zusätzliche Elemente führt Nexus ein:

- Im *Refinement* werden die Product-Backlog-Einträge teamübergreifend bearbeitet, verfeinert und wenn nötig aufgeteilt. Daran nehmen Mitglieder aus

3 Nexus™ Framework ist ein eingetragenes Warenzeichen der Firma Scrum.org.

den einzelnen Teams und der Product Owner teil. Beim Refinement ist darauf zu achten, dass die Backlog-Einträge möglichst unabhängig voneinander und ohne übermäßigen Koordinierungsaufwand von den einzelnen Teams umgesetzt werden können. Natürlich gibt es immer Abhängigkeiten, diese sind dann aber bekannt und können bei der Planung des nächsten Sprints berücksichtigt werden. Das Refinement erfolgt so oft wie nötig. Ziel ist, dass die in der nächsten Nexus-Sprint-Planung betrachteten Einträge ausreichend detailliert ausgearbeitet sind, um auf die Teams aufgeteilt zu werden.

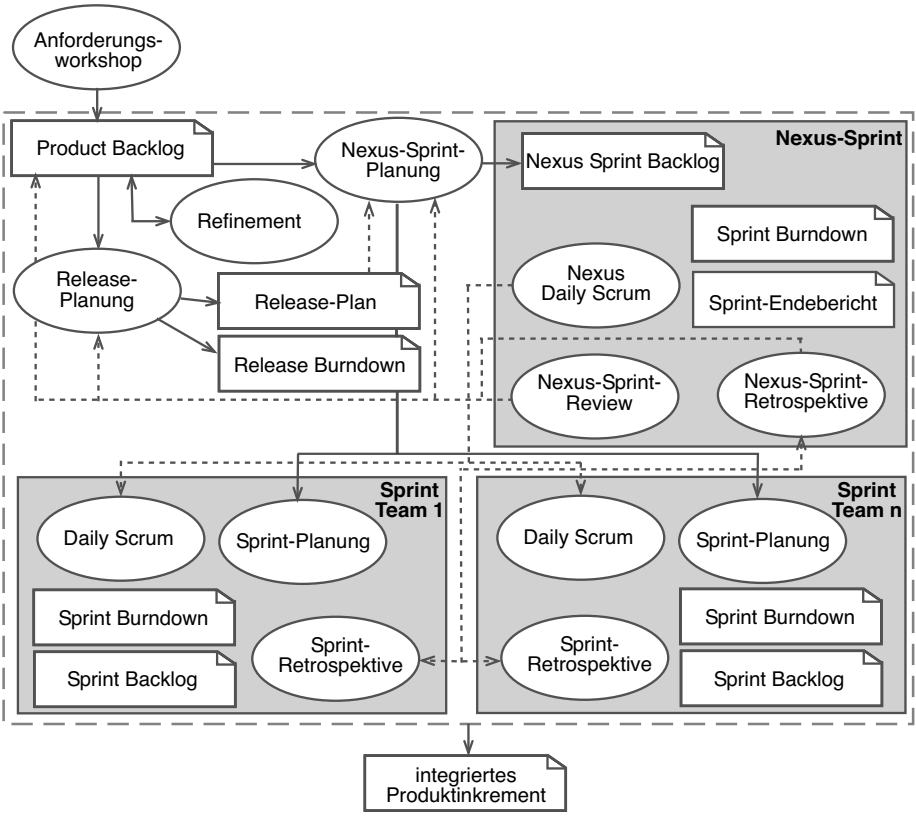

Abb. 10–18 *Elemente des Nexus-Prozesses*

- In der *Nexus-Sprint-Planung* wird das gemeinschaftliche Ziel, das sogenannte *Nexus-Ziel*, des nächsten Sprints festgelegt. Es werden die verfeinerten Product-Backlog-Einträge für den nächsten Sprint ausgewählt und in den Nexus Sprint Backlog aufgenommen. Die Einträge werden auf die Teams aufgeteilt und jedes Team erstellt dafür eine teamspezifische Sprint-Planung. Wenn Abhängigkeiten zwischen den Einträgen der Teams existieren, dann berücksichtigen die Teams diese gemeinsam im Rahmen ihrer Sprint-Planung. Die

Nexus-Sprint-Planung ist abgeschlossen, wenn alle Teams ihre Sprint-Planung erstellt haben, und diese, wenn nötig, mit den anderen Teams abgestimmt ist.

