Entwicklungsmethodiken zur kollaborativen Softwareerstellung – Stand der Technik

Tobias Hildenbrand (Universität Mannheim)

Astrid Behm (Forschungszentrum Informatik, Karlsruhe)

Asarnusch Rashid (Forschungszentrum Informatik, Karlsruhe)

Michael Geisser (Universität Mannheim)

Bezeichnung:	CollaBaWue-KSE-Arbeitsbericht-Stand_der_Technik-Methodiken
Dateipfad:	SVN/KSE/Arbeitsberichte
Änderungsdatum:	27.01.2006
Version:	v1.0
Arbeitspaket:	AP 1.1
Status:	abgeschlossen
Sichtbarkeit:	intern

Kurzfassung

Die weltweit wachsende Nachfrage nach Unternehmenssoftware erfordert immer neue Methoden und Formen der Zusammenarbeit (Kollaboration) bei der Softwareerstellung. Zu diesem Zweck untersucht und vergleicht dieses Arbeitspapier existierende Vorgehensmodelle und deren Evolution. Zusätzlich werde erste Ansätze zur kollaborativen Softwareerstellung vorgestellt und ihre Eignung für ein kommerzielles Umfeld analysiert. Die Arbeit verwendet hierzu einen eigenen Vergleichsrahmen, der u.a. auch so genannte „Kollaborationspunkte“ in Betracht zieht, d.h. Aktivitäten im Prozess an denen das Einbinden mehrerer Entwickler und/oder Anwender vorteilhaft ist. Die Erkenntnisse aus der vergleichenden Analyse des Stands der Technik werden schließlich dazu verwendet, Defizite existierender Ansätze aufzuzeigen und Anforderungen für unterstützende Werkzeuge abzuleiten.

Abstract

The growing demand for ever more business software requires novel methodologies and forms of collaboration in the software industry. Therefore, this working paper analyzes and compares existing process models and their evolution over time. Additionally, initial approaches to collaborative software development will be introduced and analyzed with regards to commercial requirements. In order to be able to do so, this study uses a proprietary framework for comparison including considerations about so-called “collaboration points”, i.e. activities within the process where collaboration of multiple developers and/or users are advantageous. Perceptions gained from the results of the comparative analysis of the state of the art will be utilized in order to unveil deficiencies of existing approaches as well as deriving requirements for supporting tools.

Inhaltsverzeichnis

1 Einführung

Dieses und die folgenden Kapitel sollen den Bezug zwischen existierenden Entwicklungsmethodiken und den Methoden der kollaborativen Softwareerstellung darlegen.

1.1 Problemstellung

Die wachsende Nachfrage nach Unternehmenssoftware und die zunehmende Durchdringung vieler Geschäftsfelder mit neuen Anwendungen stellen die Softwareindustrie vor nie da gewesene Herausforderungen. Insbesondere bei kleinen und mittleren Softwareunternehmen bieten zwischenbetriebliche Kooperationen und unternehmensübergreifende Kollaboration in gemeinsamen Softwareprojekten sowie modulare, komponentenbasierte Architekturen zahlreiche Möglichkeiten, in Wertschöpfungsnetzwerken bzw. sog. Softwareökosystemen den steigenden Anforderungen des Marktes gerecht zu werden. Große Hersteller von Unternehmenssoftware, wie bspw. SAP und Oracle, können bisher noch durch ihre enormen Personal- und Wissensressourcen autonom bestehen, versuchen aber zunehmend auch, Softwareökosysteme um ihre eigenen Softwarestandards zu bilden, wie man es z.B. im NetWeaver-Umfeld[1] beobachten kann.

Zur Unterstützung der kollaborativen, zwischenbetrieblichen Softwareerstellung bedarf es daher geeigneter Methoden und Werkzeuge, um diese unternehmensübergreifenden Entwicklungsprozesse zu realisieren. Insbesondere bei der Methodik bzw. dem Vorgehensmodell stellt sich die Frage, inwieweit existierende Ansätze geeignet sind, um diese neue Form der Zusammenarbeit innerhalb der Softwareindustrie und mit den Anwenderunternehmen zu unterstützen.

1.2 Zielsetzung

Basierend auf der dem Forschungsprojekt CollaBaWü[2] zugrunde liegendem Verständnis der kollaborativen Softwareerstellung soll dieses Papier

- existierende Entwicklungsmethodiken und Vorgehensmodelle aus der Softwaretechnik (Software Engineering) auf ihre Eignung für kollaborative, zwischenbetriebliche Softwareprojekte untersuchen und
- eventuelle Schwachstellen und Defizite identifizieren.

Die Ergebnisse dieses Papiers bilden wiederum den Ausgangspunkt für die Analyse entsprechender Werkzeuge zur Unterstützung, Abbildung und Umsetzung dieser kollaborativen, zwischenbetrieblichen Softwareerstellungsprozesse.

1.3 Vorgehensweise und Gliederung

Zu diesem Zweck werden im zweiten Abschnitt die Grundlagen für die Betrachtung von Entwicklungsmethodiken eingeführt und darauf aufbauend ein dreistufiger Vergleichsrahmen für die Analyse der unterschiedlichen Modelle und Ansätze vorgestellt. Darauf aufbauend stellt Abschnitt 3.1 anhand der klassischen Prozessmodelle die grundlegenden Paradigmen in der Softwaretechnik vor. In Abschnitt 3.2 wird die zweite Gene-

[1] Eine Beschreibung der NetWeaver-Plattform-Komponenten und -Werkzeuge findet sich unter http://www.sap.com/solutions/netweaver/index.epx (29.11.05)
[2] http://www.collabawue.de

ration von Vorgehensmodellen analysiert, die im Wesentlichen auf die Wiederverwendung von Softwarekomponenten und prozessbezogene Qualitätssicherung hin konzipiert wurden[3]. Abschnitt 3.3 beschäftigt sich eingehend mit der aktuellsten Generation von Vorgehensmodellen, nämlich dem Rational Unified Process der Firma IBM / Rational und dem neuen V-Modell XT, dem deutschen Entwicklungsstandard für IT-Systeme des Bundes. Anschließend werden in Abschnitt 4 weitere methodische Ansätze analysiert, die sich nicht als Vorgehensmodell im eigentlichen Sinn verstehen und nicht den gesamten Softwarelebenszyklus abdecken.

2 Theoretischer Bezugsrahmen

Zu diesem Zweck werden im Folgenden zunächst die grundlegenden Elemente von Vorgehensmodellen und Softwaremethodiken vorgestellt und eingeführt.

2.1.1 Grundlagen

Balzert bezeichnet eine **Methode** zum einen allgemein als „planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen“ und speziell im SWT-Kontext auch als ”Oberbegriff von Konzepten, Notationen, methodischen Vorgehensweisen und Verfahren“ (Balzert (2000), S. 44). Unter einem **Verfahren** wiederum werden „ausführbare Vorschriften oder Anweisungen zum gezielten Einsatz von methodischen Vorgehensweisen“ (Balzert (2000), S. 45) verstanden.

Im Allgemeinen bezeichnet der Begriff **Methodik** oder auch Methodologie die Lehre von den Methoden. Die dem in Abschnitt 3.3.2 vorgestellten NIMSAD-Framework zur Vorgehensmodellanalyse zugrunde liegende Definition einer Methodik lautet:

”[...] an explicit way of structuring one’s thinking and actions. Methodologies contain model(s) and reflect particular perspectives of ’reality’ based on a set of philosophical paradigms. A methodology should tell you ’what’ steps to take and ’how’ to perform those steps but most importantly the reasons ’why’ those steps should be taken, in that particular order“ (Jayaratna (1994), S. 37).

Da NIMSAD eine der wesentlichen Grundlagen für den in dieser Arbeit verwendeten Vergleichsrahmen bildet, wird die oben zitierte Definition einer Methodik als Arbeitsgrundlage übernommen. Ein Prozessmodell oder Vorgehensmodell kann somit als die systematische, formelle Beschreibung einer bestimmten Methodik angesehen werden.

2.1.2 Prozess-Metamodelle

Die vorliegende Arbeit orientiert sich im Bereich der Prozessmodelle an einem von Noack und Schienmann (1999) entworfenen Metamodell, welches u.a. von Korthaus (2001) sowie Fettke und Loos (2002) verwendet und erweitert wurde. Es soll hier dazu dienen, übergreifend über die einzelnen Prozessmodellinstanzen, eine einheitliche Struktur und Terminologie für die Beschreibung und Analyse der grundlegenden Elemente von Prozessmodellen zu schaffen. Abbildung 3.1 stellt das Metamodell als konzeptuelles UML-Klassendiagramm dar. Die Hauptbestandteile des Modells – Rollen, Ergebnisse bzw. Artefakte, Aktivitäten und Phasen – werden in den folgenden Abschnitten näher erläutert werden. Techniken geben an, wie eine Aktivität durchzuführen ist, wobei eine Aktivität mehrere Techniken erfordern kann. Beispiele wären Brainstor-

[3] Abschnitt 2 sowie die Abschnitte 3.1 und 3.2 sind der Diplomarbeit „Kritische Analyse existierender Entwicklungsmethodiken in der Softwaretechnik und Erarbeitung eines Referenzprozesses zur kollaborativen Erstellung von Unternehmenssoftware“ (Hildenbrand, 2004) entnommen.

ming oder Use Case-Analyse für die Anforderungsanalyse. Richtlinien, Standards und Notationen erleichtern die Koordination und die Kommunikation bei großen Softwareprojekten. Ein exemplarischer Standard wäre XML Metadata Interchange (XMI), und UML die entsprechende Notation dazu. Im Gegensatz zu Prozessmetamodellen wie dem in Abbildung 1, welche die Modellierungselemente für konkrete Prozessmodelle spezifizieren, stellen Metaprozessmodelle wie z.B. das Spiralmodell organisatorische Rahmenwerke zum Einbinden von konkreten Prozessmodellen dar (siehe Abschnitt 3.4.2). Im Software Process Engineering Metamodel (SPEM) der OMG (Object Management Group (2002b)) wird ein weitaus detaillierteres Metamodell als das von Noack und Schienmann (1999) vorgestellt, wobei das hier verwendete in seiner Komplexität für die Ziele dieser Arbeit als ausreichend erachtet wird.

