HANSER

Software Engineering

Gustav Pomberger, Wolfgang Pree

Architektur-Design und Prozessorientierung

ISBN 3-446-22429-7

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/3-446-22429-7 sowie im Buchhandel

2 Prozessmodelle

Prozessmodelle dienen der Benennung, Beschreibung und Ordnung von Tätigkeiten im Rahmen der Software-Entwicklung. In der Literatur sind dazu viele Vorschläge zu finden, von denen hier nur eine Auswahl vorgestellt wird. Grundlage für die Auswahl sind der Verbreitungsgrad und die Praxisrelevanz dieser Modelle sowie das Bestreben, dem Leser mehrere unterschiedliche Ansätze darzulegen.

2.1 Das klassische sequenzielle Phasenmodell

Phasenmodelle wurden in zahlreichen Variationen beschrieben. Sie postulieren im Wesentlichen folgende Phasen des Software-Entwicklungsprozesses:

- Problemanalyse und Grobplanung
- Systemspezifikation und Planung
- System- und Komponentenentwurf
- Implementierung und Komponententest
- System- und Integrationstest
- Betrieb und Wartung

Die allen Varianten von Phasenmodellen zugrunde liegende Vorgehensweise bei der Software-Entwicklung beruht auf dem Prinzip, dass für jede der Phasen klar zu definieren ist, welche Ergebnisse erzielt werden müssen, und dass eine Phase erst dann in Angriff genommen werden darf, wenn die vorhergehende vollständig abgeschlossen ist. Die Anwendung dieser streng sequenziellen Vorgangsweise soll dazu führen, dass Software-Projekte besser planbar, organisierbar und kontrollierbar werden.

Mit dem Begriff *Software Life Cycle* verbindet man die Vorstellung von einer Zeitspanne, in der ein Software-Produkt geplant, entwickelt und eingesetzt wird, bis zum Ende seiner Nutzung. Außerdem verbindet man damit die Vorstellung von einer Strukturierung dieses Zeitraums in Phasen und damit verbundenen Aktivitäten und Ergebnissen sowie die Festlegung der Reihenfolge von und den Beziehungen zwischen diesen Phasen. Abbildung 2.1 zeigt das klassische (und entsprechend idealisierte) sequenzielle Phasenmodell.

Im Folgenden beschreiben wir die Ziele, die wichtigsten Tätigkeiten und die Ergebnisse der einzelnen Phasen, ohne darauf einzugehen, welche Methoden und Techniken zur Erreichung dieser Ziele eingesetzt werden können. Wir verweisen den Leser dazu auf die weiteren Kapitel dieses Buches.

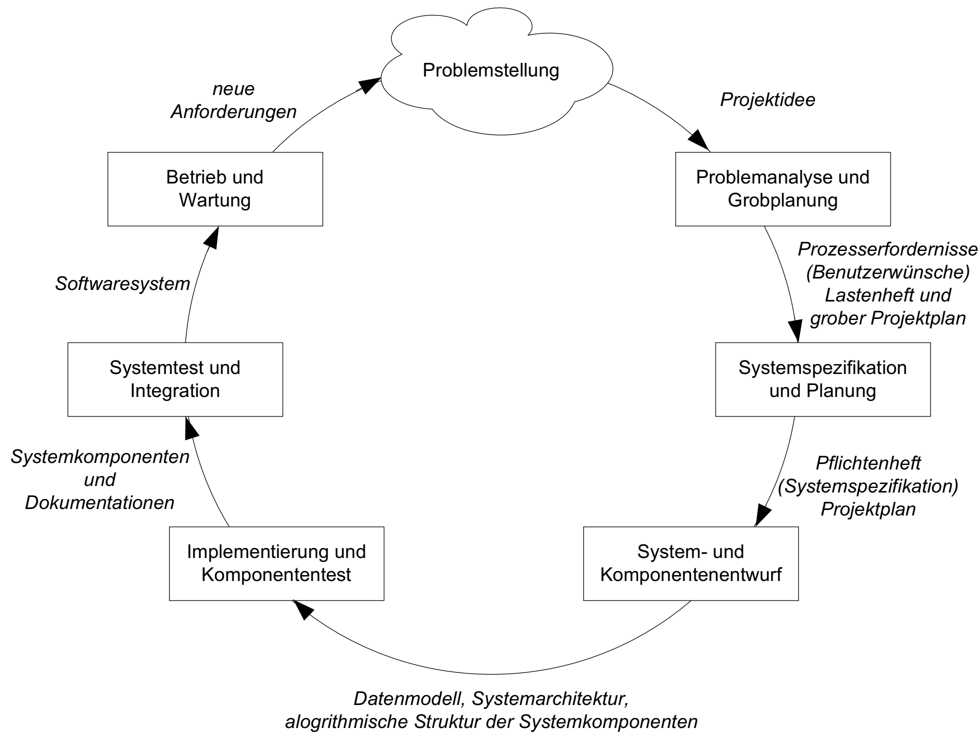

Abbildung 2.1: Sequenzielles Phasenmodell

Problemanalyse und Grobplanung

Das Ziel der Analyse- und Grobplanungsphase besteht darin, für den Einsatzbereich, für den eine Software-Lösung angestrebt wird, festzustellen und zu dokumentieren, welche Aktivitäten (Tätigkeiten) ausgeführt werden sollen und welcher Art ihre Wechselwirkungen sind, welche Teile davon automatisiert werden sollen und welche nicht, und welche technischen, personellen, finanziellen und zeitlichen Ressourcen für die Projektrealisierung zur Verfügung stehen.

Die wichtigsten Tätigkeiten sind die Erhebung des Ist-Zustands bzw. der Problemstellung und die Abgrenzung des Problembereichs, die grobe Skizzierung der Bestandteile des geplanten Systems, eine erste Abschätzung des Umfangs und der wirtschaftlichen Aspekte des geplanten Projekts und die Erstellung eines groben Projektplans.

Die Ergebnisse der Analyse- und Grobplanungsphase sind eine grobe Beschreibung des Ist-Zustands bzw. der Problemstellung, der Projektauftrag, das Lastenheft und ein grober Projektplan.

Systemspezifikation und Planung

Das Ziel der Spezifikationsphase ist ein Kontrakt zwischen dem Auftraggeber und dem Software-Hersteller, der genau festlegt, was das geplante Software-System leisten soll und welche Prämissen für seine Realisierung gelten.

Die wichtigsten Tätigkeiten sind die Erstellung der Systemspezifikation (auch Anforderungsdefinition oder Pflichtenheft genannt), die Festlegung eines genauen Projektplans, die Validierung der Systemspezifikation (das heißt die Prüfung der Vollständigkeit und Konsistenz der Anforderungen und die Prüfung der technischen Durchführbarkeit), sowie die ökonomische Rechtfertigung des Projekts.

Die Ergebnisse der Spezifikationsphase sind die Systemspezifikation in Form eines Pflichtenheftes und der genaue Projektplan.

System- und Komponentenentwurf

Das Ziel der Entwurfsphase besteht darin, festzulegen, durch welche Systemkomponenten die durch die Systemspezifikation vorgegebenen Anforderungen abgedeckt werden und wie diese Systemkomponenten zusammenarbeiten sollen.

Die wichtigsten Tätigkeiten sind der Entwurf der Systemarchitektur, das heißt die Definition der Systemkomponenten (zum Beispiel durch den Entwurf ihrer Schnittstellen) und die Festlegung ihrer Wechselwirkungen sowie, falls erforderlich, des zugrunde liegenden logischen Datenmodells, der Entwurf der algorithmischen Struktur der Systemkomponenten und die Validierung der Systemarchitektur sowie der Algorithmen, die die einzelnen Systemkomponenten realisieren.

Die Ergebnisse der Entwurfsphase sind die Beschreibung des logischen Datenmodells, der Systemarchitektur und der algorithmischen Struktur der Systemkomponenten sowie die Dokumentation der Entwurfsentscheidungen.

Implementierung und Komponententest

Das Ziel der Implementierungsphase besteht darin, die in der Entwurfsphase erhaltenen Ergebnisse in eine Form zu bringen, die auf einem Rechner ausführbar ist.

Die wichtigsten Tätigkeiten sind die vollständige Verfeinerung der Algorithmen für die einzelnen Komponenten, die Übertragung der Algorithmen in eine Programmiersprache (Codierung), die Entscheidung über den Zukauf bzw. die Wiederverwendung bereits verfügbarer Komponenten, die Übertragung des logischen Datenmodells in ein physisches Datenmodell (zum Beispiel in einer Datenbank), die Übersetzung und Prüfung der syntaktischen Richtigkeit der Algorithmen, das Testen, das heißt die Prüfung der semantischen Richtigkeit der Systemkomponenten, syntaktische und semantische Korrekturen der fehlerhaften Systemkomponenten.

Die Ergebnisse der Implementierungsphase sind die getesteten Systemkomponenten, die Protokolle der Komponententests und die physische Realisierung des logischen Datenmodells.

Systemtest und Integration

Das Ziel der Test- und Integrationsphase besteht darin, die Wechselwirkungen der Systemkomponenten unter realen Bedingungen zu prüfen, möglichst viele Fehler des neu entwickelten Systems aufzudecken, seine Funktionsfähigkeit nach Integration mit den „Umsystemen" zu garantieren und sicherzustellen, dass die Systemimplementierung die Systemspezifikation erfüllt.

Das Ergebnis der Test- und Integrationsphase ist ein einsatzfähiges System.

Betrieb und Wartung

Nach Abschluss der Test- und Integrationsphase wird das Software-Produkt zur Benutzung freigegeben. Aufgabe der Software-Wartung ist es, Fehler, die erst während des tatsächlichen Betriebs auftreten, zu beheben und Systemänderungen und -erweiterungen durchzuführen. Normalerweise ist dies, zeitlich betrachtet, die längste Phase des Software Life Cycles.

Neben den oben beschriebenen Phasen, die die sequenzielle Life Cycle-orientierte Software-Entwicklungsmethodik charakterisieren, müssen noch zwei weitere Aspekte hervorgehoben werden, die wichtige Elemente dieses Vorgehensmodells sind: die Dokumentation und die Qualitätssicherung. Die dafür notwendigen Tätigkeiten bilden keine eigentlichen Projektphasen, sondern müssen projektbegleitend, das heißt in allen Phasen durchgeführt werden.

Die Dokumentation soll während der Entwicklungsphasen die Kommunikation zwischen den an der Entwicklung beteiligten Personen ermöglichen und nach Abschluss der Entwicklungsphasen den Einsatz und die Wartung des Software-Produkts unterstützen. Sie soll außerdem den Projektverlauf zur Ermittlung der Herstellungskosten und zur besseren Planung zukünftiger Projekte dokumentieren.

Die Qualitätssicherung umfasst analytische, konstruktive und organisatorische Maßnahmen zur Qualitätsplanung sowie zur Erreichung von Qualitätszielen und Qualitätsmerkmalen, wie Korrektheit, Zuverlässigkeit, Benutzerfreundlichkeit, Wartungsfreundlichkeit, Effizienz und Portabilität. Darauf wird in Kapitel 3 näher eingegangen.

Vorgangsweise bei phasenorientierter Software-Entwicklung

Die Vorgangsweise beruht auf dem Prinzip der „Topdown-Dekomposition" so genannter „Black Boxes". Das heißt, es wird eine schrittweise Konkretisierung (siehe Abbildung 2.2) durchgeführt.

Als Ausgangsbasis für die einzelnen Phasen werden wohldefinierte „Produkte" (meist in Form von Dokumenten) verlangt. Diese Produkte werden in phasenbezogenen Prozessen – für die bestimmte Methoden und Werkzeuge eingesetzt werden – verarbeitet, und die Resultate werden an die nächste Phase weitergeleitet.

Die Phasen sind klar gegeneinander abgegrenzt und sollen erst dann verlassen werden, wenn in einem Verifikations-/Evaluierungsschritt die Ergebnisse akzeptiert worden sind.

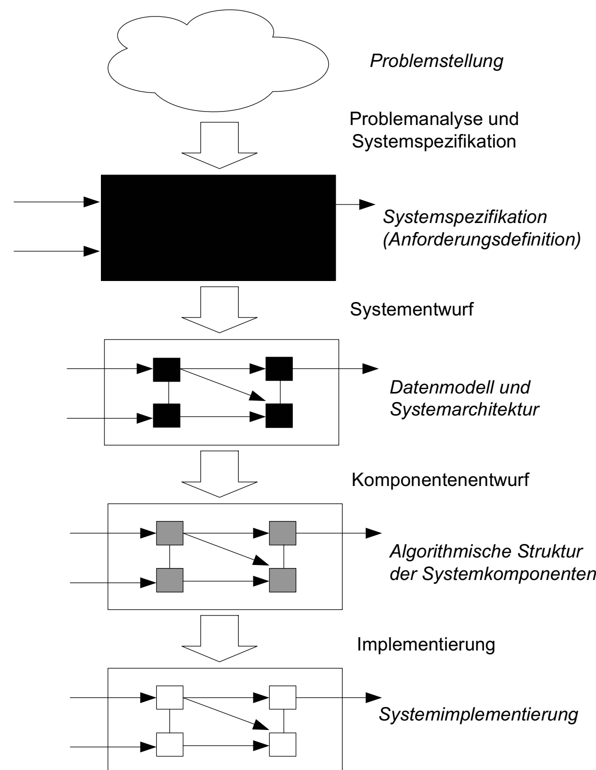

Abbildung 2.2: Schrittweise Konkretisierung nach dem sequentiellen Phasenmodell

Im Verifikationsschritt wird geprüft, ob ein Phasenergebnis (zum Beispiel die Systemarchitektur) die Systemspezifikation erfüllt, während durch den Evaluierungsschritt geprüft wird, ob das Produkt in der gerade vorliegenden Form tatsächlich die Benutzerwünsche erfüllt (es könnte ja die Spezifikation falsch sein).

Charakteristisch für die Vorgangsweise nach dem sequenziellen Phasenmodell ist, dass zunächst (und zwar während der Analyse- und Spezifikationsphase) das System „von außen" beschrieben wird. Es geht also vorerst darum, was das System leisten soll; wie diese Leistung erbracht wird, ist zunächst unerheblich. Das System selbst wird als schwarzer Kasten betrachtet, dessen Wirkung nach außen genau definiert und dessen innere Struktur nicht weiter festgelegt wird.

Danach wird, während der Entwurfsphase, wiederum in derselben Weise verfahren, indem festgelegt (spezifiziert) wird, in welche Komponenten das System zerlegt wird, was diese Komponenten leisten und wie sie zusammenarbeiten sollen – aber nicht, wie sie tatsächlich realisiert werden. Wenn dieser Prozess vollständig abgeschlossen ist, das heißt, wenn die Schnittstellen aller Systemkomponenten definiert sind, müssen sich die Entwickler beim Entwurf der algorithmischen Struktur der Systemkomponenten nur noch um die Komponentenspezifikation kümmern. Diese Vorgangsweise unterstützt eine suk-

zessive Verminderung der Komplexität sowohl des Entwicklungsprozesses als auch des Produkts. Danach werden die Algorithmen in eine Programmiersprache übertragen und einzeln ausgetestet.

Dem Zerlegungsprozess folgt eine Synthese der Systemkomponenten. Das Ergebnis dieser Synthese, das Gesamtsystem, muss dann selbstverständlich noch verifiziert werden.

Die Erfahrung zeigt, dass das sequenzielle phasenorientierte Prozessmodell, obwohl weit verbreitet, ein Idealmodell ist, dessen Vorgaben und Ziele selten erreichbar sind. Gerade die Praxis zeigt die Grenzen und Schwachstellen dieses Prozessmodells. Die folgende Aufzählung der wichtigsten Vor- und Nachteile soll dem Leser die Beurteilung der Tauglichkeit dieser Entwicklungsmethode erleichtern.

Vorteile

- Das Phasenmodell liefert einen klaren Rahmen, der die wichtigsten Tätigkeiten des Software-Entwicklungsprozesses definiert und gegeneinander abgrenzt. Eine derartige Einteilung erleichtert die Planung von Software-Projekten und die Abschätzung der Entwicklungskosten.
- Die Vorgangsweise kann unabhängig vom Anwendungsgebiet, von der Projektgröße und von der Komplexität des geplanten Produkts angewendet werden.
- Die durch das Modell vorgegebene Strukturierung des Entwicklungsprozesses fördert eine Strukturierung des Produkts, die den heute anerkannten Software-technischen Prinzipien weitgehend entspricht.
- Es wird ein arbeitsteiliger Entwicklungsprozess ermöglicht, wie er für große Projekte unabdingbar ist.