- Am *Nexus Daily Scrum* nehmen die Vertreter der Teams teil, die die teamübergreifenden Abhängigkeiten und Integrationsprobleme im aktuellen Sprint bearbeiten. Dadurch werden die Abhängigkeiten zwischen den Teams kontinuierlich betrachtet und abgestimmt, damit die Teams diese absprachegemäß umsetzen können. Das Nexus Daily Scrum findet sinnvollerweise immer vor den Daily Scrums der einzelnen Teams statt, um dort die im Nexus Daily Scrum gemachten Absprachen berücksichtigen zu können.
- Am *Nexus-Sprint-Review* nehmen alle Mitglieder aller Teams, das Nexus-Integrationsteam sowie interessierte Vertreter des Kunden teil. Das im Sprint entwickelte Produktinkrement wird durch die Teams vorgestellt. Der Product Owner und die Kundenvertreter diskutieren und kommentieren die erzielten Ergebnisse und nehmen diese ab, wenn sie erfolgreich umgesetzt wurden. Wenn alle Einträge im Nexus Sprint Backlog erfolgreich umgesetzt wurden, ist das Nexus-Ziel erreicht. Wenn dieses nicht oder nur in Teilen erreicht werden konnte, dann müssen die Gründe dafür in der Nexus-Sprint-Retrospektive herausgefunden und Konsequenzen daraus gezogen werden. Abhängig von den Ergebnissen des Sprints muss das Product Backlog ggf. angepasst werden. Da alle Teams am gemeinsamen Produktinkrement arbeiten, entfallen die teamspezifischen Sprint-Reviews in einem Nexus-Projekt.
- Die *Nexus-Sprint-Retrospektive* dient dazu, den abgeschlossenen Sprint im Bezug auf das Nexus-Ziel, auf Stärken und auf Probleme zu analysieren. So sollen hier beispielsweise neue erfolgreich eingesetzte Techniken und Verfahren kommuniziert, aber auch Bereiche identifiziert werden, in denen sich die Teams verbessern können. In der Nexus-Sprint-Retrospektive werden unter anderem folgende Fragen diskutiert:
  - Falls das Nexus-Ziel nicht oder nur in Teilen erreicht werden konnte: Was sind die Gründe dafür?
  - Wurden im Sprint neue Technische Schulden (Abschnitt 22.5) erzeugt? Wie geht man damit um? Können diese wieder abgelöst werden?
  - Wie kann verhindert werden, dass Probleme, die in diesem Sprint aufgetreten sind, wieder auftreten?

  Die Nexus-Sprint-Retrospektive besteht aus drei Teilen:
  - Zuerst treffen sich Mitglieder der einzelnen Teams, um wichtige Themen zu identifizieren, die mehr als ein Team betreffen.
  - Anschließend führt jedes Teams eine Sprint-Retrospektive durch, bei der die teamspezifischen Aspekte und insbesondere auch die im ersten Teil identifizierten teamübergreifenden Punkte diskutiert werden.

- Abschließend treffen sich wieder Vertreter aller Teams und die Mitglieder des Nexus-Integrationsteams, um die Ergebnisse der teamspezifischen Sprint-Reviews zu konsolidieren, zu diskutieren und um ggf. zu entscheiden, welche Verbesserungsmaßnahmen umgesetzt werden sollen.

Die Erfahrungen aus dem Nexus-Sprint-Review und der Nexus-Sprint-Retrospektive fließen in die Planung des nächsten Sprints ein.

Wie alle agilen Modelle definiert auch Nexus keine Projekt- oder Prozessdokumente. Natürlich ist es sinnvoll, die in verschiedenen Sitzungen diskutierten Ergebnisse zu dokumentieren. Dafür eignen sich Wiki-Systeme.