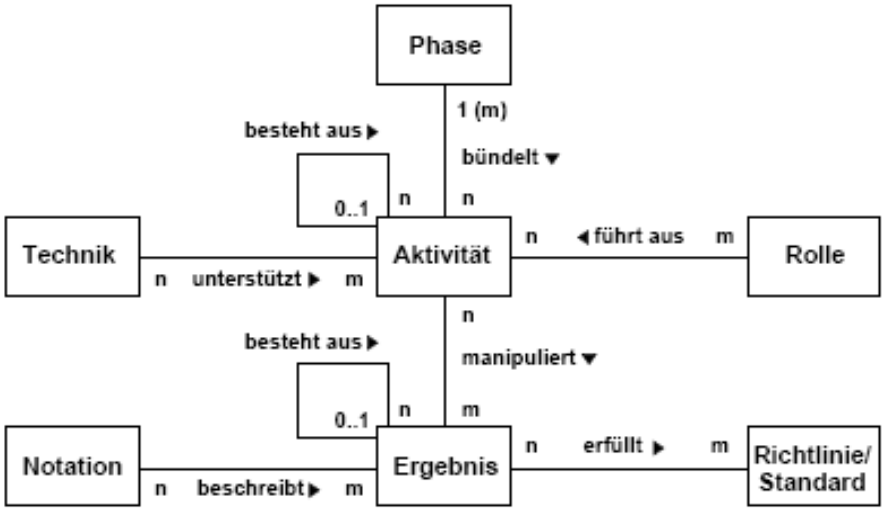

Abbildung 1: Prozess-Metamodell von Noak und Schienmann (1999)

Um sinnvolle Ergebnisse (Artefakte) hervorzubringen, sind jeder Aktivität innerhalb eines Prozessmodells bestimmte **Rollen** (Mitarbeiterprofile für menschliche Ressourcen) bzw. betriebliche (Informations-)**Ressourcen** zur Ausführung zugeordnet. Die einzelnen Rollen spezifizieren das Persönlichkeitsprofil eines Akteurs im Softwareentwicklungsprozess, bestehend aus Erfahrungen, Kenntnissen, Fähigkeiten und anderen persönlichen Charakteristiken, die Akteure zur Bewältigung der zugeordneten Aufgaben im Rahmen einer Aktivität erfüllen müssen. Prozessmodelle für umfangreiche Softwareprojekte, wie beispielsweise OPEN und der UP, beinhalten daher ein so genanntes Rollenmodell. Zur genauen soziotechnischen Charakterisierung der „intendierten Problemlöser“ stellt Jayaratna (1994) in seiner Arbeit das „dynamische mentale Konstrukt“ vor (siehe hierzu in Abschnitt 3.3.2).

Auf dem Weg hin zu einem abgeschlossenen Softwareprodukt entstehen in einem Softwareentwicklungsprozess eine Reihe von intermediären Produkten – so genannten **Artefakten** bzw. Ergebnissen: „Unter Beachtung von Methoden, Richtlinien, Konventionen, Checklisten und Mustern wird – ausgehend von einem oder mehreren gegebenen Artefakten – ein neues Artefakt erstellt oder der Zustand oder der Inhalt gegebener Artefakte geändert“ (Balzert (2000), S. 54). Aus dieser Definition geht hervor, dass einzelne Artefakte während des Prozesses inkrementell bearbeitet oder aber neu erstellt werden können. Beispiele für Artefakte innerhalb eines Prozessmodells sind unterschiedliche Dokumente, Modelle und Quelltextdateien.

Gemäß der Definition eines Softwareentwicklungsprozesses:”[...] the complete set of activities needed to transform users’ requirements into a consistent set of artifacts that represent a software product [...]“ (Jacobson u. a. (1999), S. 24), sind Aktivitäten Teil-

abläufe innerhalb eines Prozessmodells, die (Teil-)Ergebnisse für anschließende Aktivitäten liefern. Sie lassen sich weiterhin zu atomaren Aktionen zerlegen. Mehrere Aktivitäten werden häufig, anhand von inhaltlichen und/oder zeitlichen Aspekten, zu einzelnen Phasen innerhalb des Modells zusammengefasst. Aktivitäten können von unterschiedlichen Akteuren (Rollen) parallel ausgeführt werden, wohingegen Phasen in der Regel sequenziell – wenn auch teilweise in mehreren Iterationen (vgl. UP) – durchlaufen werden. Im folgenden Abschnitt werden die in der deutsch- und englischsprachigen Standardliteratur am häufigsten genannten Phasen der Softwareentwicklung konsolidiert dargestellt und dabei kurz erläutert.

Ein noch ausführlicheres Metamodell wird im australischen Standard Metamodel for Software Development Methodologies (AS 4651) beschrieben. Es basiert aber im Wesentlichen auch auf den Bestandteilen Prozessspezifikation (mit Aktivitäten, Aufgaben, Techniken etc.), Arbeitsprodukte (Artefakte), Rollen für Personen und Werkzeugen [AS4651-2004].

Softwarelebenszyklus

Um komplexe Softwareentwicklungsprozesse, wie man sie bei der Erstellung von USW in der Regel benötigt, besser zu strukturieren, bündeln so genannte „Phasen“ zusammengehörige Aktivitäten. Im Sinne eines umfassenden USW-Entwicklungsprozesses müsste hier an erster Stelle die Phase der „Unternehmensmodellierung“ (Business Engineering) aufgeführt sein. Da es sich hierbei aber eher um eine USW-spezifische „Vorstufe“ auf betriebswirtschaftlicher Ebene ohne unmittelbaren Bezug zur eigentlichen Softwareentwicklung handelt, und diese Phase bereits in Abschnitt 2.3 ausführlich im Rahmen der Kontextdefinition der Arbeit behandelt wurde, soll an dieser Stelle lediglich auf den betreffenden Abschnitt verwiesen werden.

Allen wesentlichen Phasenmodellen ist gemeinsam, dass durch sie einen „Softwarelebenszyklus“, bestehend aus zwei großen Teilen Systementwicklung und Systembetrieb, ausgedrückt wird (Stahlknecht und Hasenkamp (2003)). Diese Sichtweise lässt sich bereits in der Software Engineering-Definition von Sommerville (2002) in Abschnitt 3.1 erkennen. In den folgenden sechs Teilabschnitten und zusammenfassend in Abbildung 3.2 sind die in der Literatur am häufigsten vertretenen Phasen der Softwareentwicklung entlang des allgemeinen Softwarelebenszykluses aufgeführt.

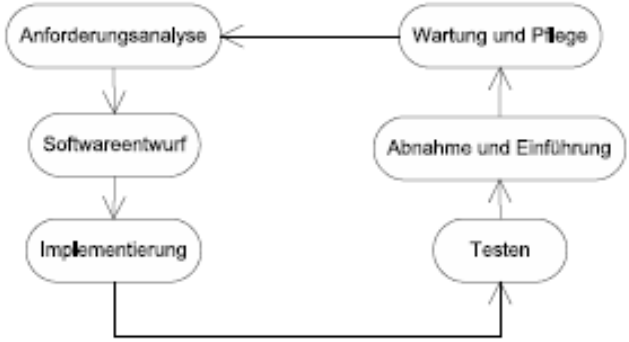

Abbildung 2: Allgemeiner Softwarelebenszyklus

Diese erste Phase im Systementwicklungsprozess, der **Anforderungsanalyse**, wird oft auch nur Analyse (Stahlknecht und Hasenkamp (2003)), Systemanalyse (Noack und Schienmann (1999)), Definitionsphase (Balzert (2000)) oder Software Requirements-Phase (Sommerville (2002) sowie Abran und Moore (2001)) genannt. Stahlknecht und Hasenkamp (2003) unterscheiden in der Analysephase die ”Istanalyse“, was in der hier verwendeten Terminologie der Unternehmensmodellierung entspricht, und die Entwicklung von ”Sollkonzepten“, was die Anforderungsanalyse im eigentlichen

Sinn darstellt. In der angloamerikanischen Literatur wird überdies häufig auch in Analogie zum Business Engineering von Requirements Engineering gesprochen (Sommerville (2002), S. 121ff.). Zu den wichtigsten Aktivitäten innerhalb der Analysephase gehören die Definition der Anforderungen an das Softwareprodukt und der Entwurf eines Fachkonzepts für das zu entwickelnde System (Analysemodell). Balzert (2000) stellt unterschiedliche Sichten, wie z.B. die objektorientierte, die datenorientierte oder die zustandsorientierte Sicht vor, welche bei der Definition des fachlichen Modells unterschieden werden können. Dieses konzeptuelle Modell soll dann in den folgenden Phasen als Basis für die informationstechnische Umsetzung der fachlichen Anforderungen, d.h. der Funktionalität, dienen.

Die Ergebnisse der Definitionsphase (Anforderungsanalyse) bilden nach Balzert (2000) den Ausgangspunkt bzw. die Grundlage für den **Software-** oder auch **Architekturentwurf**. Im Hinblick auf die nachfolgende Implementierungsphase spricht er auch von der „Programmierung im Großen“ (S. 686), da beim Entwurf bereits die Softwarearchitektur mit den technischen Details festgelegt wird, hierzu zählt auch die Bestimmung der Zielplattform für die Implementierung (wie z.B. Enterprise Java Beans (EJB) als Komponentenmodell). Im Rahmen der modellgetrieben Entwicklung (Model-Driven Development, MDD) und der entsprechenden Prozessmodellen soll anhand von semantisch eindeutig spezifizierten Entwurfsmodellen und so genannten Modellcompilern (Codegeneratoren) sogar vollständig auf eine manuelle Implementierungsphase verzichtet werden können (Kühne (2003)). Generierungstechniken als Form der Wiederverwendung werden in Kapitel 4 näher erläutert. In der englischsprachigen Literatur wird diese Phase allgemein auch häufig als Software Design bezeichnet (siehe hierzu Sommerville (2002) sowie Abran und Moore (2001)).

In der **Implementierungsphase** werden alle Aktivitäten zur tatsächlichen Umsetzung des Entwurfmodells zu einem ausführbaren Anwendungssystem (USW) gebündelt. Beispiele von typischen Implementierungsaktivitäten sind die Konzeption von Datenstrukturen und Algorithmen sowie die Umsetzung der Entwurfskonzepte und -muster in die Konstrukte der eingesetzten Programmiersprache. Weitere Bezeichner für diese Phase sind u.a. Software Construction (Abran und Moore (2001)) und Realisierungsphase (Stahlknecht und Hasenkamp (2003)).

Das **Testen** von Programmteilen, Teilsystemen (Unit Tests) oder des gesamten Anwendungssystems (Integrationstest) wird häufig auch als begleitende, parallele Aktivität in der Implementierungsphase mit aufgeführt, oder aber die Tests werden als Teil der Querschnittsfunktion „Qualitätsmanagement“ oder „Qualitätssicherung“ (Balzert (2000) „orthogonal“ zum gesamten Softwareentwicklungsprozess bzw. -lebenszyklus aufgefasst. Vor der Abnahme durch den Kunden und der Einführung im Unternehmen sollte jedoch in jedem Fall eine abschließende Testphase durchlaufen werden. Man unterscheidet beim Testen vor allem die Aktivitäten zur Verifikation und Validierung der Software (siehe Sommerville (2002)). In der **Abnahme**- und Einführungsphase wird die fertige USW erstmals als Ganzes übergeben und anschließend beim Anwenderunternehmen eingeführt, also in Betrieb genommen (daher auch: Deployment). Zu den Aktivitäten der Einführungsphase zählen u.a. Installation, Mitarbeiterschulung und Inbetriebnahme. Ab dem Zeitpunkt der ersten Inbetriebnahme (Go-Live) befindet sich die Software im Systembetriebsmodus und geht dann in die letzte Phase im allgemeinen Softwarelebenszyklus über – die Wartung und Pflege.