Nachteile

- Das Modell beruht auf der (falschen) Annahme, dass der Entwicklungsprozess in der Regel sequenziell ausgeführt werden kann und Iterationen zwischen den Phasen nur ausnahmsweise notwendig sind. In den verschiedenen Beschreibungen der phasenorientierten Entwicklungsprozesse werden zwar solche Iterationen stets mit einbezogen, jedoch bleibt meist unklar, wann und nach welchen Kriterien iteriert werden soll.
- Die strenge Anwendung dieses Prozessmodells fordert, dass eine Phase erst dann begonnen werden darf, wenn die vorhergehende vollständig abgeschlossen ist. Das heißt, dass die jeweiligen Zwischenprodukte vollständig sind. Die Realität zeigt aber, dass es nur in den seltensten Fällen auf Anhieb gelingt, zum Beispiel eine vollständige Spezifikation oder einen fehlerfreien Entwurf einer Systemarchitektur zustande zu bringen. Oft fördern erst nachgelagerte Phasen jene Informationen zutage, die notwendig sind, um eine Phase vollständig abzuschließen.
- Die strenge Trennung der einzelnen Phasen ist eine unzulässige Idealisierung. In Wirklichkeit überlappen sich die darin auszuführenden Tätigkeiten, und die Wechselwirkungen zwischen den Phasen sind viel komplexer, als dies durch das sequenzielle Modell zum Ausdruck kommt.

- Die streng sequenzielle Vorgangsweise hat zur Folge, dass erst sehr spät ein Produkt oder Produktteile vorliegen, die „greifbare“ Ergebnisse darstellen. Die Erfahrung zeigt, dass man beim Evaluierungsprozess in der Regel nicht ohne realitätsnahe Experimente auskommt. Das verhindert aber die streng sequenzielle phasenorientierte Vorgehensweise. Dies hat auch zur Folge, dass Änderungswünsche von den Benutzern erst sehr spät geäußert und dann unter Umständen nur mit hohem Aufwand berücksichtigt werden können.

Zusammenfassend lässt sich festhalten, dass die Life Cycle-orientierte Software-Entwicklungsmethode sicherlich eine wichtige Grundlage für eine ingenieurmäßige Vorgangsweise darstellt, allerdings mit gravierenden Nachteilen. Das Phasenmodell beschreibt Tätigkeiten, ihre Reihenfolge und den Charakter der Zwischenprodukte (Phasenergebnisse). Die in ihm festgelegten Tätigkeiten sind wohl notwendig, aber nicht hinreichend, und können durch keine noch so „geniale“ Methode umgangen werden. Die Praxis der Anwendung dieses Vorgehensmodells zeigt, dass man von dieser idealisierten, rein sequenziellen Vorgangsweise manchmal abgehen muss und das Ziel einer Systementwicklung nicht völlig unabhängig vom späteren Lösungsweg festlegen kann.

Die Erfahrungen in der Anwendung des klassischen sequenziellen Phasenmodells haben dazu geführt, dass mehrere Varianten entwickelt wurden, die sich vor allem auf eine weitere Unterteilung der Phasen und Aussagen über die Wechselwirkungen zwischen den einzelnen Phasen beziehen. Eine der bekanntesten Varianten ist das in Abbildung 2.3 dargestellte Wasserfall-Modell (siehe dazu Royce, 1970).

Das Wasserfall-Modell stellt eine aus den Erfahrungen entwickelte Verfeinerung des klassischen sequenziellen Phasenmodells dar und wurde in den 1970ern entwickelt. Es enthält im Wesentlichen zwei Erweiterungen:

1. Die Einführung von Rückkopplungen zwischen den Phasen zusammen mit der Einschränkung, Rückkopplungen möglichst nur bei aufeinander folgenden Phasen vorzusehen, um die „teure“ Nacharbeit, die infolge von Iterationen über mehrere Phasen hinweg entsteht, zu vermindern.
2. Die Einbindung der (möglichst experimentellen) Validierung der Phasenergebnisse in das Vorgehensmodell.

Mit diesem Ansatz wird die streng sequenzielle Vorgangsweise, die das klassische Life-Cycle-Modell vorschreibt, aufgeweicht und eine inkrementelle Entwicklungsstrategie in den Mittelpunkt gerückt. Durch so genanntes „Zweimal-Bauen“ vor allem der Systemspezifikation und der Systemarchitektur (zuerst als Prototyp zur experimentellen Validierung und schnellen Korrektur und dann erst als Phasenprodukt) soll das Risiko von unvollständigen Systemspezifikationen und von Design-Fehlern vermindert werden.

Die Einbindung einer schrittweisen Entwicklungsstrategie für die Systemspezifikation und die Systemarchitektur sowie einer phasenweisen Validierung soll helfen, die Auswirkungen von Fehlentscheidungen besser in den Griff zu bekommen und den Software-Entwicklungsprozess besser kontrollierbar zu gestalten.

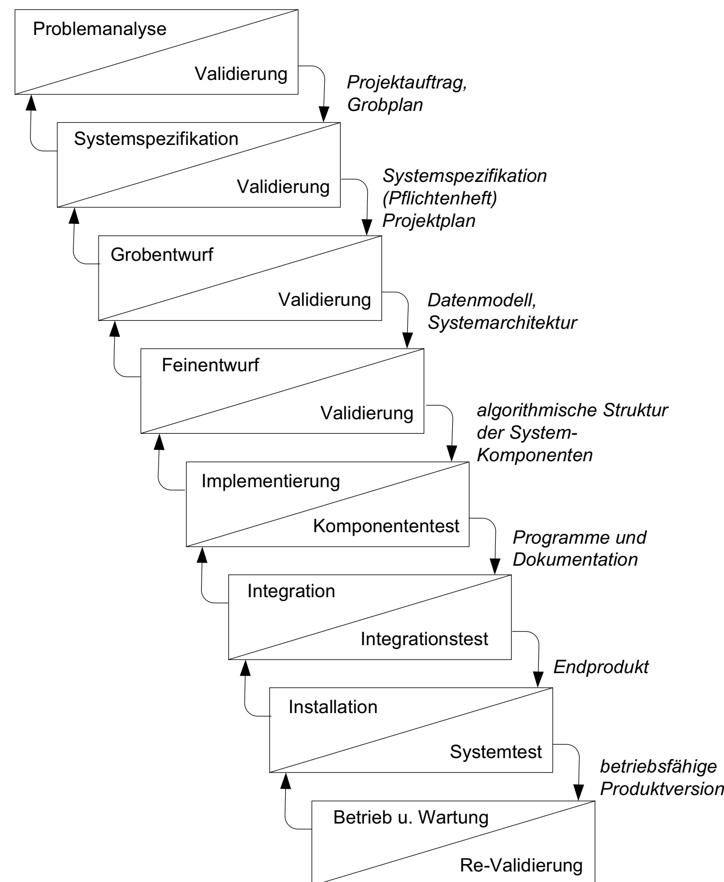

Abbildung 2.3: Das Wasserfall-Modell

2.2 Das V-Modell

Das V-Modell (siehe dazu zum Beispiel BWB IT 15, 1997, Versteegen, 2000, Balzert, 1998) ist eine Erweiterung des Wasserfall-Modells. Es gliedert den Software-Entwicklungsprozess (ähnlich dem Wasserfall-Modell) in eine Sequenz von Phasen und integriert explizit den Qualitätssicherungsprozess, und zwar so, dass eine klare Zuordnung der Phasen des Qualitätssicherungsprozesses zu den Phasen des Entwicklungsprozesses in Form eines V (siehe Abbildung 2.4) vorgenommen wird. Eine Darstellung des Vorgehensmodells zur Illustration der (Haupt-)Aktivitäten, ihrer Reihenfolge, ihrer Wechselwirkungen und der erwarteten Ergebnisse zeigt Abbildung 2.5 (Quelle: Balzert, 1998), Aktivitäten werden durch ein Rechteck, Ergebnisse durch ein abgerundetes Rechteck dargestellt und Pfeile geben an, welche Aktivitäten zu welchen Ergebnissen bzw. Produkten führen bzw. welche Produkte vor welcher Aktivität vorhanden sein müssen.

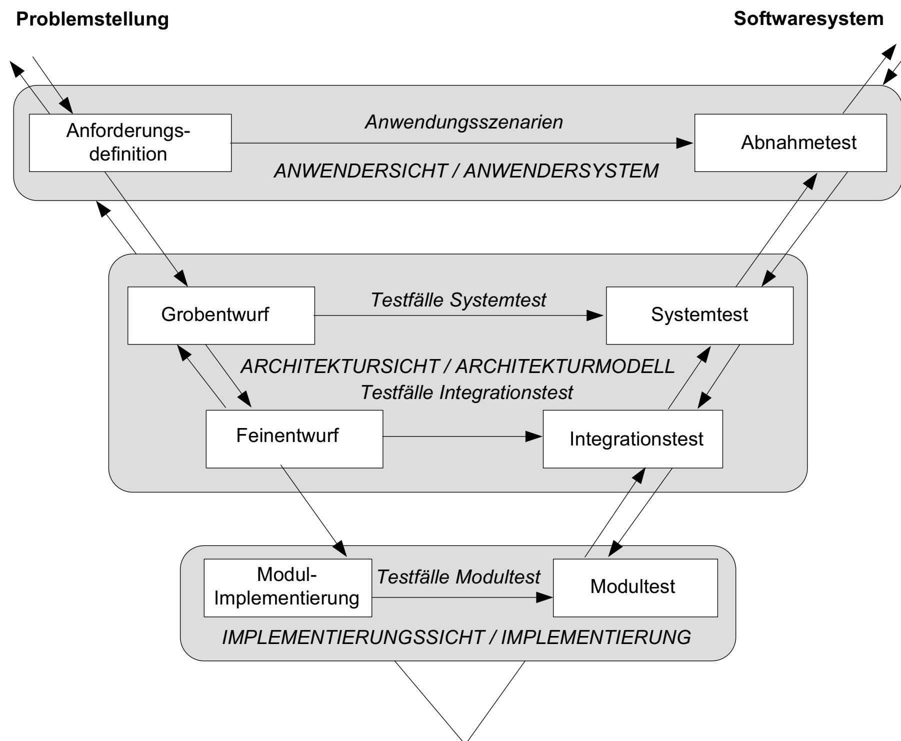

Abbildung 2.4: Das V-Modell

In diesem Prozessmodell wird insbesondere die Zusammengehörigkeit von Phasenergebnissen (Produkten) des Entwicklungsprozesses und den erforderlichen Tests hervorgehoben. Darüber hinaus strukturiert das Modell den Software-Entwicklungsprozess in drei Sichten:

- Die *Anwendersicht*, die sich in Form eines Anwendungssystems, bestehend aus einer Systemspezifikation und dem abgenommenen Software-Produkt, manifestiert.
- Die *Architektursicht*, die sich in Form eines Architekturmodells, bestehend aus den Spezifikationen von Teilsystemen, den dazu entworfenen Systemarchitekturen, den getesteten Teilsystemen und dem getesteten Gesamtsystem, manifestiert.
- Die *Implementierungssicht*, die sich in Form der Implementierung, bestehend aus den Modulspezifikationen und den getesteten Modulen, manifestiert.

Das V-Modell ist in vier Teilmodelle gegliedert:

- Teilmodell SE: Systemerstellung,
- Teilmodell QS: Qualitätssicherung,
- Teilmodell KM: Konfigurationsmanagement und
- Teilmodell PM: Projektmanagement.

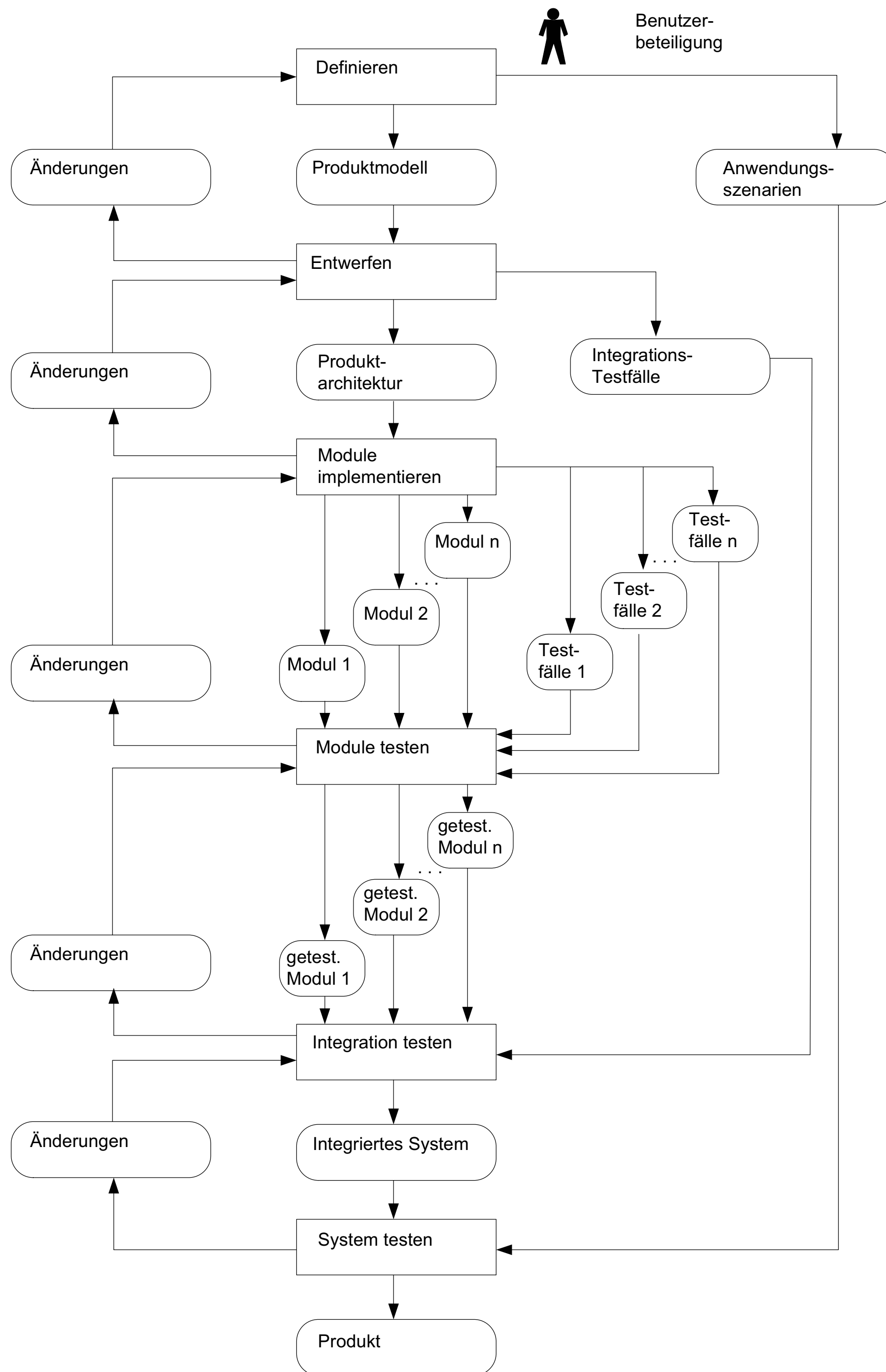

Abbildung 2.5: Aktivitäten und Produkte des V-Modells (Balzert, 1998, S. 102)