Nexus ist gut dokumentiert. Neben dem offiziellen Nexus Guide (Nexus, 2021) wird Nexus im Buch von Bittner, Kong und West (2018) detailliert beschrieben. Die Autoren stellen auch eine Reihe von Techniken vor, um die Elemente des Nexus-Prozesses erfolgreich umzusetzen. Über Probleme und Erfahrungen im Einsatz berichten Gakstatter und Schad (2018).

10.6.7 Zusammenfassung der agilen Prozesse

> *Now, a bigger gathering of organizational anarchists would be hard to find ...*
>
> aus der »History of the Agile Manifesto«

Die agilen Methoden können nicht pauschal charakterisiert oder bewertet werden; ihre Ähnlichkeit besteht vor allem in der Ablehnung der bürokratischen Prozesse. Weder das »Manifesto« noch die Prinzipien taugen als Definition eines Prozesses. Die hier behandelten Repräsentanten, Extreme Programming und Scrum, unterscheiden sich erheblich; während XP durch eine oft irrationale Begeisterung für das »Anders-Arbeiten« geprägt ist und damit seinem Etikett »extrem« gerecht wird, erweist sich Scrum als ein in vielen Situationen angemessenes Managementkonzept. Dass XP zunächst große Aufmerksamkeit genoss, lag vermutlich genau an seiner modischen Verpackung als Weltanschauung.

Die Ablehnung einer bürokratischen Verkrustung der Prozesse ist nachvollziehbar und erfreulich; wer ein Webportal oder ein Spiel für das Mobiltelefon entwickelt, ist gut beraten, nicht monatelang nach einer umfassenden Spezifikation zu streben, sondern sich in einem evolutionären Ansatz mit kurzen Zyklen an eine Lösung heranzutasten, die dem Kunden gefällt. Dies sind typische Einsatzgebiete für die agilen Prozesse.

Wer aber die Software für ein Fly-by-Wire-System im Verkehrsflugzeug oder für die Datenübertragung einer Großbank entwickelt, wird (hoffentlich) nicht auf die Sicherheit und Solidität verzichten, die die traditionellen Ansätze bieten. Bei solchen Projekten gibt es keinen einzelnen Kunden, sondern Experten und Benutzer, deren Anforderungen sehr systematisch erhoben werden müssen. Und es ist zu erwarten, dass das System über Jahrzehnte eingesetzt und gewartet wird.

Die Etablierung von Organisationsformen, um Scrum in großen Projekten oder sogar organisationsweit zu nutzen, ist für jedes Unternehmen eine Herausforderung und benötigt Zeit und Aufwand. Schon die Auswahl eines geeigneten Organisationsmodells – neben den beiden hier vorgestellten sind beispielsweise Srcum@Scale und Large Scale Scrum (LeSS) potenzielle Kandidaten – ist schwierig und birgt Risiken. Diebhold, Schmitt und Theobald (2018) haben Kriterien entwickelt, um solche Organisationsmodelle zu vergleichen, und geben Empfehlungen für den Auswahlprozess.

Letztlich muss aber jedes Unternehmen seine spezifisch an die Gegebenheiten des Unternehmens angepasste Implementierung finden. »One size fits all« gilt auch in diesem Kontext nicht.

Es soll nicht unerwähnt bleiben, dass das von Dean Leffingwell 2011 vorgestellte und stetig weiterentwickelte Scaled Agile Framework (SAFe) ein umfangreiches Organisationsmodell ist, um agile Prozesse im gesamten Unternehmen mit dem Ziel zu etablieren, die Wertschöpfung zu verbessern (Knaster, Leffingwell, 2020).

Das religiöse Fieber um XP und Scrum hat sich gelegt. Dagegen wird es weiterhin – und hoffentlich durch solide empirische Daten unterstützt – Versuche geben, für jede Klasse von Projekten den angemessenen Prozess zu finden.