Den Oberbegriff für sowohl **Wartung** als auch Pflege bildet nach Balzert (2000) im Englischen (Software) Maintenance (siehe auch Abran und Moore (2001)). Da mit sehr großer Wahrscheinlichkeit im laufenden Betrieb der USW Fehler auftreten werden und sich ändernde Umweltbedingungen und/oder Benutzeranforderungen Anpassungen erfordern, ist diese Phase wichtig, um den Softwarelebenszyklus zu verlängern und die Kapitalrendite (Return on Investment) zu maximieren. Finden Aktivitäten aus den Un-

terkategorien Stabilisierung, Optimierung, Anpassung und Erweiterung nicht statt, kann ein Softwareprodukt sehr schnell veralten (Balzert (2000)). Komponentenbasierte Systemarchitekturen, wie sie in Abschnitt 4.2 vorgestellt werden, bieten bei der flexiblen Wartung und Anpassung der USW deutliche Vorteile, da einzelne fehlerhafte oder veraltete Komponenten gemäß definierter Schnittstellen einfacher ausgetauscht werden können.

Das vorgestellte allgemeine Phasen- bzw. Lebenszyklusmodell soll in Abschnitt 4.4 dazu dienen, die Phasenmodelle der analysierten Prozessmodelle, d.h. die einzelnen Prozessarchitekturen, vergleichbarer zu machen (siehe auch Abschnitt 3.3.3).

Querschnittsfunktionen bei der Softwareerstellung

Zusätzlich zu den meist sequenziell angeordneten Phasen der eigentlichen Softwareentwicklung aus Abschnitt 3.2.6 erfordern große Softwareprojekte im Unternehmensumfeld begleitende Aktivitäten, wie Projekt-, Konfigurations- und Qualitätsmanagement, die auch häufig unter dem Begriff „(Software-) Managementprozess“ zusammengefasst werden (Balzert (1998)). Ihnen kommt prozessbegleitend eine unterstützende Funktion zu.

Als **Projektmanagement** wird laut Stahlknecht und Hasenkamp (2003) „die Gesamtheit aller Tätigkeiten bezeichnet, mit denen Projekte geplant, überwacht und gesteuert werden“ (S. 219). Dies meint insbesondere „die Gesamtheit aller Koordinations- und Führungsaufgaben, die sich auf ein Projekt beziehen. Es ist die Anwendung von Wissen, Fertigkeiten, Werkzeugen und Verfahren auf Projektvorgänge, um die Bedürfnisse und Erwartungen der Stakeholder an ein Projekt zu erfüllen oder zu übertreffen„ (Project Management Institute (2000)).

Softwareprojekte mit dem Ziel der Erstellung von USW weisen eine hohe Komplexität und immer kürzer werdende Produktlebenszyklen auf. Daher kommt dem effizienten Management der Aktivitäten und Rollen innerhalb und zwischen den am Softwareerstellungsprozess beteiligten Organisationen eine große Bedeutung zu. Die wesentlichen Merkmale von Projekten sind nach Stahlknecht und Hasenkamp (2003): die Einmaligkeit des Unternehmens (im Gegensatz zu Geschäftsprozessen), die Zusammensetzung aus Teilaktivitäten, die Beteiligung von Personen und/oder Stellen (Rollen) unterschiedlicher Fachrichtungen (Interdisziplinarität), Teamarbeit (Kollaboration), das Konkurrieren mit anderen Projekten um Personal- und Sachmittel, Mindest- und Höchstwerte für Dauer und Aufwand (Budget und Zeitrahmen) sowie ein definierter Anfang und ein definiertes Ende (Projektziel). Spezielle Merkmale von IT-Projekten sind zusätzlich, dass sie die Entwicklung von Anwendungssystemen zum Inhalt haben, ein überwiegender Teil der Projektbeteiligten IT-Spezialisten sind und der Projektleiter bei internen Projekten meistens aus der IT-Abteilung entstammt. In der Literatur finden sich weitere Bezeichnungen für diesen Aufgabenbereich, z.B. Planungsphase bei Balzert (2000), Projektbegründung und -management (Stahlknecht und Hasenkamp (2003)), Project Management (Sommerville (2002)) und Software Engineering Management bei Abran und Moore (2001). Versteegen (2000) beschreibt die Tätigkeiten im Rahmen des Projektmanagements konkret am Beispiel des Rational Unified Process, einer kommerziellen Variante des UP, als konkretes Prozessmodell, welches es zu unterstützen gilt (vgl. Abschnitt 4.4.3).

Unter der Querschnittsfunktion **Konfigurations- und Änderungsmanagement** versteht man im Allgemeinen die fortlaufende Dokumentation der Systementwicklung, insbesondere aller System- und Programmanforderungen, des aktuellen Systemstatus, der vorgenommenen Änderungen während der Systementwicklung und der neu entstehenden Programmversionen einschließlich der Unterschiede zu der jeweils vorangegangenen Version während des Systembetriebs (Versionsverwaltung, vgl. Abschnitt 3.4.2). Bei Balzert (2000) wird das Konfigurations- und Änderungsmanagement als

Mittel aufgefasst, um eine geordnete Abwicklung der Wartungsaufgaben sicherzustellen, also noch als Teil der Softwarelebenszyklusphase „Wartung und Pflege“. Bei Versteegen u. a. (2001) und Weischedel (2003) wird jeweils detailliert auf die beiden Tätigkeitsbereiche eingegangen. Die beiden Prozesse werden andererseits auch häufig unter dem Begriff „Qualitätssicherung“ bzw. „Qualitätsmanagement“ zusammengefasst (siehe Noack und Schienmann (1999)). Das wohl bekannteste Beispiel eines Werkzeuges zur Konfigurationsverwaltung ist das Concurrent Versions System (CVS), welches bereits in vielen integrierten Softwareentwicklungsumgebungen zum Einsatz kommt (Fogel und Bar (2002)). Es handelt sich dabei um einen offenen Standard3, der seinen Ursprung in der Open Source Community hat (vgl. hierzu Abschnitt 5.2.2), aber häufig auch in kommerziellen Projekten zum Einsatz kommt.

Der Standard DIN ISO 9126 liefert für das **Softwarequalitätsmanagement** die folgende Definition: „Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen“. Über den Qualitätsbegriff gibt es laut Balzert (2000) fünf verschiedene Auffassungen, die die unterschiedlichen betrieblichen Sichten auf das Softwareprodukt wiedergeben: der transzendente, der produktbezogene, der benutzerbezogene, der prozessbezogene und der Kosten/Nutzen-bezogene Ansatz. Wesentlich für die Betrachtungen im Rahmen dieser Arbeit sind vor allem die Prozess- und die Produktqualität, weil sie die anfangs angesprochene Effizienz (entspricht der Prozessqualität) und Effektivität (Produktqualität) bei der Erstellung von USW sicherstellen können – wichtige Standards in diesem Zusammenhang sind ISO 9000/9001 (vgl. Balzert (2000) und Thaller (2001)) und das Capability Maturity Model (CMM) des Software Engineering Institute (SEI)4 (siehe auch Dymond (2002) und Chrissis u. a. (2003)).

2.1.3 Entwicklung eines Vergleichsrahmens

Nachdem im vorherigen Abschnitt die grundlegende Struktur von Softwareentwicklungsprozessen innerhalb von Organisationen vorgestellt, und eine einheitliche Terminologie für die fundamentalen, in einem Prozessmodell auftauchenden Konzepte eingeführt wurde, soll nun das Rahmenwerk zur Einordnung, Analyse, Evaluation und Vergleich von existierenden Prozessmodellen konstruiert werden. Dieser theoretische Rahmen gliedert sich in drei unterschiedlich abstrakte Stufen:

1. **Klassifikation** des Prozessmodells durch taxonomische Einordnung,

2. **Analyse** und Evaluation einzelner Methodiken und

3. Gegenüberstellender **Vergleich** mehrerer Methodiken anhand bestimmter Kriterien.

In den folgenden drei Unterabschnitten werden die Elemente dieses Theoriemodells näher erläutert.

Klassifikation von Entwicklungsmethoden

Um die in dieser Arbeit vorgestellten Prozessmodelle und Softwaretechniken im bestehenden „Methodik-Dschungel“ (Avison und Fitzgerald (1995)) einordnen zu können, wurde ein vierstufiges Klassifikationsschema basierend auf den Arbeiten von Hirschheim und Klein (1989), Hirschheim u. a. (1995) sowie Iivari u. a. (2001) gewählt. Die ursprüngliche Prämisse dieses Rahmenwerks war die Entwicklung einer hierarchischen Taxonomie zur Strukturierung der Methodiken im ISD.

Abbildung 3: Hierarchische Modell der IS-Entwicklung nach Iivari et al. (2001)

Wie aus Abbildung 3.3 deutlich wird, haben Iivari u. a. (2001) oberhalb der eigentlichen Methodiken zwei weitere Abstraktionsebenen empirisch ermittelt. Diese erhöhte Abstraktion bei der Betrachtung erleichtert sowohl die Einordnung neuer Methodiken als auch die Selektion einer bereits klassifizierten für ein konkretes Softwareprojekt und unterstützt somit das Methodology Engineering, d.h. das Anpassen und Entwickeln von individuellen Methodiken (vgl. Kumar undWelke (1992)). Um den Charakter der entstehenden Graphenstruktur der Taxonomie zu verdeutlichen, soll nun ein exemplarischer Pfad durch alle vier Hierarchieebenen beschrieben werden: Auf höchster Ebene soll das funktionalistische Paradigma stehen, welches die ontologischen, epistemologischen, methodischen und ethischen Grundhaltungen spezifiziert. Der objektorientierte Ansatz ist eine Instanz dieses Paradigmas. Die Methodik des UP (Jacobson u. a. (1999)), die in Abschnitt 4.4.3 genauer analysiert wird, ist wiederum eine Instanz des objektorientierten Ansatzes, und Use Case-Modellierung ist eine Technik, die innerhalb des UP verwendet wird.