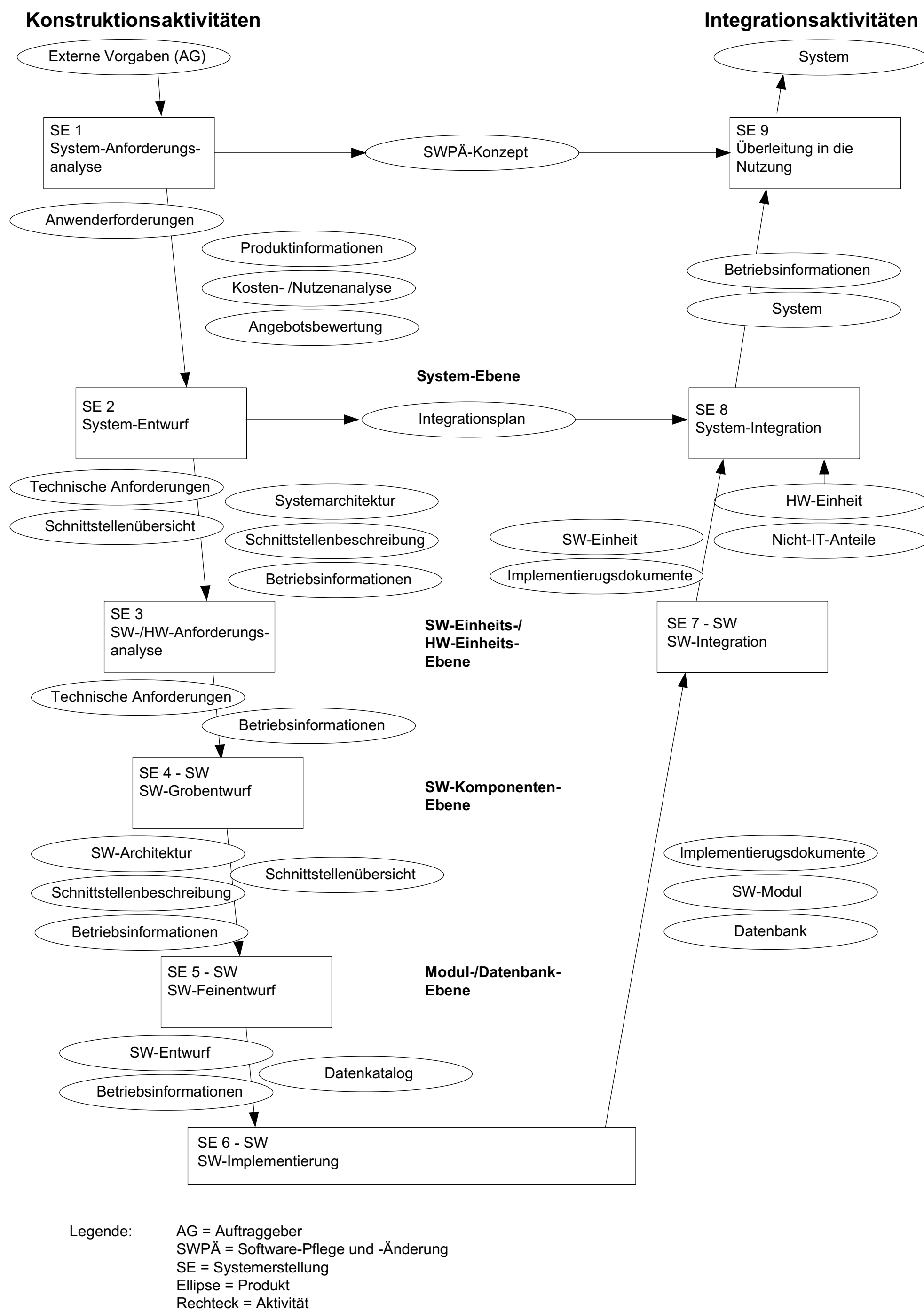

Abbildung 2.6: Teilmodell SE (BWB IT 15, 1997, S. 4-50, Balzert, 1998, S. 108)

Für jedes Teilmodell existiert eine Beschreibung, die festlegt, welche Aktivitäten in welcher Reihenfolge auszuführen und welche Ergebnisse (Produkte) dabei zu erarbeiten sind bzw. welche Produkte vor der Inangriffnahme einer Aktivität vorliegen müssen (einen Ausschnitt davon zeigt Abbildung 2.6). Auf der Ebene der Aktivitäten wird ebenso verfahren. Das heißt, die einzelnen Aktivitäten werden durch eine weitere Zerlegung präzisiert. Ein Beispiel zeigt Abbildung 2.7.

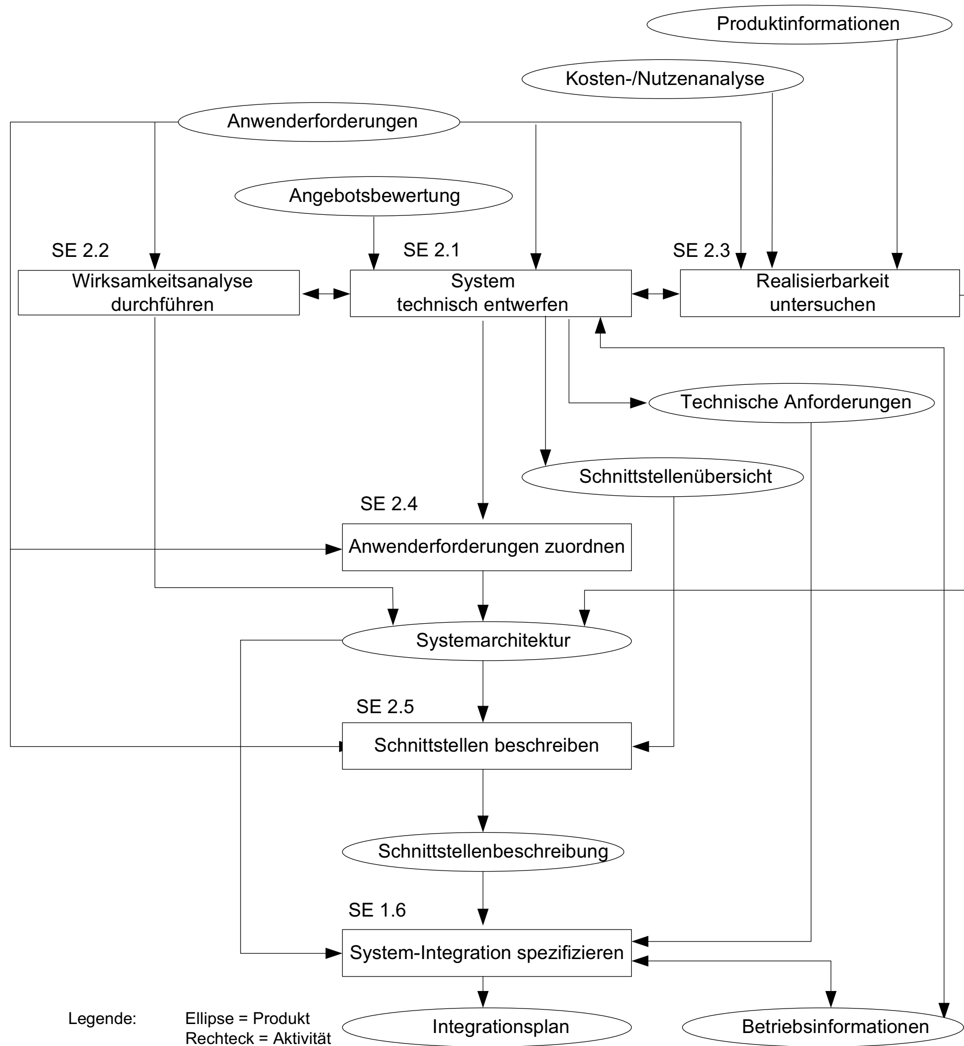

Abbildung 2.7: Aktivitätenzerlegung von SE 2 „System-Entwurf“ (BWB IT 15, 1997, S. 4–12, Balzert, 1998, S. 112)

Die vier Teilmodelle (SE, QS, KM, PM) sind miteinander verknüpft und beeinflussen sich über den Austausch von Produkten (siehe Abbildung 2.8). Das prinzipielle Zusammenwirken der Teilmodelle zeigt Abbildung 2.9.

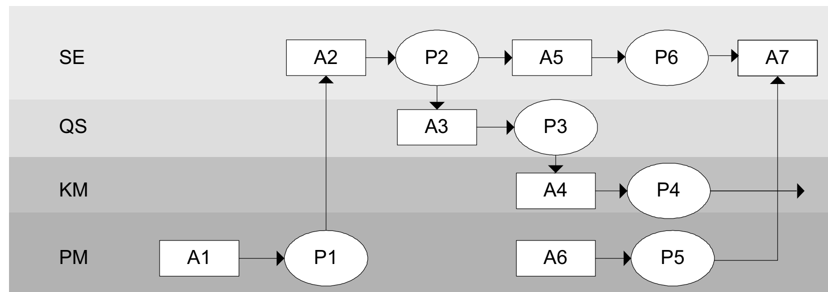

Abbildung 2.8: Schematische Darstellung eines Aktivitäten- und Produktflusses über die vier Teilmodelle SE, QS, KM, PM
Ai = Aktivitäten, Pi = Produkte
(Hansen und Neumann, 2001, S. 217)

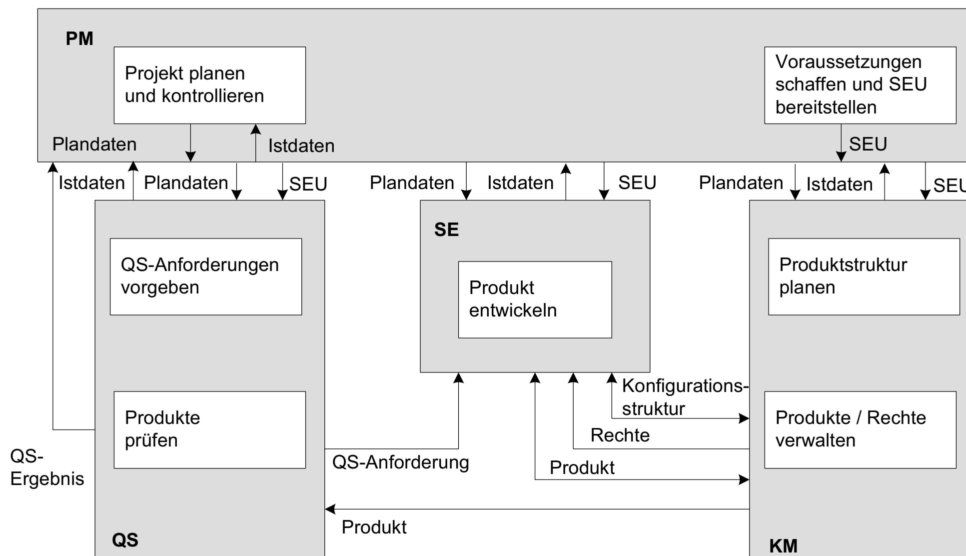

Abbildung 2.9: Zusammenwirken der Teilmodelle SE, QS, KM, PM
(BWB IT 15, 1997, S. 2–9, Balzert, 1998, S. 108)

	Manager	Verantwortliche	Durchführende
Submodell PM	Projektmanager	Projektleiter Rechtsverantwortlicher Controller	Projektadministrator
Submodell SE	Projektmanager IT-Beauftrager Anwender	Projektleiter	Systemanalytiker Systemdesigner SW-Entwickler HW-Entwickler Technischer Autor SEU-Betreuer Datenadministrator IT-Sicherheitsbeauftragter Datenschutzbeauftragter Systembetreuer
Submodell QS	Q-Manager	QS-Verantwortlicher	Prüfer
Submodell KM	KM-Manager	KM-Verantwortlicher	KM-Administrator

Abbildung 2.10: Rollen im V-Modell (BWB IT 15, 1997, S. R-2, Balzert, 1998, S. 109)

Im Rahmen des V-Modells werden auch Rollen festgelegt, aus denen die erforderliche Qualifikation zur Durchführung von Aktivitäten hervorgeht (siehe dazu Abbildungen 2.10 und 2.11).

Das V-Modell erhebt den Anspruch auf Allgemeingültigkeit. Das heißt, dass es unabhängig vom Anwendungsgebiet, von der Projektgröße und von der Art und Komplexität des zu entwickelnden Produkts angewandt werden kann. Eine Anpassung an spezifische Rahmenbedingungen und konkrete Entwicklungssituationen ist dennoch oft sinnvoll bzw. unvermeidbar, insbesondere wegen des hohen Detaillierungsgrades, der dieses Prozessmodell kennzeichnet. Die Möglichkeit, projektspezifische Modellanpassungen vorzunehmen, ist daher explizit vorgesehen. Die Anpassung geschieht durch „Maßschneiderung“ (Tailoring) von Produkten und/oder Aktivitäten. Für den Anpassungsprozess sind zwei Stufen vorgesehen:

1. Ausschreibungsrelevante Anpassungen, die vor dem Beginn des Entwicklungsprozesses vorgenommen werden;
2. technische Anpassungen, die während des Entwicklungsprozesses vorgenommen werden.

Vorteile

Die für das sequentielle Phasenmodell in Abschnitt 2.1. angeführten Vorteile gelten auch für das V-Modell. Darüber hinaus ist durch die Integration und sehr detaillierte Darstellung der vier Teilmodelle Systemerstellung, Qualitätssicherung, Konfigurations- und Projektmanagement eine wesentliche Verbesserung der klassischen phasenorientierten Prozessmodelle erreicht worden.

Aktivitäten	Rolle	Systemanalytiker	Systemdesigner	SW-Entwickler	HW-Entwickler	Technischer Autor	Systembetreuer	SEU-Betreuer	Datenadministrator	Datenschutzbeauftragter	IT-Sicherheitsbeauftragter	IT-Beauftragter	Anwender	Projektleiter
SE 1.1 Ist-Aufnahme/-Analyse durchführen		v	b										m	
SE 1.2 Anwendungssystem beschreiben		v											m	
SE 1.3 Kritikalität und Anforderungen an die Qualität definieren		v											m	
SE 1.4 Randbedingungen definieren		v											m	
SE 1.5 System fachlich strukturieren		v				m	m		m	b	m	m	m	
SE 1.6 Bedrohung und Risiko analysieren											v			
SE 1.7 Forderungscontrolling durchführen		m	m										v	m
SE 1.8 Software-Pflege und Änderungs-Konzept erstellen			v										m	m
SE 2.1 System technisch entwerfern			v			m						m		m
SE 2.2 Wirksamkeitsanalyse durchführen											v			
SE 2.3 Realisierbarkeit untersuchen		m	v								m	m		m
SE 2.4 Anwenderforderungen zuordnen			v											
SE 2.5 Schnittstelle beschreiben			v											
SE 2.6 System-Integration spezifizieren			v			m								
SE 3.1 Allgemeine Anforderungen an die SW-/HW-Einheit definieren			v											

Legende: v = verantwortlich m = mitwirkend b = beratend

Abbildung 2.11: Aktivitäten/Rollen-Matrix, Ausschnitt (BWB IT 15, 1997, S. R-14)

Nachteile

Auch die Nachteile betreffend gilt, dass die Nachteile des sequentiellen Phasenmodells, wie in Abschnitt 2.1 beschrieben, durch das V-Modell nicht in einem signifikanten Ausmaß beseitigt werden. Darüber hinaus ist dieses Prozessmodell nach Ansicht der Autoren zu aufgebläht, zu direktiv und zu wenig methodenneutral. Dies führt in der Regel zu einem Verlust an Flexibilität, zu beträchtlichem (bürokratischen) Mehraufwand, ohne dass dem ein signifikanter Nutzen gegenübersteht.

2.3 Das Prototyping-orientierte Prozessmodell

Die beiden in den vorangehenden Abschnitten beschriebenen Prozessmodelle beruhen auf der realitätsfernen Annahme, dass der Entwicklungsprozess in der Regel sequentiell durchgeführt werden kann und Iterationen über mehrere Phasen nur in Ausnahmefällen erforderlich sind. Die strenge Anwendung dieser Prozessmodelle fordert, dass eine Phase erst dann begonnen werden darf, wenn die vorhergehende abgeschlossen ist, das heißt, die vorgeschriebenen Ergebnisse (Zwischenprodukte) vollständig vorliegen. Die Praxis zeigt jedoch, dass es in den seltensten Fällen auf Anhieb gelingt, zum Beispiel eine vollständige und eindeutige Systemspezifikation (Pflichtenheft) oder einen fehlerfreien Entwurf der Systemarchitektur zustande zu bringen. Meist sind die Auftraggeber für die Entwicklung eines Software-Produktes nicht in der Lage, die Anforderungen an das gewünschte Produkt vollständig und verständlich zu formulieren, und das Anforderungsspektrum ändert sich oft kontinuierlich. Darüber hinaus gibt es für die meisten Anforderungen unterschiedliche Realisierungsmöglichkeiten. Die Entscheidung, welcher dieser Realisierungsmöglichkeiten der Vorzug zu geben ist, kann oft nicht auf analytischem Wege, sondern nur gestützt auf Experimente herbeigeführt werden.

Prototyping-orientierte Prozessmodelle sollen helfen, solche Probleme zu lösen oder zumindest zu entschärfen. Bevor wir Prototyping-orientierte Prozessmodelle beschreiben, erläutern wir noch einige wichtige Begriffe, die im Zusammenhang mit Prototyping von Bedeutung sind.

2.3.1 Begriffe und Abgrenzung

Seit vielen Jahren finden wir in der Literatur Berichte über Software-Projekte, in denen Prototypen zur Klärung von Benutzeranforderungen und von Entwicklungsproblemen eingesetzt werden. Eine Analyse der Publikationen zeigt, dass durchaus keine Einigkeit darüber besteht, was man unter „Prototyp“ und „Prototyping“ im Software Engineering versteht.