Analyse und Evaluation von Entwicklungsmethodiken

Will man nun die einzelnen Methodiken analysieren, bedarf es eines einheitlichen Betrachtungsschemas. Bei NIMSAD (Normative Information Model-based Systems Analysis and Design) handelt es sich um ein konzeptuelles Rahmenwerk, welches epistemologisch auf dem Systembegriff bzw. der Systemtheorie basiert (vgl. hierzu Checkland (1981)). Jayaratna (1994) versteht unter einem konzeptuellen Rahmen ein Metamodell, mit dessen Hilfe Konzepte, Modelle, Techniken und Methodiken entweder verdeutlicht, verglichen, kategorisiert, evaluiert und/oder integriert werden können, was NIMSAD zu einem geeigneten, universellen Werkzeug zur Erreichung der Forschungsziele dieser Arbeit macht. Im Gegensatz zu solch einem Rahmenwerk, impliziert die eigentliche Methodik (Prozessmodell) als betrachtetes Objekt immer eine zeitabhängige Denkweise und/oder Ablaufphasen (Jayaratna (1994)). NIMSAD stellt somit auch ein Werkzeug zur Evaluation von allgemeinen Problemlösungsprozessen zur Verfügung und eignet sich daher als theoretische Linse, um sowohl das analytische als auch das konstruktive Ziel dieser Arbeit zu erreichen. Jayaratna selbst bezeichnet “[...] the critical examination of methodologies“ (Jayaratna (1994), S.69) als das Hauptziel seiner Arbeit. Sein Vergleichsrahmen wurde in der Literatur bereits häufig zur Analyse und Bewertung von Softwareprozessmodellen herangezogen, wie u.a. bei Korthaus (1997), Forsell u. a. (1999) und Eriksson u. a. (2004). Die wesentlichen Elemente des Frameworks werden in Abbildung 3.4 dargestellt und im Folgenden näher erläutert, da im weiteren Verlauf der Arbeit häufig auf die NIMSAD-Terminologie und die einzelnen Elemente zurückgegriffen wird.

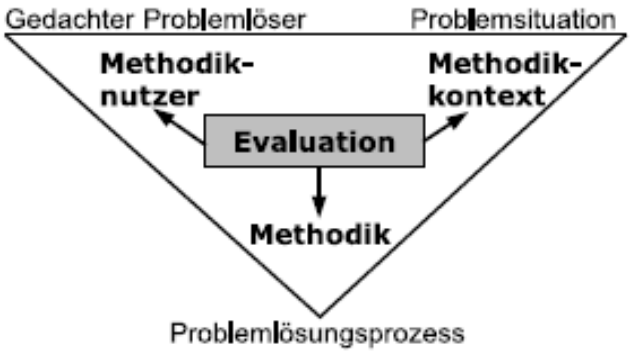

Abbildung 4: Das NIMSAD-Framework von Jayaratna (1994)

Methodikkontext (Problemsituation): Wie bereits in Kapitel 2 erläutert wurde, stellen Organisationen, d.h. Anwenderunternehmen, im Allgemeinen den Kontext für IS dar; ebenso laufen die den IS zugrunde liegenden Entwicklungsprozesse (ISD) innerhalb eines organisatorischen Kontextes ab. Wie sich im Organizational Dimensional Model aus Abbildung 2.1 erkennen ließ, kommt es bei Prozessen innerhalb einer Organisation zu Interaktionen der vier von Leavitt (1972) beschriebenen Dimensionen der Organisation (Aufgabe, Mensch, Struktur und Mensch). Der Problemlöser oder Methodikanwender benötigt daher ein fundiertes Verständnis der am Prozess beteiligten Organisationen, um die realen Probleme erkennen und mit den anderen Organisationsmitgliedern interagieren zu können (Jayaratna (1994)).

Methodikanwender (Gedachter Problemlöser): Bei den Methodikanwendern handelt es sich um diejenigen Personen, welche die Aktivitäten eines Prozessmodells ausführen sollen, also z.B. die Softwarearchitekten, Entwickler und Projektmanager (Akteure bzw. Rollen, vgl. Abschnitt 3.2.4). Um die persönlichen Charakteristiken von diesen Methodikanwendern innerhalb des Problemlösungsprozesses analysieren und bewerten zu können, beschreibt Jayaratna (1994) im Rahmen von NIMSAD das Modell des dynamischen mentalen Konstruktes. Hierbei wird der individuelle Wahrnehmungsprozess, Werte, Ethikverständnis, Motive, Vorurteile, Erfahrungen, Argumentationsvermögen, Wissen und Fähigkeiten der Akteure mit in Betracht gezogen, ebenso die individuellen Strukturierungsprozesse, Rollen, Modelle und Rahmenwerke, auf die ein Methodikanwender zurückgreifen kann. Gerade im organisatorischen Umfeld lässt sich ein Dilemma feststellen, wenn die Argumentationsprozesse der Akteure sowohl von politischen als auch von intellektuellen Denkprozessen beeinflusst werden. Das mentale Konstrukt ist im Rahmen dieser Arbeit als Ergänzung zu den üblichen Rollenmodellen, wie sie in Abschnitt 3.2.4 beschrieben sind, zu sehen und geht mehr auf soziotechnische Aspekte ein (siehe Jayaratna (1994)).

Methodik (Problemlösungsprozess): Als Problemlösungsprozess wird die eigentliche Methodik, d.h. im SWT-Kontext das Prozessmodell an sich, analysiert. Jayaratna (1994) unterscheidet zu diesem Zweck acht unterschiedliche Prozessstufen, welche dann wiederum zu drei Phasen gebündelt werden. Bei diesem generischen Phasenmodell lassen sich die Analogien zu den aus der SWT-Literatur empirisch ermittelten grundlegenden Phasen aus Abschnitt 3.2.6 deutlich erkennen.

Problemformulierung: In dieser ersten Phase soll die Situation of Concern zunächst verstanden und durch eine genaue Diagnose analysiert werden. Danach wird durch NIMSAD untersucht, ob das analysierte Prozessmodell über Aktivitäten zur Definition einer Prognose (Sollkonzept) verfügt, und ob Abweichungen zwischen Diagnose und Prognose als Problemdefinition spezifiziert werden können. Am Ende dieser Phase soll ein so genanntes Notional System abgeleitet werden, welches die idealisierte Problemlösung widerspiegelt. Es entspricht in etwa einem Analysemodell in der SWT und soll

die Transformation zur Überwindung der zuvor spezifizierten Problemsituation ermöglichen, z.B. die Erstellung einer passenden USW.

Lösungsentwurf: Diese Phase von NIMSAD bildet die Analogie zur Entwurfsphase aus Abschnitt 3.2.6 und unterteilt sich in die beiden Stufen „Durchführen des logischen/konzeptuellen Entwurfs“ und „Durchführen des physischen Entwurfs“. Der logische Entwurf konkretisiert hierbei das „Notional System“ mittels konzeptueller Elemente, wohingegen der physische Entwurf wiederum nach konkreten Mitteln und Wegen (Techniken) sucht, um das logische Modell zu realisieren.

Entwurfsumsetzung: Die letzte Phase des Problemlösungsprozesses beschreibt Aktivitäten zur Implementierung der zuvor entworfenen Lösung.

Evaluation von Kontext, Anwender und Methodik: Die Bewertung der Problemsituation beinhaltet beispielsweise eine Eingangsbewertung vor dem Prozessdurchlauf, Messungen des Commitments der Kunden und von Veränderungen während der Problemlösung und schließlich Messungen der Projekt-Performance nach dem Einschreiten durch die Problemlöser. Im Allgemeinen ist die Evaluationsdimension in existierenden Prozessmodellen wenig spezifiziert.

In Anhang 1 ist der im Rahmen dieser Arbeit verwendete Fragenkatalog zur NIMSAD-Analyse tabellarisch, nach Elementen, Phasen und Stufen gegliedert, dargestellt.

Neben Jayaratna (1999) bieten auch Heinrich und Häntschel (2000) Ansätze zur Evaluation des „im Unternehmen gelebten und definierten“ Softwareentwicklungsprozesses als ersten Schritt zur zielgerichteten Verbesserung der Softwareerstellung. Rezagholi (2000) vergleicht in diesem Kontext die vier wesentlichen Ansätze zur Evaluation und Verbesserung von Softwareentwicklungsprozessen: das Software Capability Maturity Model (CMM), die BOOTSTRAP-Methode, Software Process Improvement and Capability Determination (SPICE) und das Modell der European Foundation for Quality Management (EFQM bzw. European Quality Award (EQA)). Neben den genannten Standards kommt Rezagholi (2000) zu dem Schluss, dass zur Evaluation von Softwareentwicklungsprozessen insbesondere Metriken von zentraler Bedeutung sind. Als erste Ansätze hierzu werden Goal-Question-Metric (GQM) und ami hervorgehoben. Heinrich und Pomberger hingegen stellen einen prototypenbasierten Ansatz zur Evaluation des Softwareprodukts, und damit des Prozessfortschritts. Detaillierte Pflichtenhefte sollen hierbei durch Prototypen ersetzt und eine starke Benutzerbeteiligung erreicht werden (Heinrich und Häntschel (2000)).

Vergleichsrahmen

Nachdem mit NIMSAD ein relativ generisches und universelles Analysewerkzeug eingeführt wurde, soll nun im Folgenden ein spezifischerer Vergleichsrahmen vorgestellt werden, welcher u. a. auf ähnlichen Arbeiten von Avison und Fitzgerald (1995), Balzert (1998), Noack und Schienmann (1999) und Fettke und Loos (2002) aufbaut.

Die dort verwendeten Kriterien wurden konsolidiert und hinsichtlich dreier Leitmotive von CollaBaWü, nämlich Methodik (methodisches Vorgehen), Wiederverwendung (u.a. von Softwarekomponenten) und Kollaboration erweitert. Das gewählte Kriterienraster, bestehend somit aus den acht Kategorien Philosophie, Terminologie, Modellumfang, Prozessarchitektur und -steuerung, Komponentenverständnis, Kollaborationspunkten, Adaptionsmöglichkeiten sowie Werkzeugunterstützung, soll in den folgenden Unterabschnitten kurz näher erläutert werden und im weiteren Verlauf der Arbeit der Analyse und dem Vergleich von existierenden Prozessmodellen dienen. überdies bietet sich dieses Raster in Verbindung mit NIMSAD auch an, um eigene Ansatzpunkte für die Konstruktion eines Referenzprozesses zu strukturieren:

1. In der Kategorie **Philosophie** sollen allgemeine Eigenschaften und Charakteristiken, wie das zugrunde liegende Paradigma, das primäre Ziel, das antreibende Moment und die intendierten Domänen der Modelle (Methodikkontext), betrachtet werden (vgl. Balzert (1998)).
2. Es soll weiterhin geklärt werden, wie die in Abschnitt 3.2.2 bzw. Abbildung 3.1 vorgestellten, grundlegenden Elemente eines Prozessmodells bei den untersuchten Prozessmodellen – wenn vorhanden – benannt sind (**Terminologie**). Diese Betrachtung soll dazu beitragen, den Vergleich der einzelnen Prozessmodellkonstrukte transparenter zu gestalten (vgl. Prozessmetamodell, Abbildung 3.1).
3. Der **Umfang** eines Vorgehensmodells ergibt sich zum einen aus dem Umfang und dem Gehalt der Prozessdokumentation, wobei daraus keine direkten qualitativen Erkenntnisse gewonnen werden können. Jedoch vertreten Noack und Schienmann (1999) die These, dass umfangreiche Modelle oft steilere Lernkurven als „leichtgewichtigere“ erfordern. Man nennt sehr voluminöse Prozessmodelle wie beispielsweise das V-Modell und OPEN daher auch „Heavyweights“. Der zweite und ausschlaggebende Aspekt ist der Umfang der Phasen-, Prozess- und Rollenabdeckung („Breite“ des Modells). Die Phasenabdeckung bezieht sich auf den zeitlichen Verlauf des Softwareentwicklungsprozesses bzw. -lebenszykluses, wohingegen die Prozess- und Rollenabdeckung eher funktionale Aspekte analysieren. Unter der Prozessabdeckung versteht man neben dem eigentlichen Entwicklungsprozess (Softwarelebenszyklus) die Unterstützung der Querschnittsfunktionen (vgl. Abschnitt 3.2.7). Die Rollenabdeckung hingegen untersucht, welche unterschiedlichen Rollen und Mitarbeiterprofile durch das Prozessmodell spezifiziert werden. Bei sehr komplexen Prozessmodellen wie dem UP und Catalysis ist eine vorgeschaltete NIMSAD-Analyse hilfreich, um das Modell übersichtlicher zu strukturieren und die Methodik zu hinterfragen (siehe Abschnitt 3.3.2 bzw. Anhang A).
4. Neben der reinen Abdeckung der Standardprozesselemente wird bei dieser Arbeit besonders Wert auf die Analyse der Architektur und der Steuerung der Aktivitätenfolge des Prozessmodells (**Prozessarchitektur und -steuerung**) gelegt, da dies für die Konstruktion des Referenzprozesses als elementar erachtet wird. Die Prozessarchitektur beschreibt die Anordnung und Zusammenstellung von Phasen und Aktivitäten in einer sachlogischen (inhaltlichen) oder zeitlichen Reihenfolge (Noack und Schienmann (1999)) und ist in der Regel in diagrammartiger Form Teil der Prozessdokumentation. Man unterscheidet hierbei z.B. rein sequenzielle, iterative und nebenläufige Architekturen, wie sie in Abschnitt 3.4 bereits beschrieben werden. Die Steuerung der Durchführung einzelner Entwicklungsaktivitäten eines Vorgehensmodells (Aktivitätenfolge) kann laut Noack und Schienmann (1999) entweder aktivitätsorientiert (activity-oriented), ergebnisorientiert (product-oriented), vertragsorientiert (contract-driven) oder aber entscheidungsorientiert (decision-oriented) erfolgen. In der Praxis sind dann häufig Mischformen bei der Prozesssteuerung zu beobachten.
5. Da die in der vorliegenden Arbeit analysierten Prozessmodelle von unterschiedlichen Komponentendefinitionen ausgehen, wird bei der Analyse des **Komponentenverständnisses** versucht, jeweils die Vereinbarkeit und die Unterschiede mit der zuvor in Abschnitt 4.2 vorgestellten Arbeitsdefinition einer (Geschäfts-) Komponente herauszustellen. Hierzu zählen Merkmale wie beispielsweise Wiederverwendbarkeit, Abgeschlossenheit und Vermarktbarkeit (Komponentenbegriff). Außerdem wird analysiert, wie Komponenten im Vorgehensmodell gebildet werden – top-down oder bottom-up (Komponentenbildung). Schließlich wird noch bewertet, inwieweit die Phasen des allgemeinen Komponentenlebenszykluses nach Turowski (1999) durch das Modell abgedeckt werden (vgl. hierzu auch Abbildung 4.5).

6. Im Hinblick auf das neben der reinen Analyse zweite Forschungsziel dieser Arbeit, nämlich die Erarbeitung eines Referenzprozesses zur kollaborativen, komponentenorientierten Softwareerstellung, sollen bei den betrachteten Prozessmodellen besonders Phasen und Aktivitäten, die die Beteiligung von mehreren Akteuren in ihren jeweiligen Rollen erfordern, auf die Möglichkeiten zwischenbetrieblicher Aufgabenverteilungen und Zusammenarbeit (**Kollaborationspunkte**) hin untersucht und extrahiert werden.
7. Bei der Untersuchung der **Adaptionsmöglichkeiten** soll beschrieben werden, inwiefern die Vorgehensmodelle Anpassungen bezüglich des zu erstellenden Typs von USW, der organisationsspezifischen Rahmenbedingungen der Prozessanwender, der domänenspezifischen Anforderungen der Anwenderunternehmen und der Projektstruktur zulassen und gegebenenfalls mittels Modellvarianten oder Prozessprofilen unterstützen. Von besonderem Interesse ist im Rahmen dieser Arbeit die Ausgestaltung des Prozessmodells zu einem Referenzprozess für eine bestimmte Anwendungsdomäne, z.B. für die speziellen Anforderungen bei der Erstellung von USW für Finanzdienstleister (siehe Abschnitt 2.5).
8. In der letzten Kategorie **Werkzeugunterstützung** soll überprüft werden, ob die im Prozessmodell beschriebenen Aktivitäten sowohl im Rahmen des eigentlichen Entwicklungsprozesses (horizontale Werkzeuge) als auch in den Querschnittfunktionen (vertikale Werkzeuge) mit entsprechenden Werkzeugen oder zumindest mit Empfehlungen bzw. Anforderungen für passende Werkzeuge unterstützt werden (vgl. hierzu ECMA (1994) und ECMA (1997)). Laut Noack und Schienmann (1999) kann „[...] die Akzeptanz eines Vorgehensmodells durch eine angemessene Werkzeugunterstützung wesentlich verbessert werden [...]“ (S. 178). Auch Jacobson u. a. (1999) betonen in ihrer Arbeit immer wieder die reziprok unterstützende Beziehung von Prozessen und Werkzeugen in der SWT. Insbesondere interessant ist hierbei im Rahmen dieser Arbeit, ob und wie weit der Modellierungsstandard UML und entsprechende Werkzeuge explizit berücksichtigt wurden.

Die obige Aufstellung von Kriterien zur Analyse von Prozessmodellen erhebt keinen Anspruch auf Vollständigkeit. Die Zusammenstellung erfolgte anhand der Ergebnisse der durchgeführten Literaturrecherche und mit den Leitgedanken aus der Zielformulierung der Arbeit – der kollaborativen, komponentenbasierten Erstellung von USW. Eine Zusammenfassung des vorgestellten Kriterienkataloges findet sich in Anhang 2. Allgemeine Produktauswahlkriterien, wie Einflussmöglichkeiten auf die Weiterentwicklung des Prozessmodells, Herstellerunabhängigkeit und Marktposition des Anbieters (vgl. Noack und Schienmann (1999)) werden bei der Analyse der Prozessmodelle in den folgenden Abschnitten nicht explizit betrachtet. Auch Generierungstechniken, wie beispielsweise MDD als komplementäre Form der Wiederverwendung neben der „reinen“ Komponentenorientierung, sind im oben stehenden Raster nicht explizit mitberücksichtigt, da die derzeit etablierten Modelle diesen Aspekt der Softwareerstellung nur rudimentär betrachten (z.B. Booster von Korthaus (2001) und KobrA von Atkinson u. a. (2002)).

Im folgenden Abschnitt werden einige grundlegende Prozessmodelle der SWT kurz charakterisiert, wobei der entwickelte Vergleichsrahmen zunächst nur exemplarisch eingesetzt wird, da die klassischen Prozessmodelle an dieser Stelle lediglich dazu dienen sollen, fundamentale Prinzipien einzuführen die zur Erläuterung der moderneren Modelle dienen. Vollständig kommt der Vergleichsrahmen dann zum ersten Mal bei der kritischen Analyse der aktuellen Prozessmodelle in den Abschnitten 3.2 und 3.3 zum Einsatz.

3 Vergleich existierender Vorgehensmodelle

Im folgenden Abschnitt werden existierende Vorgehensmodelle und Methodiken aus der Software Engineering-Literatur im Hinblick auf ihre Eignung für kollaborative, komponentenorientierte Softwareerstellungsprozesse untersucht. Ein besonderer Fokus liegt hierbei auf den sog. „Kollaborationspunkten“ im Prozess, d.h. Aktivitäten, die durch die Miteinbeziehung unterschiedlicher Akteure aus möglicherweise verschiedenen Unternehmen effizienter und/oder effektiver ausgeführt werden können.

3.1 Klassische Vorgehensmodelle

Im folgenden Abschnitt werden aus den Kategorien sequenzielle, iterative, nebenläufige und objektorientierte Vorgehensmodelle einige klassische Vertreter vorgestellt, um die fundamentalen Prinzipien der Prozessmodelle und Methodiken „der ersten Generation“ einzuführen, da diese sich – häufig auch in kombinierter Form – in den fortgeschrittenen Modellen wieder finden (vgl. Abschnitt 3.2).

3.1.1 Sequenzielle Vorgehensmodelle

Das so genannte „Wasserfallmodell“ basiert auf frühen Arbeiten von Bennington (1956), Royce (1970) und Boehm (1981), wobei erst Letzterer die metaphorische Bezeichnung des Modells prägte. Es wurde von anderen Prozessen aus dem Ingenieurwesen abgeleitet und verdankt seinen Namen der kaskadenartigen Prozessarchitektur, d.h., dass „die Ergebnisse einer Phase wie bei einem Wasserfall in die nächste Phase fallen“ (Balzert (1998), S. 99). Die Arbeit von Royce (1970) führte bereits direkte Rückkopplungen zwischen den Phasen ein, zusätzlich zu dem reinen „stagewise model“ von Bennington (1956). Die Prozesssteuerung erfolg rein ergebnis- bzw. dokumentenorientiert. Seine starre aber einfache Struktur verhalf diesem Vorgehen zunächst zu großer Beliebtheit, doch setzen auch genau an dieser Struktur die zahlreichen Kritikpunkte an (vgl. dazu Balzert (1998), S. 101). Trotzdem wird dieses Modell noch von zahlreichen „modernen“ Softwareunternehmen eingesetzt.

3.1.2 Iterative Vorgehensmodelle

Iterative Modelle werden im Gegensatz zu rein sequenziellen Modellen auch „Zyklenmodelle“ genannt (Stahlknecht und Hasenkamp (2003)). Der grundsätzliche Unterschied zu den Wasserfallmodellen besteht darin, dass einzelne Phasen und Prozessabschnitte mehrfach in so genannten „Zyklen“ (Iterationen) durchlaufen werden können. Dieses Vorgehen ermöglicht dann Techniken wie z.B. das inkrementelle Verfeinern von Softwareprototypen.