Einigkeit herrscht darüber, dass Software-Prototypen ausführbar sein müssen. Ob diese Ausführbarkeit direkt gegeben sein muss oder durch Simulation erreicht werden kann, ist bereits kontrovers. Nahezu Einigkeit herrscht auch darüber, dass Software-Prototypen – im Gegensatz zu anderen Prototypen, zum Beispiel in der Maschinenindustrie – schnell und billig realisiert werden müssen. Die einen schlagen vor, echte Prototypen im Sinne von Mustern zu entwickeln, die alle wesentlichen Eigenschaften eines geplanten Produkts aufweisen, und diese dann als Spezifikation für den eigentlichen Produktentwicklungsprozess zu verwenden – also herkömmliches Prototyping, wie in anderen technischen Disziplinen. Andere wiederum sind der Meinung, Software-Prototyping sei ein inkrementeller Entwicklungsprozess: Man implementiert schnell einige wenige, von vornherein klare Basisfunktionen, lässt diese durch den Benutzer erproben und beurteilen, verbessert sie und implementiert weitere Benutzeranforderungen, bis das Produkt fertig ist – also evolutionäre Software-Entwicklung.

Vielfach wird hervorgehoben, dass Prototypen wichtige Bestandteile der Software-Entwicklung im Hinblick auf die Qualitätssicherung und Risikominderung sind und dass ökonomisches Prototyping nicht ohne entsprechende Werkzeuge möglich ist.

Da es keine allgemein anerkannte Definition der Begriffe gibt, ist es notwendig, festzulegen, was wir in diesem Buch unter Prototyp und Prototyping verstehen. In Anlehnung an Boar, 1984, und Connell und Shafer, 1989, legen wir die Begriffe folgendermaßen fest:

> *Ein Software-Prototyp ist ein – mit wesentlich geringerem Aufwand als das geplante Produkt hergestelltes – einfach zu änderndes und zu erweiterndes ausführbares Modell des geplanten Software-Produkts oder eines Teiles davon, das nicht notwendigerweise alle Eigenschaften des Zielsystems aufweisen muss, jedoch so geartet ist, dass vor der eigentlichen Systemimplementierung der Anwender die wesentlichen Systemeigenschaften erproben kann. Prototyping umfasst alle Tätigkeiten, die zur Herstellung solcher Prototypen notwendig sind.*

Prototyping verfolgt verschiedene Ziele. Dies hat zur Folge, dass sowohl zwischen verschiedenen Arten des Prototyping als auch verschiedenen Arten von Prototypen unterschieden werden muss.

In der Literatur findet man häufig folgende Einteilung für das Prototyping (vgl. Floyd, 1984):

- Exploratives Prototyping
- Experimentelles Prototyping
- Evolutionäres Prototyping

Exploratives Prototyping

Das Ziel ist eine möglichst vollständige Systemspezifikation. Der Zweck besteht darin, den Entwicklern einen Einblick in den Anwendungsbereich zu ermöglichen, mit den Anwendern verschiedene Lösungsansätze zu diskutieren und die Realisierbarkeit des geplanten Systems in einem gegebenen organisatorischen Umfeld abzuklären. Vorgangsweise: Ausgehend von ersten Vorstellungen über das geplante System wird ein Prototyp entwickelt, der es erlaubt, diese Vorstellungen anhand von Anwendungsbeispielen zu prüfen und die erwünschte Funktionalität zu ermitteln. Maßgeblich dabei ist nicht die Qualität des Prototyps, sondern seine Funktionalität, die leichte Veränderbarkeit und auch die Kürze der Entwicklungszeit. Die Realisierung des Prototyps wird von Anwendern und Entwicklern gemeinsam vorgenommen. Das explorative Prototyping ist eine Technik zur Unterstützung der Problemanalyse und der Systemspezifikation.

Experimentelles Prototyping

Das Ziel ist eine vollständige Spezifikation von Teilsystemen als Grundlage für die Implementierung. Der Zweck besteht darin, die Tauglichkeit der Teilspezifikationen, von Architekturmodellen und von Lösungsideen für einzelne Systemkomponenten experi-

mentell nachzuweisen. Vorgangsweise: Ausgehend von ersten Vorstellungen über die Zerlegung des Systems wird ein Prototyp entwickelt, der es erlaubt, die Wechselwirkungen zwischen den Systemkomponenten zu simulieren, anhand von Anwendungsbeispielen die Schnittstellen der einzelnen Systemkomponenten und die Flexibilität der Systemzerlegung im Hinblick auf Erweiterungen zu erproben. Für die Qualtität der Prototypen gilt Gleiches wie beim explorativen Prototyping. Realisiert werden die Prototypen für die Durchführung von Experimenten mit Architekturmodellen in der Hauptsache von den Entwicklern. Das experimentelle Prototyping ist eine Technik zur Unterstützung beim System- und Komponentenentwurf.

Evolutionäres Prototyping

Das Ziel ist eine inkrementelle Systementwicklung, das heißt eine schrittweise aufbauende Entwicklung nach folgender Vorgangsweise: Entwicklung eines Prototyps für die von vornherein klaren Benutzeranforderungen. Das Ergebnis dient als Basissystem für den Anwender und für den nächsten Schritt, in dem neue Benutzeranforderungen integriert werden und der Prozess von Neuem beginnt. Damit verliert Systementwicklung den Charakter eines abgeschlossenen Projekts und wird ein Prozess, der die Anwendung ständig begleitet. Das ist leichter gesagt als getan. Dadurch wird zwar das Spezifikationsproblem zum Teil entschärft, aber das Problem des Systemdesigns keineswegs gelöst, weil eine unvollständige Spezifikation gerade der Grund dafür sein kann, dass die Brauchbarkeit einer Zerlegungsstruktur nicht beurteilt werden kann. Bei dieser Vorgangsweise lässt sich eigentlich nicht mehr zwischen Prototyp und Produkt unterscheiden, doch ist die Bezeichnung Prototyping angebracht, weil die ersten Versionen sicher nicht als praktisch einsetzbares Produkt zu sehen sind. Beim evolutionären Prototyping werden die Prototypen nicht simuliert und in der Regel auch nicht weggeworfen, sondern Schritt für Schritt zum Produkt ausgebaut.

Hinsichtlich der Arten von Prototypen unterscheiden wir:

- vollständige und unvollständige Prototypen
- Wegwerfprototypen und wiederverwendbare Prototypen
- horizontale und vertikale Prototypen
- Demonstrationsprototypen, Labormuster, Pilotsysteme

Unter einem *vollständigen Prototyp* soll ein Prototyp im herkömmlichen Sinne verstanden werden, in dem alle wesentlichen Funktionen des geplanten Systems vollständig verfügbar sind. Die bei der Herstellung und beim Einsatz gemachten Erfahrungen und der Prototyp selbst bilden die Grundlage für die endgültige Systemspezifikation. Vollständige Prototypen werden für Software-Systeme kaum hergestellt.

Unter einem *unvollständigen Prototyp* soll Software verstanden werden, die es gestattet, die Brauchbarkeit und Machbarkeit einzelner Aspekte (zum Beispiel Benutzungsschnittstelle, Systemarchitektur, Systemkomponenten) des geplanten Systems zu untersuchen.

Von einem *Wegwerfprototypen* spricht man dann, wenn seine Implementierung bei der Implementierung des Zielsystems nicht weiterverwendet wird, sondern der Prototyp „nur“ als ablauffähiges Modell dient.

Von *wiederverwendbaren Prototypen* spricht man, wenn wesentliche Teile des Prototyps bei der Implementierung des Zielsystems übernommen werden können.

Ein *horizontaler Prototyp* realisiert nur spezifische Aspekte eines Software-Produkts wie zum Beispiel die Benutzungsschnittstelle oder die Datenhaltung, ohne die dahinter liegende Funktionalität vollständig zur Verfügung zu stellen.

Ein *vertikaler Prototyp* implementiert einzelne Komponenten des geplanten Produkts vollständig, um zum Beispiel die Erfüllung von Performance-Anforderungen oder die technische Machbarkeit experimentell abzuklären.

Demonstrationsprototypen dienen als Akquisitionsinstrumente, um potenziellen Auftraggebern die wichtigsten Produkteigenschaften auf anschauliche Weise demonstrieren zu können.

Labormuster werden verwendet, um konstruktionsbezogene Aspekte zu evaluieren und zu demonstrieren, zum Beispiel die Beschaffenheit und Tauglichkeit von Architekturmustern.

Unter einem *Pilotsystem* verstehen wir ein Software-System, das bereits einen gewissen Reifegrad (Funktionsumfang, Stabilität) erreicht hat und als Basis für eine Produktentwicklung herangezogen wird.

2.3.2 Prozessmodell

Ansätze der Idee, Software-Produkte Prototyping-orientiert zu entwickeln, sind bereits im Wasserfall-Modell enthalten. Eine Prototyping-orientierte Software-Entwicklungsstrategie unterscheidet sich nicht grundsätzlich von der klassischen phasenorientierten Entwicklungsstrategie. Die beiden Strategien sind mehr komplementär als alternativ. Wesentlich ist, dass das Phasenmodell nicht als lineares, sondern als iteratives Modell anzusehen ist, und dass angegeben wird, wo und in welcher Weise diese Iteration nicht nur möglich, sondern sogar notwendig ist. Das modifizierte Modell zeigt Abbildung 2.12.

Die Prototyping-orientierte Entwicklungsmethode (wie die Autoren sie sehen) unterscheidet sich von der konventionellen Entwicklungsmethode nach dem sequenziellen Phasenmodell vor allem durch die Vorgangsweise und die Ergebnisse in den einzelnen Phasen. Die Phaseneinteilung bleibt zwar erhalten, aber mit dem Unterschied, dass Problemanalyse und Spezifikation zeitlich stark überlappt ablaufen und Entwurf, Implementierung und Test stark ineinander verschmelzen. Die Phasen sind somit nicht mehr Abschnitte einer stetigen Entwicklung. Wir sprechen deshalb nicht mehr von Phasen, sondern von Aktivitäten, weil es keine vollständige Trennung der Teilaufgaben mehr gibt, wie dies nach dem klassischen sequenziellen Phasenmodell (siehe Abschnitt 2.1) gefordert wird.

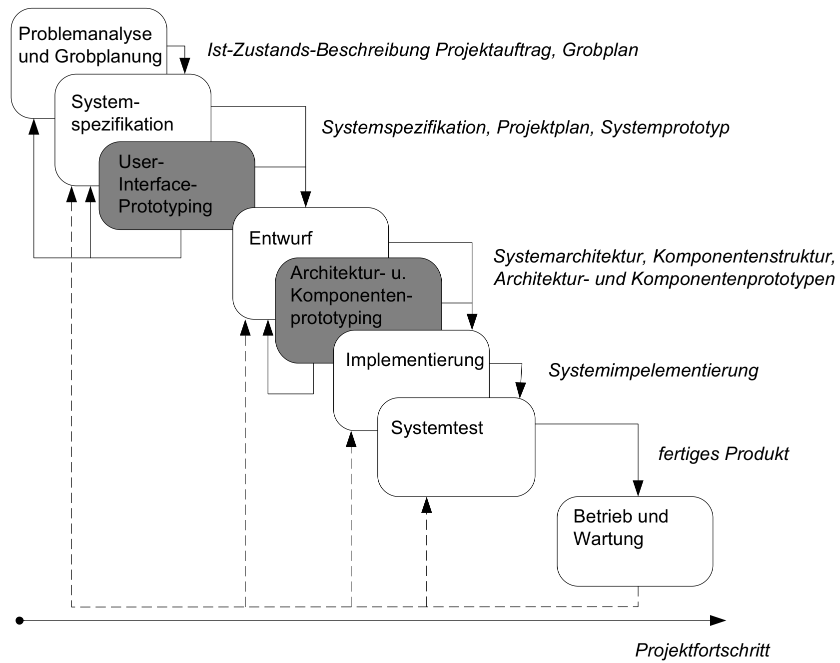

Abbildung 2.12: Prototyping-orientiertes Prozessmodell

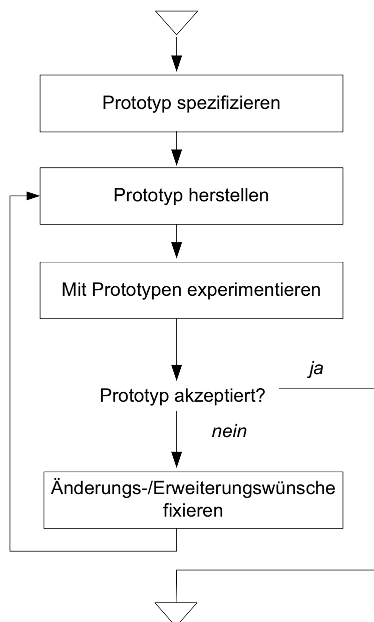

Abbildung 2.13: Prototyping-Aktivitäten

Die Prototyp-Herstellung ist ein iterativer Prozess (siehe Abbildung 2.13*)*. Zuerst wird ein Prototyp aufgrund von Resultaten vorausgegangener Aktivitäten (vorzugsweise werkzeugunterstützt) produziert. Während der Spezifikationsphase wird ein Prototyp der Benutzungsschnittstelle (hinter der sich wesentliche Teile der funktionalen Anforderungen an das geplante Software-System verbergen) entwickelt. Anhand dieses Prototyps wird durch Experimente, die den realen Einsatzbedingungen entsprechen, untersucht, ob die Anforderungen des Anwenders erfüllt werden. Software-Entwickler und Anwender können unter realitätsähnlichen Bedingungen ausprobieren, ob das Systemmodell Fehler hat, ob es die Vorstellungen der Anwender erfüllt oder ob Änderungen notwendig sind. Damit wird das Risiko einer falschen oder unvollständigen Systemspezifikation vermindert und eine wesentlich bessere Ausgangsbasis für die folgenden Aktivitäten geschaffen. Es wird also ein Zyklus im Phasenmodell eingeführt, um zu einer verbesserten Systemspezifikation zu kommen.

Gleiches gilt für die nächsten Aktivitäten: Nachdem der Entwurf der Systemarchitektur eine bestimmte Komplexität erreicht hat, wird anhand eines Architekturprototyps (noch vor Beginn der Implementierungsphase) überprüft, ob die Entwurfsentscheidungen zielführend sind. Also auch hier wird wiederum ein Zyklus im Gesamtmodell eingeführt. Damit kann die Güte der gewählten Systemzerlegung, die Vollständigkeit und Richtigkeit der Komponentenschnittstellen und die Erweiterbarkeit der Systemarchitektur unter realitätsähnlichen Bedingungen experimentell geprüft werden, wodurch das Risiko, während der Implementierung umfangreiche und teurere Modifikationen durchführen zu müssen, wiederum gesenkt wird.

Aus dem oben Gesagten folgt ein weiterer wesentlicher Unterschied zum sequenziellen Phasenmodell: Bei diesem wird so spät wie möglich implementiert, erst wenn alle Einzelheiten des Spezifikations- und Entwurfsprozesses geklärt sind. Bei Anwendung der Prototyping-orientierten Entwicklungsmethode dagegen wird so früh wie möglich (ein Prototyp) implementiert. Dies deshalb, weil die Praxis zeigt, dass man viel eher zum Ziel kommt, wenn man die Systemspezifikation und die Systemarchitektur schrittweise anhand eines Modells entwickelt, das es gestattet, auch das dynamische Verhalten des Systems darzustellen, also durch ausführbare Prototypen und nicht bloß durch Beschreibungen.

Durch die Überlappung der einzelnen Aktivitäten (siehe Abbildung 2.12) und durch die Art der Ergebnisse – die nicht mehr bloß Beschreibungen, sondern lauffähige Prototypen sind – wird das Risiko, eine Fehlentscheidung zu treffen, wesentlich verringert, und die Effekte, die mit den Zwischenergebnissen durch Experimente erzielt werden können, erleichtern die Qualitätssicherung.

Für die allgemeine Vorgangsweise bei der Prototyping-orientierten Software-Entwicklung ist es unerheblich, ob die einzelnen Prototypen (für die Benutzungsschnittstelle, die Systemarchitektur oder einzelne Komponenten) Wegwerfprototypen oder wiederverwendbare Prototypen sind. Das Ziel ist eine Risikominderung, eine erfolgreiche Qualitätssicherung und die Ausnutzung des Lerneffekts durch Experimente unter realen Bedingungen. Im Hinblick auf eine Verminderung der Kosten für den Herstellungs-

prozess ist es natürlich wünschenswert, dass wiederverwendbare Prototypen entstehen, die zu einer evolutionären Entwicklungsstrategie führen. Inwieweit dieses Ziel erreichbar ist, hängt in erster Linie von der Qualität der verwendeten Prototyping-Werkzeuge ab.