Prototypenmodell

Einer der ersten Ansätze, welcher versuchte, die bekannten Probleme der sequenziellen Modelle anzugehen – beispielsweise, dass Auftraggeber und Endanwender oft nicht in der Lage sind, die Anforderungen an ein neues System explizit und/oder vollständig in einer einzelnen Anforderungsanalysephase (vgl. Abschnitt 3.2.6) zu formulieren – war der Einsatz von Softwareprototypen (Prototyping). Dieses Vorgehen ermöglicht es, frühzeitig ablauffähige Softwaremodelle in den Entwicklungsprozess einzubringen und somit eine zielgerichtetere Kommunikationsplattform mit den Anwendern zu schaffen. Die beschriebenen Prototypen unterscheiden sich von Prototypen aus anderen Ingenieurdisziplinen dadurch, dass sie nicht das „erste Muster einer großen Serie von Produkten“ (Balzert (1998), S. 114) darstellen, wie beispielsweise in der Automobilindustrie, sondern funktional eingeschränkte Demonstratoren. Dies ist u.a.

auf die grundsätzlichen Eigenschaften von Software als immateriellem Gut zurückzuführen (vgl. Abschnitt 2.1). Kieback u. a. (1992) und Budde u. a. (1992) unterscheiden daher vier Arten von Prototypen: Demonstrationsprototypen, Prototypen im engeren Sinn, Labormuster und Pilotsysteme, welche jeweils für unterschiedliche Phasen des Prozesses geeignet sind. Demonstrationsprototypen dienen als „Ausstellungsstück" in der Projektplanungsphase der Auftragsakquisition. Prototypen i.e.S. hingegen kommen in der Definitions- und Anforderungsanalysephase als ablauffähiges „Provisorium" zum Einsatz und Labormuster können hilfreich sein, um Entwurfsalternativen experimentell zu selektieren. Ein Pilotsystem schließlich enthält als Ergebnis der ersten Implementierungsphase bereits den eigentlichen Systemkern und wird dann zyklisch erweitert. Autoren wie Floyd (1984) sowie Stahlknecht und Hasenkamp (2003) unterscheiden orthogonal zur obigen Klassifikation bezogen auf mehrschichtige Systemarchitekturen horizontale und vertikale Prototypen. Das Musterbeispiel für einen horizontalen Prototyp ist die reine „GUI-Attrappe" und das Pilotsystem kann in diesem Zusammenhang auch als vertikaler Prototyp bezeichnet werden. Das Modell zeichnet sich vor allem durch seine über die verschiedenen Prototypen gewährleisteten zahlreichen Kollaborationspunkte zwischen Entwickler und Benutzer und seine Kombinierbarkeit mit anderen Prozessmodellen aus. Es stellt jedoch hohe Ansprüche an die Werkzeugunterstützung und sorgt für einen durchschnittlich höheren Entwicklungsaufwand als bei herkömmlichen Vorgehensmodellen, da die Prototypen im Allgemeinen zusätzlich zum eigentlichen Anwendungssystem erstellt werden. Es besteht überdies die „Gefahr", dass Wegwerfprototypen aus Zeitmangel doch in das Endprodukt übernommen werden und Qualitätsprobleme verursachen (Balzert (1998)).

Evolutionäres und inkrementelles Vorgehen

Wie auch das Prototypenmodell entstanden die evolutionären und inkrementellen Vorgehensweisen aus der in der Praxis beobachteten Problematik bei der einmaligen, verbindlichen Formulierung der Anforderungen durch den Auftraggeber, d.h. das Anwenderunternehmen. Evolutionär meint im Kontext der Prozessmodelle, dass ein definierter Produktkern aus „Mussanforderungen" als „Startwert" für den Prozess genommen wird (Balzert (1998)). Dieser Produktkern läuft durch die Entwurfs- und Implementierungsphase und wird dann an das Anwenderunternehmen als Kernsystem oder „Nullversion" ausgeliefert. Nach der ersten Auslieferung werden Anforderungen ergänzt und weitere Iterationen durchlaufen, bis das System den Ansprüchen des Auftraggebers entspricht. Diese Form der Entwicklung wird daher häufig auch als Versionen-Entwicklung (Versioning) oder evolutionäres Prototyping bezeichnet (vgl. Prototypenmodell). Die Phase „Wartung und Pflege" des allgemeinen Softwarelebenszykluses aus Abschnitt 3.2.6 wird bei diesem Modell lediglich als die Erstellung weiterer Versionen betrachtet. Evolutionäre Modelle bergen im Wesentlichen dieselben Vorteile wie die Prototypenmodelle, es besteht jedoch immer die Gefahr, dass bei ungeeigneten „Startwerten" nicht alle Evolutionspfade verfolgt werden können bzw. die komplette Systemarchitektur gegebenenfalls neu konstruiert werden muss (Balzert (1998)). Durch eine möglichst vollständige Erfassung der Anforderungen bereits in der ersten Analysephase können die oben genannten Probleme bei der inkrementellen Entwicklung vermieden werden. Trotzdem werden dann analog zum evolutionären Vorgehen bei jeder Iteration nur Teile der Anforderungen entworfen und implementiert. Das vollständige Analysemodell am Anfang soll verhindern, dass während des Projektes Passfehler zwischen einzelnen Inkrementen entstehen (Balzert (1998), siehe auch Abschnitt 3.4.3).

Spiralmodelle

Im Gegensatz zu den bisher vorgestellten Modellen handelt es sich bei dem von Boehm (1986, 1988) entwickelten Spiralmodell um ein Metaprozessmodell. „Metamodell" deshalb, weil es einen Prozessrahmen für den Einsatz unterschiedlicher Vorge-

hensmodelle beschreibt (vgl. Abschnitt 3.2.2). Das Metaprozessmodell gliedert sich in vier Schritte, welche für jedes Teilergebnis zyklisch durchlaufen werden müssen. Im ersten Schritt werden die Ziele, Alternativen und Randbedingungen für jedes Teilergebnis identifiziert. Es folgt in Schritt zwei die Evaluierung der Alternativen unter Beachtung der Ziele und Randbedingungen aus Schritt eins. Deckt die Evaluierung Risiken der einzelnen Alternativen auf, so müssen kosteneffektive Strategien zur Risikominimierung entwickelt werden (Balzert (1998)). Erst im dritten Schritt wird – wieder nach dem Kriterium des minimalen Risikos – ein geeignetes Prozessmodell oder eine Kombination aus mehreren ausgewählt und angewendet. Im letzten Schritt wird dann wiederum der nächste Schritt geplant, die vorangegangenen Schritte überprüft und die Entscheidung über die Durchführung des nächsten Zykluses getroffen. Das typische Bild der Spirale entsteht durch die Darstellung mehrerer durchlaufener Zyklen, wobei die Fläche der Spirale die akkumulierten Kosten und der Winkel des „Zeigers“ den Entwicklungsfortschritt innerhalb des jeweiligen Zykluses repräsentieren. Charakteristisch für dieses Modell sind die Risiko- und Kostenminimierung sowie das Verfolgen von Qualitätszielen. Es wird überdies versucht, ein Overengineering des Zielsystems durch geschickte Optimierung des Ressourcenverbrauchs zu vermeiden. Es ist somit das erste Modell, welches explizit auch ökonomische Aspekte betrachtet. Es ist damit ein sehr flexibles Modell, das kein bestimmtes Vorgehensmodell für die Softwareentwicklung vorschreibt und immer versucht, die ökonomischste Alternative für die nächste Iteration zu finden. Es unterstützt durch die Betrachtung von Alternativen ebenso die Wiederverwendung von Softwareprodukten im Entwicklungsprozess (vgl. Kapitel 4). Als nachteilig erweist sich der relativ hohe zusätzliche Management- und Planungsaufwand, der es für kleine und mittelgroße Projekte ungeeignet macht (Balzert (1998)). In Abschnitt 4.1 wird eine Erweiterung des Spiralmodells speziell für die kollaborative Anforderungsspezifikation vorgestellt (WinWin-Ansatz, Boehm u. a. (1995)).

3.1.3 Nebenläufige Vorgehensmodelle

An dieser Stelle soll zunächst das Kodak-Beispiel aus dem Einleitungsabschnitt wieder aufgegriffen werden. Unter den Bezeichnungen Concurrent Engineering, Simultaneous Engineering oder Parallel Engineering (vgl. Balamuralikrishna u. a. (2000) und Sträter (2002)) haben nebenläufige Prozessmodelle in der Fertigungsindustrie schon lange Einzug gehalten. Das Beispiel vom Hammer und Champy (2001) in der Einleitung machte deutlich, dass eine Parallelisierung und gleichzeitige Integration von Entwurfs- und Fertigungsprozessen enorm verkürzte Produkteinführungszeiten (Time-to-Market) zur Folge haben können. Dies wird auch durch die Bildung von funktionsübergreifenden Prozessteams, z.B. aus Fertigung, Marketing und Vertrieb, unterstützt, da somit möglichst alle relevanten Aspekte früh im Prozess berücksichtigt werden können. Weil aber ein verfrühtes paralleles „Anstoßen“ von logisch nachgelagerten Aktivitäten auch Risiken mit sich bringen kann, erfordert nebenläufiges Entwickeln „ein ständiges Abwägen, inwieweit es vertretbar erscheint, finanzielle Risiken durch vorzeitige Parallelisierung von Arbeitsfolgen einzugehen“ (Balzert (1998), S. 127). Will man nun das oben beschriebene Konzept auf die Entwicklung von USW übertragen, bedeutet dies, möglichst viele Aktivitäten und/oder Phasen zu parallelisieren oder zumindest stark überlappen zu lassen. Balzert charakterisiert nebenläufige Modelle weiterhin als Sammlung von organisatorischen und technischen Maßnahmen zur nebenläufigen Ausführung von einzelnen Aktivitäten „im Rahmen eines prinzipiell sequenziell angelegten Entwicklungsprozesses“. Außerdem werde bei dieser Klasse von Modellen die „auf Problemlösung gerichtete Zusammenarbeit“ (Balzert (1998), S. 127) der intendierten Problemlöser (vgl. Jayaratna (1994) und Abschnitt 3.3.2) – d.h. die funktionsübergreifende Kollaboration – gefördert und das gesammelte Expertenwissen frühzeitig zusammengeführt. Dabei sollen Zeitverzögerungen durch Techniken wie teilweise Parallelisierung, Vermeidung von Trial-and-Error-Verfahren und die „Reduktion von Wartezeiten zwischen

arbeitsorganisatorisch verbundenen Aktivitäten“ (Balzert (1998), S. 128) weitgehend eliminiert werden. Im Vergleich zu den iterativen Modellen wird hier von vornherein darauf abgezielt, das vollständige „Produkt“ auszuliefern und keine Inkremente (vgl. Abschnitt 3.4.2). Ein großer Vorteil der Kollaboration innerhalb der Prozessteams aus unterschiedlichen Problemlösern ist die Möglichkeit, Probleme wie z.B. Passfehler bei Komponentenarchitekturen, frühzeitig zu erkennen und zu beseitigen. Wenn das Prozessmodell effizient umgesetzt wird, bietet es eine „optimale Zeitausnutzung“ (Balzert (1998)). Allerdings stellt es, anhand der in Abschnitt 2.1 beschriebenen Charakteristika von Software bzw. USW, kein triviales Ziel dar, das Produkt auf Anhieb und ohne Iterationen bedarfsgerecht anzufertigen. Das Modell erfordert weiterhin einen hohen Management-, Planungs- und Personalaufwand, um kritische Entscheidungen rechtzeitig zu treffen, Ergebnisse zu synchronisieren und Fehler zu vermeiden bzw. Probleme rechtzeitig zu antizipieren (Balzert (1998)). Der hohe zusätzliche Koordinationsaufwand lässt sich aber teilweise durch informationstechnische Innovationen und Standards kompensieren. Diese Gedanken werden im folgenden Kapitel, in Anschnitt 5.2.3 und letztendlich in Kapitel 6 weiter verfolgt und letztendlich expliziert.