Vorteile

- Das Prototyping-orientierte Prozessmodell hat seine Stärken dort, wo das sequenzielle phasenorientierte Prozessmodell Schwächen aufweist; die Stärken des sequenziellen phasenorientierten Prozessmodells bleiben aber erhalten.
- Die projektbegleitende Qualitätssicherung wird gefördert. Spezifikations- und Entwurfsfehler können früh erkannt werden, dies senkt die Fehlerbehebungskosten und verbessert die Gebrauchstauglichkeit und damit die Produktakzeptanz.
- Prototyping-orientierte Software-Entwicklung führt in der Regel zu einer Senkung der Produktlebenszyklus-Kosten, weil durch Qualitätssteigerung insbesondere die Wartungs- und Betriebskosten gesenkt werden.

Nachteile

- Die Anwendung des Prototyping-orientierten Prozessmodells erfordert die Verfügbarkeit spezieller (Prototyping-)Werkzeuge.
- Es besteht die Gefahr, dass die Produktspezifikation zu einem nichtkonvergenten Prozess ausartet („the more they get, the more they want“) und damit die Entwicklungskosten steigen.
- Es besteht die Gefahr der Vernachlässigung von Software-technischen Prinzipien, wenn Prototypen oder Komponenten von Prototypen, deren Software-technische Qualität nicht ausreichend ist, in der Produkt-Implementierung verwendet werden.

2.4 Das Spiralmodell

Boehm hat 1988 ein Prozessmodell vorgeschlagen, das die bisher vorgestellten Modelle kombiniert bzw. es gestattet, sie in dieses Modell einzubetten. Beim Spiralmodell handelt es sich daher um ein Metamodell, das die Möglichkeit bietet, die für ein Software-Entwicklungsprojekt am besten geeignete Vorgehensweise zu wählen.

2.4.1 Das Spiralmodell von Boehm

Der Entwicklungsprozess wird nach Boehm in vier Schritte gegliedert, die im Sinne einer evolutionären Entwicklungsstrategie mehrmals zyklisch durchlaufen werden (siehe Abbildung 2.14).

Die radiale Ausdehnung in Abbildung 2.14 stellt den Gesamtaufwand, der bis zu einem bestimmten Zeitpunkt geleistet wurde, dar, und die Winkeldimension bezieht sich auf den Projektfortschritt in den einzelnen Spiralenzyklen. Jeder Zyklus umfasst die gleiche Schrittfolge für jeden Teil des zu entwickelnden Produkts und für jede Ausarbeitungsstufe.

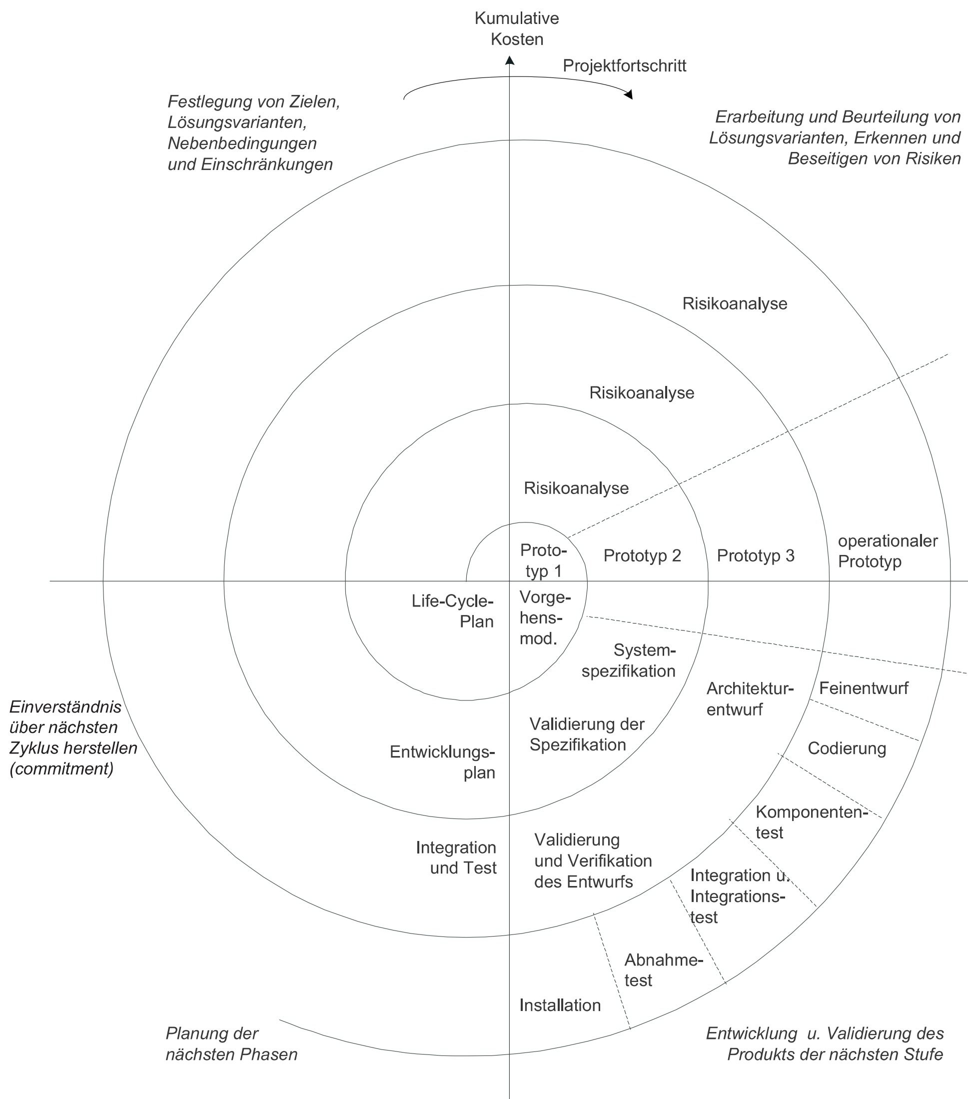

Abbildung 2.14: Das Spiralmodell von Boehm, 1988

Jeder Spiralenzyklus beginnt mit der Festlegung folgender Punkte (Schritt 1):

- Ziele für und Anforderungen an das (Teil-)Produkt (Einsatzbereich, Funktionalität, Modifizierbarkeit etc.)
- Alternativen zur Realisierung des (Teil-)Produkts (Lösungsvariante A, Lösungsvariante B, Wiederverwendung, Zukauf etc.)
- Nebenbedingungen und Einschränkungen (Kosten, Termine, Schnittstellen etc.)

Im nächsten Schritt (Schritt 2) wird eine Beurteilung der vorgeschlagenen Lösungsvarianten im Hinblick auf die Projektziele und gegebenen Nebenbedingungen vorgenommen. Dabei geht es vor allem darum, mögliche Risikoquellen und Unklarheiten aufzudecken. Werden solche gefunden, schließt sich daran die Überlegung von Maßnahmen und Strategien zur Ausschaltung von Risikoquellen und ihrer Auswirkung an. Dabei soll vor allem Prototyping eingesetzt werden. Falls zum Beispiel Unklarheit über den Leistungsumfang, die Gestaltung der Benutzungsschnittstelle oder das Datenmodell für eine Datenbank bestehen, sollen Prototypen helfen, diese Aspekte näher abzuklären.

Im Schritt 3 wird unter Berücksichtigung der verbleibenden Risiken das Prozessmodell für diesen Schritt festgelegt (dies kann auch eine Kombination verschiedener Prozessmodelle sein) und es werden die daraus resultierenden Aktivitäten durchgeführt.

Basierend auf den Ergebnissen einer Validierung als abschließende Aktivität in Schritt 3 wird in Schritt 4 der nächste Zyklus einschließlich der dazu benötigten Ressourcen geplant, und es muss ein Einverständnis über den nächsten Zyklus herbeigeführt werden.

Diese Vorgangsweise setzt sich im Uhrzeigersinn fort.

Ein wichtiger Aspekt des Spiralmodells ist, dass jeder Zyklus mit einem Validierungsschritt versehen ist, in dem alle am Projekt beteiligten Personen und davon betroffene Anwender oder Organisationseinheiten mit einbezogen werden. Die Validierung umfasst alle Produkte, die während des Zyklus entstanden sind, darunter auch die Planung für den nächsten Zyklus sowie die dafür notwendigen Ressourcen. Das Hauptziel sieht Boehm darin, dass sich alle Beteiligten über das bisher Geleistete und das im nächsten Zyklus Geplante einig sind. Die Zyklusplanung kann selbstverständlich zu einer Aufteilung des Projekts in Teilprojekte führen, die dann in eine Reihe von parallel durchzuführenden Spiralenzyklen münden.

Es bleibt offen, wann die Spirale endet und wie die Phase der Wartung in dieses Vorgehensmodell eingebettet ist. Es wird davon ausgegangen, dass das Spiralmodell gleich gut sowohl auf die Entwicklung als auch auf die Wartung von Software-Produkten angewandt werden kann. In beiden Fällen beginnt man mit der Annahme, dass ein bestimmtes Aufgabenfeld durch den Einsatz oder die Anpassung von Software-Produkten sinnvoll bewältigt werden kann. Der (spiralförmige) Entwicklungsprozess dient der Prüfung dieser Hypothese. Die Spirale endet, wenn die Hypothese widerlegt ist oder der inkrementelle Entwicklungsprozess ein fertiges Produkt ergibt.

Das Spiralmodell eignet sich gut für die Einbettung von Qualitätssicherungsmaßnahmen in den Software-Entwicklungsprozess, orientiert sich wie das Prototyping-orientierte Prozessmodell an dem Ziel, möglichst früh Fehler zu erkennen und verschiedene Lösungsalternativen (mit vertretbarem Aufwand) zu erwägen und erfordert keine verschiedenen Ansätze für die Entwicklung und Wartung von Software-Systemen. Die Software-Entwicklung wird als kontinuierlicher Wartungsprozess angesehen. Dadurch verliert die Wartung ihren „zweitklassigen“ Status. Viele Probleme, die entstehen, wenn – wie häufig der Fall – Wartungs- und Erweiterungsaktivitäten mit wesentlich geringerer Systematik und Planung durchgeführt werden als die Neuentwicklung eines Software-Pro-

dukts, werden dadurch vermieden. Als nachteilig hat sich in der Praxis herausgestellt, dass zu Beginn eines Projektes Zeit- und Kostenpläne nur schwer zu erstellen sind und der Managementaufwand beträchtlich ansteigt.

2.4.2 Das Spiralmodell von Pomberger und Pree

Den in den vorhergehenden Abschnitten behandelten Prozessmodellen ist gemeinsam, dass sie nur auf den Software-Entwicklungsprozess ausgerichtet sind. Wenn man zur Unterstützung eines Geschäftsprozesses oder im Rahmen der Herstellung eines technischen Produktes Software benötigt, dann ist es allerdings sinnvoll, die Frage zu stellen, ob die benötigte Software überhaupt entwickelt werden muss oder ob nicht die Möglichkeit besteht, sie zuzukaufen, und wenn diese Möglichkeit besteht, ob dies nicht wirtschaftlicher als eine Neuentwicklung ist. Es ist also eine „make or buy"-Entscheidung erforderlich und es ist sinnvoll, die dazu erforderlichen Aktivitäten in das Prozessmodell zu integrieren.

In den bisher diskutierten Prozessmodellen ist nach Ansicht der Autoren der Aspekt der Wiederverwendung bereits existierender (vorhandener oder käuflicher) Komponenten nicht in ausreichendem Maße berücksichtigt. Das Spriralmodell von Boehm weist zudem die Schwäche auf, dass die Risikoanalyse der Prototypentwicklung vorgelagert ist, aber gerade im Zuge der Protypentwicklung signifikante Risikofaktoren identifiziert werden können, sodass es sinnvoll ist, diese beiden Aktivitäten zu vertauschen.

Zur Beseitigung dieser Nachteile haben die Autoren ein modifiziertes Spiralmodell entwickelt, das in Abbildung 2.15 dargestellt ist.

Dem Vorschlag von Boehm folgend wird auch hier der Entwicklungsprozess in vier Schritte gegliedert, die im Sinne einer evolutionären Entwicklungsstrategie mehrmals zyklisch durchlaufen werden.

Dieses Prozessmodell postuliert, dass jedes zu beschaffende Software-Produkt, unabhängig davon, ob der Auftraggeber über eine eigene Software-Entwicklungsgruppe verfügt, ausgeschrieben werden soll. Das hat einerseits den Vorteil, dass auch die firmeninternen Software-Entwickler ein Angebot abgeben und sich damit dem Wettbewerb stellen, ständig ihre Wettbewerbsfähigkeit unter Beweis stellen müssen und damit zu mehr Kosten-, Technologie- und Leistungsbewusstsein gezwungen werden und andererseits für leistungsfähige Anbieter die Marktsegmente größer werden.

Der Beschaffungsprozess beginnt also mit einer *Ausschreibung*. Diese erfolgt zweckmäßigerweise auf Basis eines formalisierten, generischen Ausschreibungsdokumentes, in dem der geforderte Leistungsumfang überblicksartig beschrieben wird. Durch ein generisches Ausschreibungsdokument kann sowohl der Ausschreibungsaufwand für den Auftraggeber als auch der Analyseaufwand für die Anbieter gering gehalten werden. Die Erfahrung zeigt, dass trotz reduzierten Ausschreibungs- und Kalkulationsaufwands die kalkulierten Aufwände nicht ungenauer sind als bei detaillierten Pflichtenheften.

Als Nächstes geht es darum, im Zuge der *Angebotsevaluation* die drei besten Anbieter zu selektieren. Die dem Auftraggeber vorliegenden Angebote werden, um die Zuverläs-

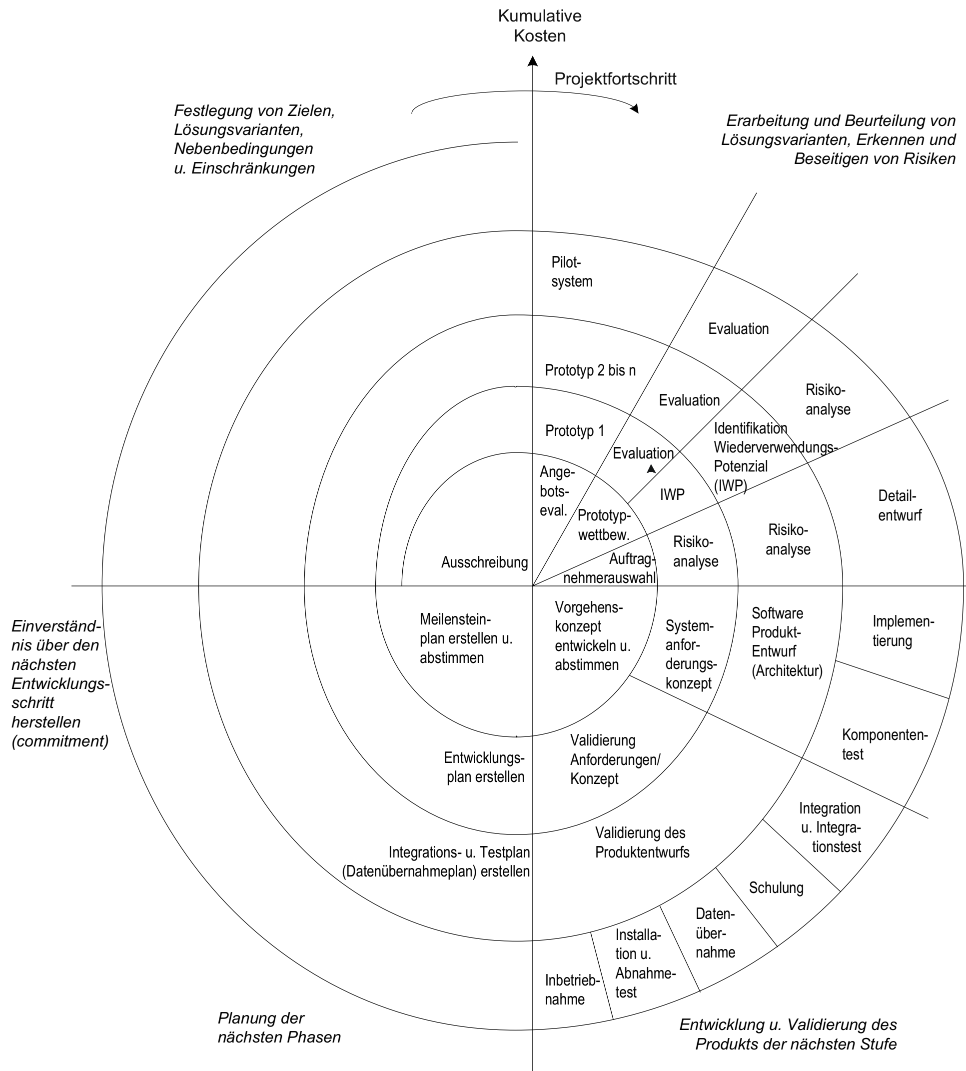

Abbildung 2.15: Das Spiral-Modell von Pomberger und Pree

sigkeit des Auswahlergebnisses zu erhöhen, vorzugsweise in einem Multiview-Verfahren von Auftraggeber-Seite und von einem oder mehreren externen Experten getrennt evaluiert. Die eingesetzte Evaluationsmethodik soll jedenfalls darauf abzielen, dass mit geringstmöglichem Aufwand und größtmöglicher Wahrscheinlichkeit anhand quantitativer und qualitativer Metriken die (drei) Bestbieter ermittelt werden können. Dazu empfiehlt es sich, zum Beispiel in einer gemeinsamen Sitzung (Joint Session), die einzelnen Evaluationsergebnisse offenzulegen und unterschiedliche Ergebnisse so lange methodengestützt zu hinterfragen, bis Konsens bezüglich der (drei) Bestbieter besteht.