3.1.4 Objektorientierte Vorgehensmodelle

Bei den bisher betrachteten grundlegenden Prozessmodellen handelte es sich nach Iivari u. a. (2001) um Methodiken, welche unabhängig vom zugrunde liegenden Ansatz, beispielsweise strukturierter oder objektorientierter Programmierung, zum Einsatz kommen können. Es standen die zunächst die grundsätzlich unterschiedlichen Prozessarchitekturen und -charakteristiken im Vordergrund. Der objektorientierte Ansatz, auf den alle in diesem Abschnitt adressierten Modelle aufbauen, bringt neue Konzepte wie Klassen, Vererbung und Polymorphie mit sich (vgl. hierzu beispielsweise Oestereich (2001b,a)). Diese Konzepte bergen im Wesentlichen neue Möglichkeiten zur Wiederverwendung durch Komposition von Klassen aus Klassenhierarchien. Klassen können dann wiederum zu Subsystemen bzw. Komponenten aggregiert und ihrerseits zur Wiederverwendung freigegeben werden. In Abschnitt 4.1 wird auf die unterschiedlichen Techniken der Wiederverwendung näher eingegangen.

Balzert (1998) unterscheidet bei objektorientierten Modellen unterschiedliche Ebenen, Gebiete und Quellen der Wiederverwendung (Wiederverwendungsdimensionen). Die Wiederverwendungsebenen setzt er gleich mit den drei Phasen Analyse, Entwurf und Implementierung, und bei den Wiederverwendungsgebieten macht er einen Unterschied zwischen anwendungs- bzw. domänenbezogenen, auf die Anwendungsumgebung bezogenen sowie Klassen und Komponenten zur Systemanbindung. Klassen und Komponenten können intern hinterlegt sein oder müssen extern am Markt bezogen werden. Außerdem ist es für das allgemeine objektorientierte Modell entscheidend, „wann neue und wieder verwendbare Klassen und Subsysteme in das eigene Wiederverwendungsarchiv (Repository) eingeordnet werden“ (Balzert (1998)). Dies kann noch während des laufenden Entwicklungsprojektes, nach abgeschlossener Entwicklung oder nach einem speziellen „Aufbereitungsprozess“ im Anschluss an die Entwicklung geschehen (siehe Abbildung 4.1). Der Wiederverwendungsaspekt impliziert erstmals ein „Bottom-Up“-Vorgehen, ausgehend von vorhandenen Bausteinen, wohingegen die bislang vorgestellten Modelle „top-down“-orientiert waren (Balzert (1998)). Der Fokus liegt daher nicht mehr nur auf der Eigenentwicklung und das Verfahren lässt sich gut mit den evolutionären und inkrementellen Vorgehensweisen kombinieren, „da Erweiterungen und Änderungen mit der objektorientierten Technik gut durchführbar sind“ (Balzert (1998), S. 126).

Vorteile der objektorientierten Prozessmodelle sind die Verbesserung der Produktivität und der Qualität, die Möglichkeit, sich auf seine eigenen Stärken zu konzentrieren und fehlende Komponenten als Halbfabrikate zuzukaufen (vgl. hierzu Abbildung 4.1). Die

Wiederverwendungstechniken müssen allerdings durch eine geeignete Infrastruktur unterstützt und in der Firmenkultur verankert werden (siehe Abschnitt 4.5).

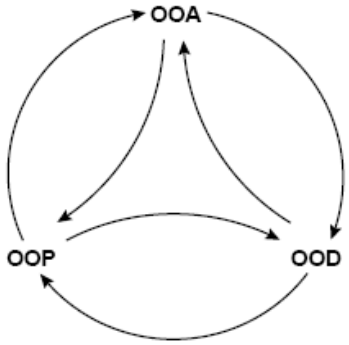

Abbildung 5: Objektorientiertes Baseballmodell von Coad und Nicola (1993)

Zu der Klasse der objektorientierten Modelle zählen u.a. das Fontänenmodell von Henderson-Sellers und Edwards (1990) und das Baseballmodell von Coad und Nicola (1993), welches in Abbildung 5 dargestellt wird. Es ist die Grundlage für den wiederverwendungsorientierten Mikroprozess des Booster-Ansatzes, welcher in Abschnitt 4.4.7 untersucht wird (siehe Korthaus (2001), S. 132). Das einfache Motto dieses Modells lautet: „Analyze a little, design a little, program a little, repeat".

3.1.5 Bewertung der klassischen Vorgehensmodelle

Tabelle 1 fasst noch einmal abschließend die primären Eigenschaften der bisher vorgestellten Modelle zusammen.

Prozessmodell	primäres Ziel	antreibendes Moment	Benutzer-beteiligung	Charakteristika
Wasserfallmodell	minimaler Managementaufwand	Dokumente	gering	sequenziell, volle Breite
Evolutionäres Modell	minimale Entwicklungszeit	Code	mittel	sofort: nur Kernsystem
Inkrementelles Modell	minimale Entwicklungszeit, Risikominimierung	Code	mittel	volle Definition, dann zunächst nur Kernsystem
Prototypenmodell	Risikominimierung	Code	hoch	nur Teilsysteme (horizontal oder vertikal)
Spiralmodell	Risikominimierung	Risiko	mittel	Entscheidung pro Zyklus über weiteres Vorgehen
Nebenläufiges Modell	minimale Entwicklungszeit	Zeit	hoch	volle Breite, nebenläufig
Objektorientiertes Modell	Zeit- und Kostenminimierung durch Wiederverwendung	Wiederverwendbare Komponenten	n/a	volle Breite in Abhängigkeit von wieder verwendbaren Komponenten

Tabelle 1: Bewertung der klassischen Vorgehensmodelle

Aufgrund ihres grundlegenden Charakters für die weitere Arbeit und der Repräsentation unterschiedlicher Prinzipien ist es an dieser Stelle nicht angebracht, schon einen expliziten Vergleich durchzuführen. Vielmehr gilt es zu hinterfragen, in wie weit methodisch-ingenieurmäßiges Vorgehen und Prozessmodelle überhaupt sinnvoll zu der Erstellung von USW beitragen können. Iivari u. a. (2001) stellen in ihrer Arbeit zwei völlig gegenläufige Tendenzen bei der Entwicklung von USW fest. Zum einen die totale Missachtung bzw. großzügige Auslegung von vorgeschriebenen Vorgehensmodellen in

softwareerstellenden Organisationen, was sich teilweise auf die weit verbreitete Einstellung zurückführen lässt, dass Prozessmodelle nur für „Anfänger“ nützlich seien. In den Beiträgen von Fitzgerald (1996, 1998b,a) finden sich weitere Argumente gegen die Verwendung von Methodiken, wie beispielsweise die Unterbindung von individueller Kreativität und Intuition. Andererseits aber beschreiben Iivari u. a. (2001) in ihrer Arbeit ein „Wiederaufleben“ des Interesses an Methodiken und methodischen Ansätzen, was sich zum Teil aus der Evolution der IT-Plattformen und Technologien erklärt – z.B. Client/Server-Architekturen, Internetanwendungen und Computer Supported Cooperative Work (CSCW) Werkzeugen. Auch der Siegeszug der Objektorientierung, der Standards der OMG und integrierte Lösungen, bestehend aus Computer Aided Software Engineering(CASE) Werkzeugen und Vorgehensmodellen (z.B. der Unified Process und XDE von Rational), haben ein neue Welle von Methodiken zur Folge gehabt (Iivari u. a. (2001)).

Wie auch schon in Abschnitt 2.2 festgestellt wurde, ist der Einsatz von disziplinierten methodischen Ansätzen eine logische Konsequenz aus der wachsenden Komplexität der USW. Daher wird auch der Prozess zu deren Erstellung sehr komplex, was eine Unterteilung in einzelne, beherrschbare Schritte, wie bei anderen komplexen Produkten, nach sich zieht. Außerdem erleichtert die so gewonnene Transparenz das Projektmanagement und die Kontrolle des Prozesses, was zu einer Reduktion von Risiken und Unsicherheiten führen kann. Fitzgerald (1998a) sieht des Weiteren eine ökonomische und eine epistemologische Begründung für die Anwendung von Methodiken: zum einen die implizierte Spezialisierung und Arbeitsteilung, die eine variable Vergütung der Prozessbeteiligten ermöglicht und zum anderen das strukturelle Rahmenwerk zur Wissensbeschaffung und -wiederverwendung, welches durch Prozessmodelle geschaffen wird (siehe Abschnitt 4.4). Gefahren sieht er u.a. in der teilweise weder konzeptionell noch empirisch fundierten Generalisierung von Methodiken, der Vernachlässigung von sozialen bzw. soziotechnischen Aspekten und der mangelnden Berücksichtigung der Eventualitäten jeder einzelnen Problemsituation. Auch darf bei der Einhaltung von Prozessvorschriften nie der Blick für das Wesentliche, nämlich die Erstellung von USW, verloren gehen. Versteegen (2000) nennt in seiner Arbeit konkrete Kenngrößen, welche den Einsatz von Prozessmodellen rechtfertigen und sich hauptsächlich auf das beteiligte Projektteam und die Projektart beziehen. In Abschnitt 5.2.1 werden im Rahmen der Analyse von Kollaborationstechniken agile Vorgehensweisen zur Softwareerstellung näher betrachtet, welche sich als Gegenbewegung zu den etablierten, stringenten Vorgehensmodellen verstehen (siehe auch Agile Alliance (2001)).