Die Bestbieter werden zu einem *Prototyping-Wettbewerb* eingeladen. Das heißt, sie werden aufgefordert, für einen der Kernprozesse, die mit dem zu entwickelnden Software-Produkt unterstützt werden sollen, innerhalb eines vorgegebenen Zeitraums, der üblicherweise vier bis sechs Wochen beträgt, einen Prototyp zu entwickeln. Umfang und Ausprägung des Prototyps bestimmen die Wettbewerbsteilnehmer selbst, denn daraus lassen sich wichtige Schlüsse über die Leistungsfähigkeit der Wettbewerbskandidaten ziehen. Der Prototypingprozess wird wieder mittels eines Multiview-Evaluationsverfahrens beobachtet. Dabei soll von den Evaluatoren das Augenmerk auf die Organisiertheit, die Produktivität, die Innovationskraft und die Technologie-Kompetenz des von den Wettbewerbsteilnehmern eingesetzten Personals und den Leistungsumfang und die Qualität des erstellten Prototyps gelegt werden.

Der Auftraggeber erhält dadurch eine wesentlich fundiertere Entscheidungsgrundlage über die Leistungsfähigkeit der Anbieter, als üblicherweise aus den Angeboten zu ermitteln ist. Die Anbieter haben ihrerseits die Möglichkeit, sich über die tatsächliche Komplexität des geplanten Produktes und über die oft für den Erfolg oder Misserfolg entscheidenden Rahmenbedingungen beim Auftraggeber ein ausreichend genaues Bild zu verschaffen und können so ihre Aufwandskalkulation überprüfen. In einer gemeinsamen Sitzung werden die einzelnen Evaluationsergebnisse über den Prototypingprozess offen gelegt und diskutiert. Unterschiedliche Ergebnisse werden methodengestützt so lange hinterfragt, bis Konsens bezüglich der Reihung der Wettbewerbsteilnehmer besteht; diese wird vom Evaluationsteam dem Auftraggeber zur Entscheidung präsentiert.

Die im Wettbewerb ausgeschiedenen Anbieter erhalten üblicherweise für den im Prototyping-Wettbewerb getätigten Aufwand eine Pauschalabgeltung, deren Höhe bereits in der Ausschreibung den Anbietern bekannt gemacht wurde. Sollte sich während des Prototyping-Wettbewerbs herausgestellt haben, dass der Projektumfang von dem abweicht, der aus den Dokumenten der Ausschreibung und der Rücksprache mit dem Auftraggeber im Angebotslegungsprozess zu entnehmen war, wird vor der Auftragsvergabe die Preis- und Terminbildung erneut verhandelt.

Auf Basis der Ergebnisse aus dem Prototyping-Wettbewerb trifft der Auftraggeber die Entscheidung über die Auftragsvergabe (*Auftragnehmerauswahl*) und damit über „make or buy“. Zur Absicherung der partnerschaftlichen Zusammenarbeit zwischen Auftraggeber und Auftragnehmer wird empfohlen, dass gemeinsam von beiden Parteien ein externer Fachmann als *Projektkoordinator* nominiert wird. Der Projektkoordinator agiert auch als Konfliktmanager. Kommt der Auftragnehmer im Verlaufe der Projektabwicklung zur Auffassung, dass das Projektvolumen nicht mit dem übereinstimmt, das Basis für die Preisbildung war, und kann darüber, weil zum Zeitpunkt der Ausschreibung noch kein Pflichtenheft vorlag, kein Konsens mit den Auftraggeber hergestellt werden, entscheidet der externe Projektkoordinator, ob eine Preisänderung und/oder Terminänderung gerechtfertigt ist. Der externe Koordinator muss seine Entscheidung ausführlich begründen, Auftraggeber und Auftragnehmer verpflichten sich bei Vertragsunterzeichnung, die Entscheidungen des externen Koordinators rechtsverbindlich anzuerkennen.

Der weitere Ablauf des Entwicklungsprozesses erfolgt im Wesentlichen so, wie dies im Spiralmodell von Boehm (siehe oben) vorgesehen ist, jedoch mit dem Unterschied, dass in Schritt 2

- die Evaluierung von Alternativen erst nach den Prototyping-Aktivitäten erfolgt, weil dies den Evaluationsprozess sicherer und effizienter macht;
- eine in der Boehm'schen Modellvariante nicht vorgesehene Aktivität, nämlich die Identfikation von Wiederverwendungs-Potenzialen, eingeführt wird, um die „make-or-buy-" bzw. wiederverwendungsorientierte Produktentwicklung auch auf Komponentenebene zu fördern;
- die Risikoanalyse erst als letzte Aktivität im Schritt 2 (und nicht als erste wie bei Boehm) durchgeführt wird, weil auf Basis der Prototyp-Evaluationsergebnisse und der vorhandenen oder nicht vorhandenen Wiederverwendungs-Potenziale sich präzisere Risikoszenarien herleiten lassen.

Vorteile

- Das Prozessmodell von Pomberger und Pree integriert den „make-or-buy"-Entscheidungsprozess und den Prozess der Identifikation vorhandener Wiederverwendungs-Potenziale. Es ist daher umfassender als die meisten anderen Prozessmodelle.
- Wie beim Spiralmodell von Boehm handelt es sich bei diesem Prozessmodell um ein Metamodell. Es gestattet beispielsweise die Integration anderer Prozessmodelle als Spezialfälle. Dies steigert die Flexibilität, die Anpassungsmöglichkeiten an spezielle Rahmenbedingungen und die Akzeptanz.
- Wissenschaftliche Begleituntersuchungen der Autoren zur Anwendung dieses und anderer Prozessmodelle zeigen, dass durch diese Vorgehensweise das Risiko eines Scheiterns (Floprisiko) deutlich abnimmt; insbesondere weil durch den dem eigentlichen Entwicklungsprozess vorgeschalteten Prototyping-Wettbewerb sowohl die zu erwartende Problemkomplexität als auch die Tauglichkeit der zu ihrer Meisterung vorgesehenen Ressourcen (Personal, Technologie, verfügbare Komponenten, Werkzeuge etc.) wesentlich besser beurteilbar sind, als dies bei konventioneller Vorgehensweise möglich ist.
- Das Prozessmodell zielt darauf ab, dass Fehler und ungeeignete Lösungsansätze (nicht nur im technischen Bereich, sondern zum Beispiel auch in der Personalzusammensetzung oder in der Ausprägung des Qualitätssicherungsprozesses) möglichst früh erkannt werden. Die Organisation des Entwicklungsprozesses kann dadurch schnell und zielgerichtet angepasst werden. Dies führt zur Vermeidung unnötiger Kosten und zur Steigerung der Produktqualität.

Nachteile

- Wie alle Metamodelle hat auch dieses Modell den Nachteil, dass das richtige „Tailoring" in entscheidendem Ausmaß von den Fähigkeiten und Erfahrungen der Projektverantwortlichen abhängt; die als Vorteil angeführte Flexibilität kann sich bei falscher Ausnutzung schnell in einen Nachteil verwandeln.

- Einer der wichtigsten und innovativsten Aspekte dieses Prozessmodells, der Prototyping-Wettbewerb und die darauf beruhende „make-or-buy"-Entscheidung, ist ein revolutionärer Ansatz, der bei manchen Software-Entwicklern mit großer Skepsis gesehen wird, weil es in dieser Branche ungewohnt ist, sich vor einer Beauftragung einem Qualifikationswettbewerb stellen zu müssen. Das kann dazu führen, dass potenzielle Anbieter aus Angst, sich im Wettbewerb zu blamieren, ausscheiden.
- Die Abschlagszahlungen an die im Prototyping-Wettbewerb ausgeschiedenen Anbieter werden manchmal von Auftraggeberseite als verlorene Zusatzkosten gesehen und schmälern daher die Modellakzeptanz. Dies ist zwar falsch, weil – wie die Erfahrung zeigt – diese Prototyping-Wettbewerbe oft erschreckende Defizite auf Anbieterseite aufdecken und sich damit diese Investition durch die Senkung des Floprisikos für den Auftraggeber allemal lohnt.

2.5 Der Unified Process

Das Prozessmodell „Unified Process" wurde von der Firma Rational[1] entwickelt (das Modell wird deshalb häufig als „Rational Unified Process" oder kurz RUP bezeichnet) und baut auf Überlegungen von Ivar Jacobson zur Systematisierung des Software-Entwicklungsprozesses auf (siehe dazu Jacobson, 1996, Jacobson et al., 1999, Kruchten, 2000).

Das Prozessmodell (siehe Abbildung 2.16) postuliert im Wesentlichen

- sechs grundlegende Entwicklungsschritte: Geschäftsprozessmodellierung (*Business Modeling*), Anforderungsanalyse (*Requirements*), Analyse und Entwurf (*Analysis & Design*), Implementierung (*Implementation*), *Test* und Installation (*Deployment*);
- drei Unterstützungsprozesse: Konfigurations- und Änderungsmanagement (*Configuration & Change Management*), Projektmanagement (*Project Management*) und Umgebungsmanagement (*Environment*);
- vier Phasen: Anfangsphase (*Interception*), Ausarbeitungsphase (*Elaboration*), Konstruktionsphase (*Construction*) und Übergangsphase (*Transition*).

Abbildung 2.16 illustriert den Zusammenhang zwischen den Entwicklungsschritten und Unterstützungsprozessen und den vier Phasen. Die in Abbildung 2.16 grau hinterlegten Flächen symbolisieren die phasenbezogene Aufwandsverteilung eines jeden Entwicklungsschrittes bzw. Unterstützungsprozesses. Die gesamte Fläche stellt 100 % des Aufwandes je Entwicklungsschritt bzw. Unterstützungsprozess dar. Mit dieser Darstellung wird visualisiert, mit wie viel Aufwand in welcher Phase für einen Entwicklungsschritt zu rechnen ist. Alle Flächen gemeinsam ergeben 100 % des Projektaufwandes. Abbildung 2.16 zeigt, dass jede Phase mit einem Meilenstein endet: Die Interception-Phase

[1] Rational Software ist eine Tochterfirma von IBM (IBM Rational Software). Informationen dazu findet man unter www.rational.com.

mit LO (*Lifecycle Objective*), die Elaboration-Phase mit LA (*Lifecycle Architecture*), die Construction-Phase mit OC (*Initial Operational Capability*) und die Transition-Phase mit PR (*Product Release*).

Der Unified Process ist ein iterativer und inkrementeller Prozess. Nach anfänglichen Planungsschritten wird jeder Entwicklungsschritt iterativ mehrmals durchgeführt (siehe Abbildung 2.17). Im Zuge der wiederholten Ausführung der Entwicklungsschritte werden Umfang und Qualität der Arbeitsergebnisse (Produkte) schrittweise und kontinuierlich verbessert. Im Zuge einer Iteration, das heißt eines Entwicklungsschrittes, werden so viele Produkte wie möglich bzw. notwendig weiterentwickelt. Wenn alle gewünschten Produkte des geplanten Software-Systems in der erforderlichen Qualität vorliegen, endet der Iterationsprozess, und das Software-System kann ausgeliefert bzw. installiert werden. Die Abfolge der Iterationen ist in Phasen gegliedert und in jeder Phase können eine oder mehrere Iterationen durchlaufen werden (siehe dazu Abbildung 2.16). Die Anzahl erforderlicher Iterationen variiert von Projekt zu Projekt und ist abhängig von dessen Komplexität. In Kruchten, 2000 finden wir dazu die in Abbildung 2.18 angeführten Richtwerte.

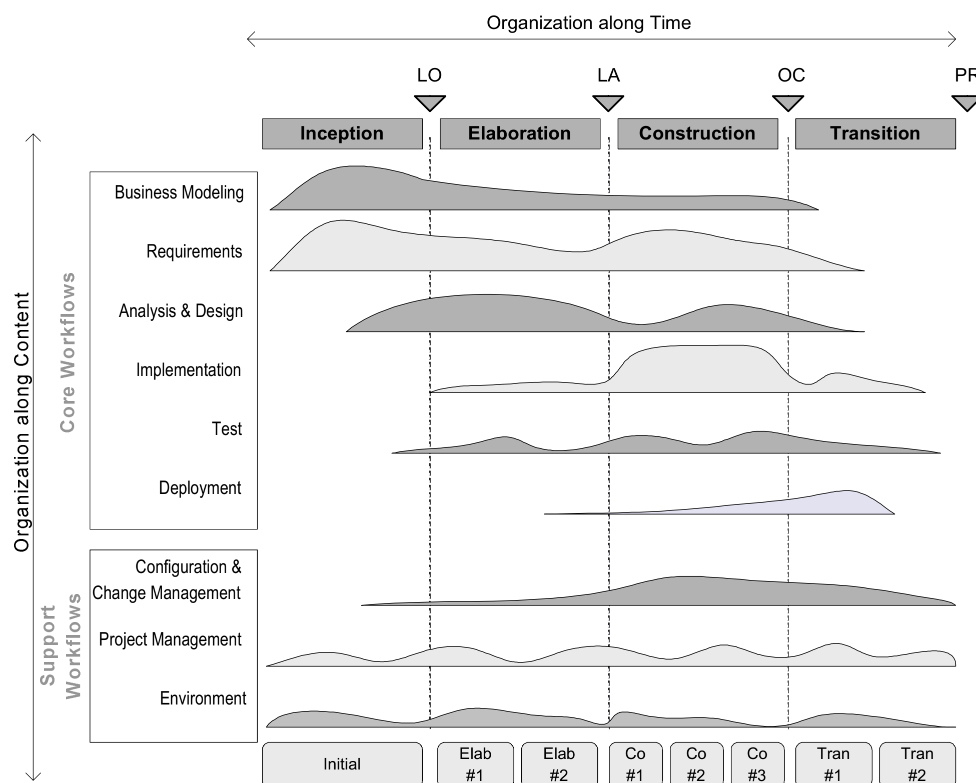

Abbildung 2.16: Der Rational Unified Process (Kruchten, 2000)

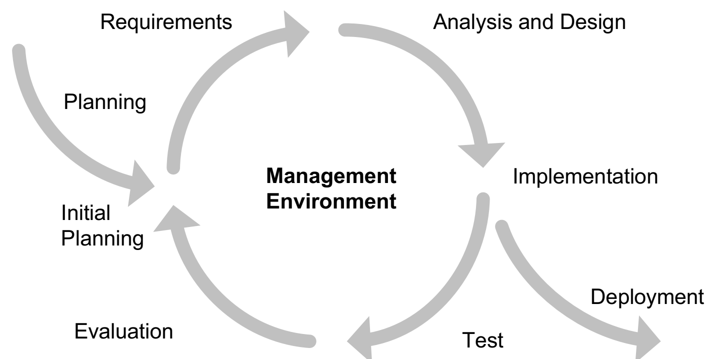

Abbildung 2.17: RUP-iterativer und inkrementeller Prozess (Kruchten, 2000)

Projekt-Komplexität	Iterationen Interception	Iterationen Elaboration	Iterationen Construction	Iterationen Transition	Summe der Iterationen
niedrig	0	1	1	1	3
normal	1	2	2	1	6
hoch	1	3	3	2	9

Abbildung 2.18: Anzahl von Iterationen in Abhängigkeit von der Projektkomplexität (Kruchten, 2000)

Das dem Unified Process zugrunde liegende iterative und inkrementelle Prozessmodell ist *anwendungsfall-* und *architekturzentriert*.

Bevor ein Software-Produkt entworfen und implementiert werden kann, muss Klarheit darüber herrschen, welche Anforderungen und Leistungsmerkmale das Produkt erfüllen bzw. aufweisen muss. Im RUP wird vorgeschlagen, die Anforderungen und Leistungsmerkmale in Form von typischen Anwendungsfällen (*Use Cases*) zu beschreiben. Sowohl beim Architekturentwurf als auch bei der Implementierung und beim Test müssen sich die Verantwortlichen darauf konzentrieren, die korrekte Umsetzung bzw. Erfüllung der in der Anforderungsdefinition beschriebenen Anwendungsfälle nachzuweisen. Die Anwendungsfälle sind somit das Bindeglied zwischen den Entwicklungsschritten. Der Entwicklungsprozess wird quasi von den Anwendungsfällen gesteuert.

Der Architekturentwurf beginnt nach dem RUP bereits in der ersten Phase des Entwicklungsprozesses (also früher als in anderen Prozessmodellen). Es werden zuerst jene Teile der Architektur entworfen, die unabhängig von den vorgegebenen Anwendungsfällen sind, zum Beispiel technologiebedingte Architekturteile, und danach konzentriert man sich beim Architekturentwurf auf die Umsetzung der Anwendungsfälle. Der Architekturentwurf erfolgt parallel zur Beschreibung der Anwendungsfälle, um sicherzustellen, dass nicht an den Anforderungen vorbei entworfen wird und um Fehlentscheidungen im Zuge der Architekturgestaltung frühzeitig zu erkennen. Im Sinne der inkrementellen Entwicklungsstrategie wird bei der Detaillierung und Vervollständigung der Anwendungsfälle auch die Systemarchitektur Schritt für Schritt ausgereifter.

2.6 Ein objektorientiertes Phasenmodell

Ein wesentliches Merkmal der objektorientierten Software-Entwicklung (siehe dazu Buchteil II – Konstruktions- und Architektur-orientierte Sicht) besteht darin, dass die Wiederverwendung von (Objekt-)Klassen, von Klassenbibliotheken und von Frameworks durch Komposition, Vererbung und Polymorphie unterstützt wird. Die vielfältigen Aspekte der Wiederverwendung – es können eigen- oder fremdentwickelte Architekturmodelle, Frameworks, Subsysteme, Klassenbibliotheken, Klassen etc. wiederverwendet werden – müssen im Prozessmodell berücksichtigt werden. Der Rational Unified Process ist zwar im Zuge der Entwicklung des objektorientierten Programmierparadigmas entstanden, dies schlägt sich aber primär auf der Ebene der empfohlenen Methoden zu den einzelnen Prozessschritten und weniger in der Prozessstruktur nieder; deshalb halten wir es für angebracht, das folgende objektorientierte Prozessmodell vorzustellen.

Beim Einsatz objektorientierter Techniken kann prinzipiell die Einteilung eines Software-Projektes in Phasen (wie zum Beispiel beim sequenziellen phasenorientierten Prozessmodell) erhalten bleiben. Die ersten beiden Phasen (Problemanalyse und Systemspezifikation) sind bereits vom Denkmodell – das vereinfacht ausgedrückt lautet: ein System besteht aus miteinander kommunizierenden Objekten – und von den im Zusammenhang mit diesem entwickelten Darstellungstechniken, zum Beispiel UML (siehe dazu zum Beispiel Hitz und Kappel, 2002) betroffen. Dies schlägt sich aber nicht im Prozessmodell selbst nieder, sondern betrifft die zur Ausführung der in den beiden Phasen heranzuziehenden bzw. empfohlenen Methoden, Techniken und Darstellungsmittel.

Ein wesentlicher Unterschied zu den konventionellen Phasenmodellen kommt bei der objektorientierten Programmierung dadurch zustande, dass die Implementierung durch das Zusammenfügen bereits existierender Bausteine geprägt ist. Daraus ergeben sich folgende Konsequenzen:

- Der Entwurf kann nicht mehr losgelöst von der späteren Implementierung durchgeführt werden, weil schon während des Entwurfs berücksichtigt werden muss, welche Bausteine zur Lösung des Problems zur Verfügung stehen. Entwurf und Implementierung rücken dadurch näher zusammen, und schon eine andere Wahl der Programmiersprache oder Klassenbibliothek kann zu einer völlig anderen Systemarchitektur führen.
- Die Dauer der Implementierungsphase wird verkürzt. Insbesondere liegen schon sehr früh (Teil-)Ergebnisse vor, an denen man die Adäquatheit und Korrektheit des Entwurfs kontrollieren kann. Fehlentscheidungen können so früher erkannt und korrigiert werden. Dadurch entsteht eine enge Rückkopplung zwischen der Entwurfs- und der Implementierungsphase.
- Die Klassenbibliothek, aus der die Bausteine stammen, muss ständig gewartet werden. Einsparungen, die bei der Implementierung gemacht wurden, werden so teilweise wieder zunichte gemacht. Es wird sogar eine neue Funktion/Rolle geschaffen, nämlich die des „Klassenbibliothekars“. Seine Aufgabe ist es, dafür zu sorgen, dass die Klassenbibliothek auf effiziente Weise benutzt werden kann.

- In der Testphase wird nicht nur die spezifikationskonforme Funktionsweise des neuen Produkts überprüft, sondern auch die der dabei verwendeten Bausteine. Festgestellte Mängel müssen genauestens protokolliert werden. Die daraus resultierenden Änderungen müssen zentral in der Klassenbibliothek vorgenommen werden, damit sie sich auch auf andere (sowohl gleichzeitig ablaufende als auch zukünftige) Projekte auswirken.
- Neu entstandene Klassen müssen im Hinblick auf ihre allgemeine Verwendbarkeit geprüft werden. Wenn Aussicht besteht, dass ein Baustein auch in anderen Projekten verwendet werden kann, muss er in die Klassenbibliothek aufgenommen und entsprechend dokumentiert werden. Dazu gehört auch, dass die neue Klasse anderen Entwicklern bekannt und zugänglich gemacht wird, die möglicherweise davon profitieren können. Es werden dadurch auch neue Anforderungen an die Kommunikationsstrukturen gestellt.

Abbildung 2.19 zeigt das objektorientierte Prozessmodell. Die Spezifikations- und Entwurfsphase wachsen zusammen, denn bereits bei der Spezifikation der Eigenschaften, die das geplante Software-Produkt haben soll, ist es ratsam, den durch die Klassenbibliothek determinierten Vorrat an bereits vorhandener Funktionalität einzubeziehen.

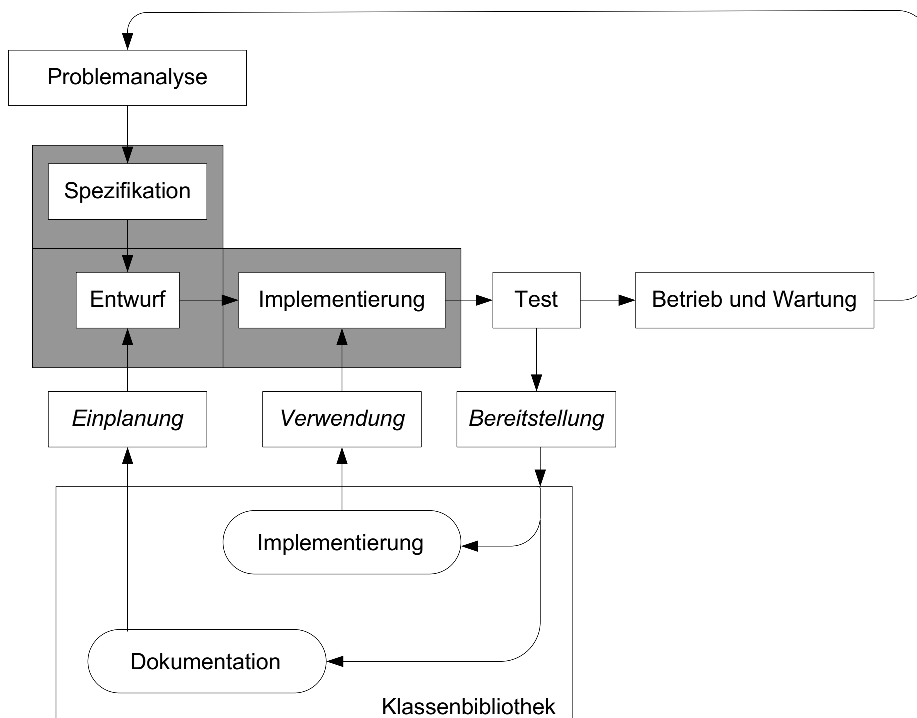

Abbildung 2.19: Objektorientiertes Prozessmodell

Ebenso rücken Entwurfs- und Implementierungsphase zusammen, denn der vorhandene Komponentenvorrat beeinflusst den Entwurfsprozess massiv, und Entwurfsentscheidungen können (durch Einsatz bereits vorhandener, implementierter Komponenten) experimentell verifiziert werden. Der Entwurf und die Implementierung erfolgen quasi verschränkt. Abbildung 2.19 darf nicht so interpretiert werden, dass durch die auf den Test folgende Bereitstellung neuer Bausteine ein Zyklus innerhalb desselben Projekts beginnt. Die Klassenbibliothek dient als projektübergreifendes „Werkzeug". Die in einem Projekt entwickelten Klassen können sich in Folgeprojekten produktivitätssteigernd auswirken. Der eigentliche Software Life Cycle kommt zustande, wenn beim Betrieb neue Anforderungen auftreten, die zu einer erneuten Problemanalyse führen.

Die Klassenbibliothek nimmt beim objektorientierten Vorgehensmodell eine so zentrale Stellung ein, dass wir ihre Bedeutung und die Organisation der Administrationsprozesse in einem weiteren Bild (Abbildung 2.20) veranschaulichen.

Eine für wiederverwendbar befundene Klasse gelangt durch Bereitstellung erstmalig in die Klassenbibliothek. Sie wird dort aber nicht sogleich archiviert, sondern gelangt als „unreife" Klasse vorerst in ein Zwischenlager. Um als „reif" erklärt zu werden, muss sie erst eine bestimmte Anzahl von erfolgreichen Wiederverwendungen bestehen. Sie durchläuft also mehrmals den so genannten Stabilisierungszyklus. Bei jeder Wiederverwendung wird beim Test festgehalten, wie gut sich die unreifen Klassen bewährt haben. Fehler werden in Zusammenarbeit mit dem Klassenbibliothekar (an seiner Stelle kann auch ein „Reuse-Team" eingerichtet sein) korrigiert. Positive Rückmeldungen werden ebenfalls von ihm (oder dem Reuse-Team) gesammelt. Er (beziehungsweise das Reuse-Team) ist dafür zuständig, Klassen für reif zu erklären und sie zu archivieren.

Reife Klassen müssen nicht mehr so streng beobachtet werden. Trotzdem kann es geschehen, dass sie veralten oder durch neuere Klassen ganz oder teilweise ersetzt werden. Die Einführung einer neuen Klasse kann auch eine Neustrukturierung der Klassenbibliothek zur Folge haben. Wenn sich dabei Schnittstellenänderungen ergeben, kann es

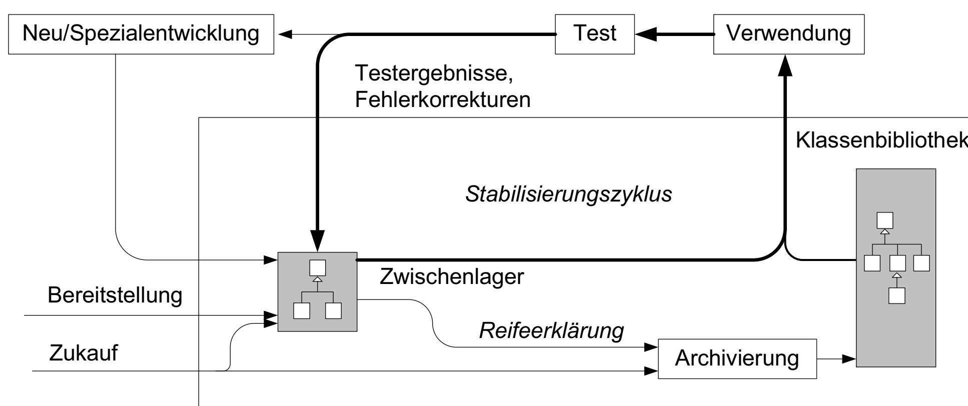

Abbildung 2.20: Administration von Klassenbibliotheken

angebracht sein, auch bereits archivierte Klassen neu zu überdenken. In diesem Fall übernimmt ein Spezialist oder ein eigenes Design-Team (losgelöst von allen anderen Projekten) eine Neuentwicklung der Klasse.

Klassen können auch losgelöst von einem speziellen Produktentwicklungsprojekt, quasi auf Vorrat, entwickelt werden (in Abbildung 2.20 mit Neu-/Spezialentwicklung bezeichnet). Auch diese Klassen kommen wie solche, die von einem Produkt-Entwicklungsprojekt bereitgestellt werden, in das Zwischenlager und müssen einen Reifeprozess durchmachen, ehe sie in die Klassenbibliothek integriert werden. Bei zugekauften Klassen hängt es davon ab, welche Qualität vom jeweiligen Lieferanten erwartet werden kann, ob eine Klasse zuerst in das Zwischenlager kommt.

Der *Vorteil* dieses Prozessmodells liegt darin, dass es speziell auf eine bestimmte Design- und Implementierungstechnologie zugeschnitten ist und die Nachteile der klassischen phasenorientierten Prozessmodelle technologiebedingt weitgehend beseitigt sind. Durch die Konzentration auf die Wiederverwendung zugekaufter Komponenten kann man sich bei der Produktentwicklung besonders auf die eigenen Stärken konzentrieren. Durch Nutzung von Halbfertig- und Fertigfabrikaten steigt die Produktivität und Qualität.

Der *Nachteil* dieses Prozessmodells ist, dass es die Implementierungstechnologie explizit vorschreibt und nicht geeignet ist, wenn andere Implementierungstechnologien in einem bestimmten Problemstellungskontext besser geeignet wären.

2.7 Leichtgewichtige (agile) Prozessmodelle

Seit einigen Jahren hat es sich eingebürgert, dass man Software-Entwicklungsprozessmodelle in zwei Gruppen unterteilt: schwergewichtige und leichtgewichtige Prozessmodelle. Unter „schwer“ bzw. „leicht“ versteht man dabei den Grad der Formalisierung des Prozesses und die Anzahl der mit diesem verbundenen (Zwischen-)Ergebnisse bzw. (Zwischen-)Produkte.

Schwergewichtige Prozessmodelle sind demnach die streng phasenorientierten Modelle, wie zum Beispiel das klassische sequenzielle Prozessmodell, das Wasserfallmodell und das V-Modell.

Leichtgewichtige Prozessmodelle, auch agile Prozessmodelle genannt, sind flexible, schwach formalisierte, iterative Prozessmodelle, zum Beispiel eXtreme Programming (XP), das erstmals von Kent Beck beschrieben wurde (siehe dazu Beck, 2000). Scrum (siehe dazu zum Beispiel Schwaber und Beedle, 2002) stellt einen Versuch dar, beide Welten zu verbinden. Die im Abschnitt 2.4 vorgestellten Spiralmodelle stellen gemäß dieser Klassifizierung ebenfalls einen Mittelweg dar. Stellvertretend für leichtgewichtige Prozessmodelle beschreiben wir hier XP.

XP – eXtreme Programming

Das Prozessmodell postuliert im Wesentlichen *vier* so genannte „prägende Werte“, man könnte diese auch besser die vier kritischen Erfolgsfaktoren nennen, und *zwölf* so genannte „Merkmale“. Die vier prägenden Werte sind: *Kommunikation, Einfachheit, Feedback* und *Courage.*

Durch die Betonung des Kommunikationsaspektes soll deutlich gemacht werden, dass der Projekterfolg ganz wesentlich von einem angemessenen Ausmaß und von der richtigen Art und Weise der Kommunikation zwischen den Projektbeteiligten abhängt, und dass daher dem Schaffen von geeigneten Rahmenbedingungen zur Förderung der Kommunikation im Rahmen der Projektorganisation besondere Bedeutung zukommt.

Durch die Betonung des Einfachheitsaspektes soll der Tatsache Rechnung getragen werden, dass viele Software-Produkte und auch die bei deren Herstellung ablaufenden Entwicklungsprozesse oft eine unnötig hohe Komplexität aufweisen und dass man daher bei jedem Schritt und bei jedem Zwischenergebnis darauf achten muss, Produkte und Prozesse so einfach wie möglich zu halten.

Durch die Betonung des Feedback-Aspektes soll sichergestellt werden, dass die Entwickler laufend über den Grad der Zufriedenheit der Auftraggeber informiert werden, den Grad der Anwenderakzeptanz für jede ausgelieferte Komponente kennen, und dass der Auftraggeber bei allen anstehenden Entscheidungen abschätzen kann, welche Folgen sich daraus im Hinblick auf Kosten, Termine, Produkteigenschaften und Produktqualität ergeben.

Durch die Betonung des Courage-Aspektes sollen die Projektbeteiligten zu verantwortlicher Eigeninitiative ermutigt werden. Sie sollen ohne pompöse Planung und hundertprozentige Absicherung die ihrer Meinung nach für den Projekterfolg notwendigen Schritte in Angriff nehmen und an die anderen Projektbeteiligten offen kommunizieren, auch wenn dabei Konflikte entstehen können, zum Beispiel bei Änderungen an „fremdem“ Code.

Ausgehend von diesen vier grundlegenden Aspekten postuliert das XP-Prozessmodell folgende Strukturierung bzw. folgenden Prozessablauf (siehe dazu auch Abbildung 2.21):

Der Entwicklungsprozess umfasst drei Teilprozesse: (1) Release-Planung, (2) iterative Release-Erstellung, (3) Akzeptanztests und Release-Veröffentlichung. Das Prozessmodell postuliert eine release-bezogene Produktentwicklung. Man beginnt mit der Entwicklung eines ersten Release (Systemkern), wobei die Entwicklungsdauer möglichst kurz gehalten wird (maximal drei bis vier Monate) und testet die Akzeptanz. Danach versucht man, in einer kontinuierlichen Abfolge neue Releases (Systemerweiterungen) zu erstellen. Durch kontinuierliche Akzeptanztests versucht man das für die Erstellung des jeweils folgenden Release erforderliche Feedback von den Benutzern zu bekommen. Diese Vorgehensweise entspricht exakt der Vorgehensweise bei evolutionärer, Prototyping-orientierter Software-Entwicklung.

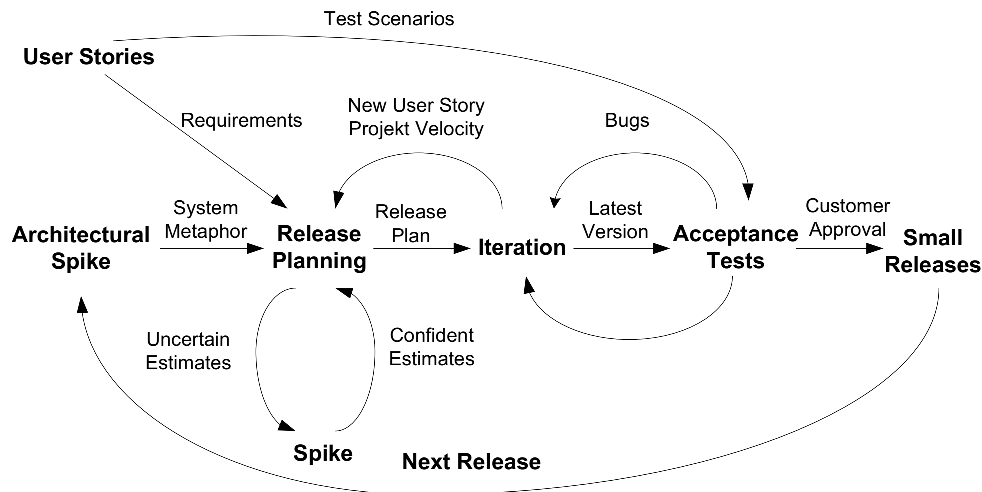

Abbildung 2.21: XP – Prozessstruktur (vgl. XP, 2004, Original v. J. D. Wells)

Ein Release-Zyklus beginnt mit der Release-Planung. Die eigentliche Entwicklung erfolgt danach in mehreren aufeinander folgenden Iterationsschritten. Der Teilprozess Release-Planung wird immer dann angestoßen, wenn sich Änderungen oder Erweiterungen der Benutzeranforderungen ergeben, die Systemarchitektur verändert wird oder sich Rahmenbedingungen ändern, zum Beispiel die Teamzusammensetzung.

Die Strukturierung des Teilprozesses Iteration, in dem die eigentliche (Release-)Entwicklung erfolgt, zeigt Abbildung 2.22. Der Teilprozess beginnt mit einem Planungsschritt, in dem Umfang und Dauer der Weiterentwicklung festgelegt werden. Die Iterationsplanung wird wie die Release-Planung von den Entwicklern gemeinsam mit dem Auftraggeber in Form eines Planungsspiels durchgeführt. Der Auftraggeber ist dabei primär für die Priorisierung der Weiterentwicklungsalternativen zuständig. Die Entwickler haben die Aufgabe, den dafür erforderlichen Aufwand in Abhängigkeit von den technischen und organisatorischen Rahmenbedingungen abzuschätzen. Im Planungsprozess ist zu berücksichtigen, dass eine Iteration nicht länger als zwei, maximal vier Wochen dauern soll.

Ein wichtiges Merkmal der Programm-(Release-)Entwicklung ist die so genannte Paarprogrammierung (pair programming). Nach der Iterationsplanung werden von den Entwicklern Paare gebildet, die während des folgenden Iterationsprozesses ihre Aufgaben gemeinsam nach dem „Vier-Augen-Prinzip“ erledigen. Die Zusammensetzung der Paare wird bei jedem Release-Zyklus geändert. Durch diese Organisationsform soll der Kommunikationsprozess gefördert und sichergestellt werden, dass mindestens zwei Personen über jedes Detail der Produktgestaltung und des Entwicklungsprozesses Bescheid wissen.

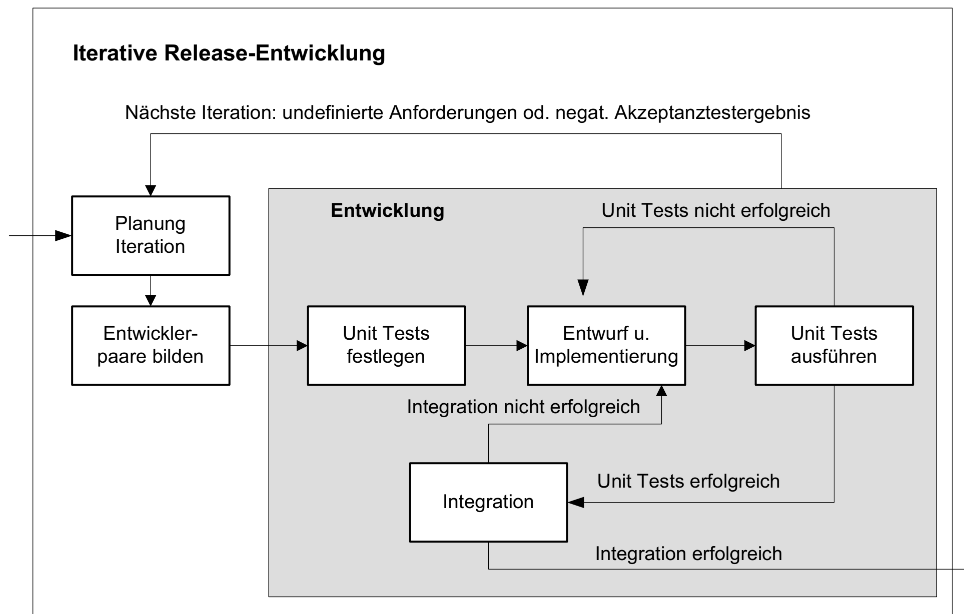

Abbildung 2.22: Teilprozess iterative Release-Entwicklung

Der Entwicklungsprozess beginnt mit der Festlegung von Komponententests (so genannter Unit Tests), mit denen die Entwicklerpaare später überprüfen, ob die Anforderungen an die zu implementierende Systemkomponente erfüllt werden. Danach folgen Entwurf und Implementierung der jeweiligen Systemkomponente. Wenn diese einen ausreichenden Reifegrad aufweist, werden die Komponententests ausgeführt und Entwurf und Implementierung so lange überarbeitet, bis alle Tests zufriedenstellend abgeschlossen sind. Danach wird vom Entwicklerpaar die Integration der entwickelten Systemkomponente in das Gesamtsystem vorgenommen. Diese ständige Integration ist ein charakteristisches Merkmal des XP-Prozessmodells. Im Zuge des Integrationsversuches kann sich herausstellen, dass der Entwurf und/oder die Implementierung trotz erfolgreicher Komponententests nochmals überarbeitet werden müssen, um die Integrationserfordernisse zu erfüllen.

Nach erfolgreicher Integration einer Systemkomponente wird der Teilprozess Akzeptanztests und Release-Veröffentlichung angestoßen. Wenn alle (in Form so genannter User Stories) spezifizierten Akzeptanztests erfolgreich beendet sind, wird das System-Release zur Benutzung freigegeben (veröffentlicht). Wenn das Release noch nicht alle geplanten Funktionen bereitstellt, wird erneut der Teilprozess Release-Planung angestoßen und die Entwicklung eines weiteren Release begonnen. So schließt sich der Kreis der Systementwicklung.

Die im Folgenden skizzierten zwölf Punkte, respektive Praktiken, charakterisieren das *X*P-Entwicklungsparadigma:

- Die Ziele, Leistungsmerkmale und der Funktionsumfang der einzelnen Releases werden im Rahmen eines Planungsspieles (*planning game*) gemeinsam von Auftraggebern und Entwicklern diskutiert, beschlossen und in Form von *Story Cards/User Stories* dokumentiert.
- Die Vorgehensweise der Systementwicklung und -bereitstellung ist Release-orientiert. Das heißt, in kurzen Abständen werden System-Releases mit überschaubarem Funktionszuwachs entwickelt und ausgeliefert (*small releases*).
- Eine möglichst einfache Systemmetapher bildet die Ausgangsbasis für den Entwicklungsprozess; jede unnötige Komplexität soll von vornherein vermieden werden.
- Entwurf und Implementierung jedes neuen Release muss mit besonderem Augenmerk nach dem Prinzip „so einfach wie möglich“ vorgenommen werden und unnötige (nicht zweifelsfrei erforderliche) Komplexität muss sofort beseitigt werden.
- Kontinuierliches Testen gehört zu den Hauptaufgaben der Entwickler und der Anwender. Die Entwickler beginnen jeden Entwicklungsschritt stets mit der Festlegung von Komponententests und beenden ihn erst nach erfolgreicher Durchführung aller spezifizierten Komponententests. Die Anwender definieren Szenarien für System- und Akzeptanztests und spielen diese vor der Freigabe eines neuen System-Release konsequent durch.
- Die Entwickler sind zu kontinuierlicher Restrukturierung von Entwurf und Implementierung der Komponenten verpflichtet (*refactoring*), um unnötige Komplexität zu beseitigen und den Code zu vereinfachen und flexibler zu gestalten. Die Funktionalität wird dabei nicht geändert.
- Die Entwickler arbeiten in Paaren (*pair programming*). Jede Codezeile wird von zwei Entwicklern gemeinsam entwickelt und verantwortet. Diese Praktik stützt sich auf die These, dass zwei Personen mehr sehen und gemeinsam bessere Ideen entwickeln können als eine Person alleine. Darüber hinaus ist Paarprogrammierung ein wichtiger Beitrag zur Wissenssicherung. Der dadurch erzielte Nutzen hinsichtlich Qualitätssteigerung und Wissenssicherung sollte den notwendigen Mehraufwand deutlich übertreffen und damit die Wirtschaftlichkeit verbessern.
- Der gesamte Code ist gemeinsames Eigentum aller Entwickler (*collective ownership*). Jeder Entwickler darf (gemeinsam mit einem anderen) jede Codezeile zu jedem Zeitpunkt ändern.
- Der Integrationsprozess ist ein kontinuierlicher und kein punktueller Prozess. Ein Integrationsschritt findet immer dann statt, wenn eine Aufgabe erledigt bzw. eine Komponente fertig gestellt ist, das heißt alle dafür spezifizierten Komponententests erfolgreich durchgeführt wurden. Vor der Freigabe eines neuen System-Release werden die definieren Szenarien für System- und Akzeptanztests von Anwendern durchgespielt.
- Die Entwickler arbeiten in der Regel nicht mehr als vierzig Stunden pro Woche, auch wenn es Terminprobleme gibt. Sollte in einer Woche ausnahmsweise länger als vierzig Stunden gearbeitet worden sein, dann darf dies in der folgenden Woche keinesfalls

mehr geschehen. Diese Regel stützt sich auf die These, dass die Gefahr von erschöpften oder unzufriedenen Mitarbeitern den Projektverlauf stets negativ beeinflusst.

- Dem Entwicklungsteam gehören ein oder mehrere Mitarbeiter des Auftraggebers an (*on-site customer*), die für die Software-Entwickler stets verfügbar sind, um mit ihnen Problemstellungen diskutieren bzw. offene Fragen an sie stellen zu können.
- Zur Förderung der Kommunikation und Steigerung der Effizienz werden die Programme nach einheitlichen Richtlinien und Programmierstandards entwickelt, die das Team selbst festlegt.

In einem Software-Projekt, in dem der Entwicklungsprozess nach den Grundsätzen des *X*P-Prozessmodells strukturiert bzw. organisiert wird, kommen alle oben angeführten Praktiken zur Anwendung. Teilweise bedingen sie einander oder es herrscht ein direkter oder indirekter Zusammenhang zwischen ihnen. Zudem soll bei *X*P ein Team aus nicht mehr als zehn Software-Entwicklern bestehen, die alle etwa gleich gut qualifiziert sein sollten. Zur Förderung der Kommunikation soll in so genannten „Einraumteams“ gearbeitet werden. *X*P betont ausdrücklich, dass dem Dokumentationsaspekt eine untergeordnete Rolle zukommt, ja dass man auf Dokumentation im üblichen Sinne überhaupt verzichten kann. Die Dokumentation ist somit keine wesentliche Entwicklungstätigkeit. An ihre Stelle tritt die Intensivierung der Kommunikation, das kontinuierliche Anwender-Feedback sowie die Know-how- und Wissenssicherung durch Paarprogrammierung.

Die *Vorteile* des *X*P-Prozessmodells liegen vor allem in der Flexibilität bei der Anpassung an sich ändernde Rahmenbedingungen, in der Beschränkung auf das Wesentliche, im iterativen Grundmodell, in der starken Einbindung des Auftraggebers *(on-site customer-Prinzip)*, im Verzicht darauf, bereits bei der Produktentwicklung künftige Anwenderanforderungen vorauszudenken und gegebenenfalls umzusetzen, und in der Möglichkeit, Entwicklungsziele von einem Release auf das nächste verschieben zu können. *X*P eignet sich deshalb besonders für Produktentwicklungen, bei denen die Anforderungen an das Produkt nur in informeller, sehr vager Form vorliegen und/oder sich die Anforderungen häufig ändern.

Die *Nachteile* des *X*P-Prozessmodells sind, dass die postulierte Prozessorganisation, die für „kleine“ Projekte recht gut geeignet ist, mit zunehmender Produkt-/Projektkomplexität immer unpassender wird und keine verteilte Produktentwicklung möglich ist, die bei großen Projekten unabdingbar sein kann. Eines der Referenzprojekte zur Erprobung von *X*P ist deshalb auch gescheitert (siehe dazu Stephens und Rosenberg, 2003). Im *X*P-Prozessmodell wird dem (im Hinblick auf den Übergang von handwerklicher zu industrieller Herstellung von Software) immer wichtiger werdenden Aspekt der Wiederverwendung von Komponenten und der Schaffung wiederverwendbarer Komponenten zu wenig Beachtung geschenkt. Der *X*P-Prozess ist darauf ausgerichtet, jeweils nur ein spezifisches Problem zu lösen. Der Verzicht auf eine Systemdokumentation kann bei langlebigen Produkten, das heißt, wenn die personalisierte Wissenssicherung nicht mehr greift, schwerwiegende Folgen haben.