In den folgenden Abschnitten sollen zunächst acht fortgeschrittene, auf Wiederverwendung hin optimierte Prozessmodelle charakterisiert und entlang des Vergleichsrahmens detailliert betrachtet werden. Darauf aufbauend werden abschließend die beiden aktuellsten und am meisten etablierten Prozessmodelle einer eingehenden Analyse unterzogen und gegenübergestellt. Da hierbei ist ein genealogischer Zusammenhang zwischen einzelnen Modellen feststellbar, dienen die zuvor dargestellten klassischen Modelle als Grundlage für die weiteren Betrachtungen.

3.2 Fortgeschrittene Vorgehensmodelle

Durch die Analyse moderner Vorgehensmodelle wird versucht, bereits kodifizierte Best-Practices bei der Softwareerstellung festzuhalten und im Hinblick auf einen eigenen Referenzprozess zu konsolidieren. Das Erkennen von Schwächen bei existierenden Methodiken kann ebenfalls wertvolle Erkenntnisse für das laufende Forschungprojekt liefern. Die hier analysierten Modelle wurden sowohl anhand ihrer praktischen Bedeutung als auch der Qualität der entsprechenden Publikationen ausgewählt. Zusätzlich zur Anwendung des Vergleichsrahmes wurde bei einigen Vorgehensmodellen eine

NIMSAD-Analyse durchgeführt, um grundlegende Eigenschaften der Methodik herauszustellen (vgl. Abschnitt 2.1.3 bzw. Anhang 1). Diese eingehende Analyse der existierenden methodischen Ansätze erleichtert die Bewertung hinsichtlich deren Eignung als Prozessrahmen für kollaborative, zwischenbetriebliche Entwicklungsaktivitäten und Beziehungen.

3.2.1 V-Modell'97

„*Das V-Modell [Vorgehensmodell] ist ein anerkannter Entwicklungsstandard für IT-Systeme [EstdIT], der einheitlich und verbindlich festlegt, was zu tun ist, wie die Aufgaben durchzuführen sind und womit dies zu geschehen hat*" (Industrieanlagen-Betriebsgesellschaft (1997), S. 1).

Dieser Zusammenhang von Vorgehensmodell (was?), Methodenzuordnung (wie?) und funktionalen Werkzeuganforderungen (womit?), aus dem sich auch die Gliederung der Prozessdokumentation ableitet, ist noch einmal in Abbildung 4.12 dargestellt. Aus dem Schaubild lassen sich auch die vier grundlegenden Submodelle des V-Modells, Softwareerstellung (SE), Qualitätssicherung (QS), Konfigurationsmanagement (KM) und Projektmanagement (PM) erkennen, wobei QS, KM und PM die bereits identifizierten Querschnittsfunktionen zum eigentlichen Erstellungsprozess darstellen (vgl. Abschnitt 3.2.7).

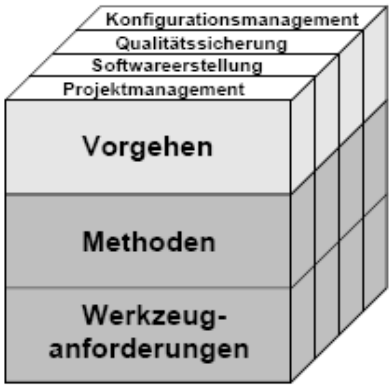

Abbildung 6: Submodelle und Struktur des V-Modells

Philosophie

Balzert (1998) bezeichnet das V-Modell'97 als „Erweiterung" des Wasserfallmodells (vgl. Abschnitt 3.4.1), und zwar um die Aspekte Qualitätssicherung, Verifikation und Validierung, wobei das primäre Ziel in der Erreichung maximaler Qualität (Safe-to-Market) besteht. Aus dieser Charakteristik ist auch die Bezeichnung als „kontrolliertes Prototyping" erwachsen (siehe auch Abschnitt 3.4.2). Es ist somit besonders für große Projekte im Bereich sicherheitskritischer Software, wie z.B. eingebetteter Systeme, geeignet. Außerdem verfolgt die noch aktuelle Version von 1997 zusätzlich zur ersten Veröffentlichung die Unterstützung inkrementeller Entwicklungsprozesse und den Einsatz von objektorientierten Techniken (siehe dazu Dröschel u. a. (1998) und Müller-Ettrich (1999)), was in der kommenden Version von 2004 gezielt weiter in Richtung der UML ausgebaut werden soll.

Terminologie

Bei der Wahl der Begriffe im V-Modell'97 fallen vor allem viele unnötige englische Bezeichner (z.B. Tailoring) und unübliche deutsche Ausdrücke, wie beispielsweise Forderungen statt Anforderungen, unangenehm auf (vgl. hierzu Fettke und Loos (2002) und Abschnitt 4.4.9). Als einziges rein deutschsprachiges Prozessmodell zusammen mit

Booster nimmt es terminologisch ohnehin eine Sonderstellung unter den analysierten Modellen ein.

Modellumfang

Das Modell verfügt über eine äußerst umfangreiche Dokumentation, bestehend aus den drei oben genannten Bänden „Vorgehensmodell“, „Methodenzuordnung“ und „Werkzeuganforderungen“ sowie einem Hypertextsystem mit Animationen, welche zusammen eine integrierte und detaillierte Darstellung der vier Submodelle bieten (Dröschel und Wiemers (2000)). Es lässt sich daher klar als „schwergewichtiges“ Modell klassifizieren und deckt alle in Abschnitt 3.2.6 beschriebenen Phasen der Softwareentwicklung von der „System-Anforderungsanalyse“ bis hin zur „Überleitung in die Nutzung“ samt der wesentlichen prozessbegleitenden Querschnittsfunktionen (PM, KM und QS) in der vollen Breite ab (Phasen- und Prozessabdeckung). Überdies definiert das Modell 25 unterschiedliche Rollen, was es wiederum für den Einsatz in großen Projekten qualifiziert und die Anwendung in kleiner angelegten Projekten erschwert (Balzert, 1998).

Prozessarchitektur

Wie bereits erwähnt, sind die Aktivitäten innerhalb des V-Modells auf oberster Ebene in die vier Submodelle PM, KM, QS und SE untergliedert. Die Aktivitäten, welche der eigentlichen Softwareentwicklung dienen, lassen sich, wie in Abbildung 4.13 dargestellt, V-förmig anordnen. Man unterscheidet konstruktive Aktivitäten links der „Spiegelachse“ (SE 1-6) und integrative Aktivitäten (SE 6-9) auf der rechten Seite. Die Architektur an sich ist unabhängig bezüglich der Elementarmethoden, Techniken und somit auch vom grundsätzlichen Ansatz (z.B. strukturierte Programmierung oder Objektorientierung, vgl. Abschnitt 3.3.1). Es handelt sich aufgrund der detailliert beschriebenen Arbeitspläne um ein primär aktivitätsorientiert gesteuertes Prozessmodell. überdies wird die Bearbeitung der „Produkte“ in den einzelnen Aktivitäten sehr genau spezifiziert („Produktfluss“), d.h. die Prozesssteuerung weist ebenfalls eine ergebnisorientierte Komponente auf. Ein Beispiel für diese Dualität wäre, dass die Produkte „Software-Entwurf“ und „Datenkatalog“ durch die Aktivität „Software-Feinentwurf“ (SE 5, siehe Abb. 4.13) in den Zustand „akzeptiert“ überführt werden müssen.

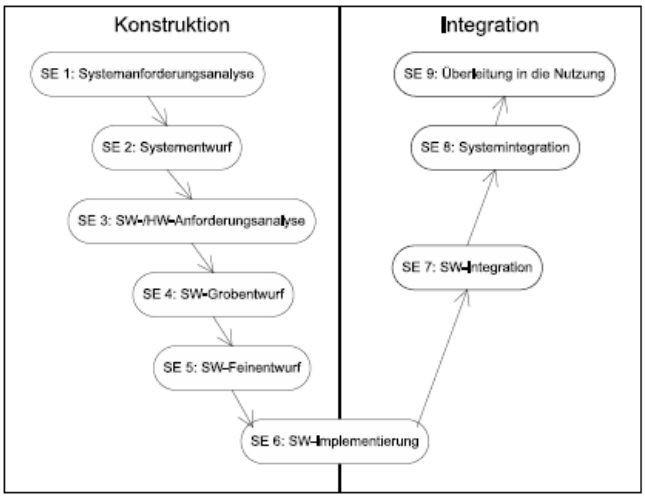

Abbildung 7: Das Submodelle SE als Aktivitätsdiagramm

Komponentenverständnis

Komponenten im Sinne dieser Arbeit werden beim V-Modell als „Fertigprodukte“ und „Halbfabrikate“ bezeichnet und sind aufgrund der nachträglichen Evolution des Modells hin zur komponentenbasierten Wiederverwendung bisher wenig konkretisiert. Doch

werden kommerzielle Fertigprodukte, welche auf dem freien Markt bezogen werden können, und solche, die innerhalb der entwickelnden Organisation vorliegen, unterschieden. Obwohl die Autoren eine „breite“ Komponentendefinition auf allen Ebenen der Erzeugnisstruktur, vom Quelltext bis hin zur Architektur- oder Anforderungsebene, proklamieren, werden im eigentlichen Prozessmodell dann doch nur kommerzielle Komponenten für die Implementierungsphase berücksichtigt (Dröschel und Wiemers (2000)). Das Modell vertritt überdies einen Top-Down-Ansatz bei der Komponentenbildung und deckt den Komponentenlebenszyklus (siehe Abbildung 4.5) nur rudimentär ab (vgl. Fettke und Loos (2002)).

Kollaborationspunkte

Der Endanwender der Software wird beim V-Modell lediglich in der Definitionsphase (Anforderungsanalyse) eingebunden, ansonsten laufen die Prozesse weitgehend innerhalb der softwareentwickelnden Organisation ab. Es ist auch bei der eigentlichen Softwareerstellung keinerlei unternehmensübergreifende Zusammenarbeit (Kollaboration) vorgesehen.

Adaptionsmöglichkeiten

Da das Modell „Anspruch auf Allgemeingültigkeit“ (Balzert (1998), S. 109) erhebt, ist von vornherein ein Prozess zum „Maßschneidern“ (Tailoring) vorgesehen, welcher in den beiden Stufen „Ausschreibungsrelevantes Tailoring“ und „Technisches Tailoring“ abläuft. Beim Tailoring kann eine Untermenge der Standard- V-Modell-Konzepte gebildet werden, um die jeweilige Prozess- bzw. Projektstruktur adäquat abbilden zu können (vgl. hierzu Balzert (1998)). Mit dem V-Modell werden sechs verschiedene Szenarien wie beispielsweise „Inkrementelle Entwicklung“, „Grand Design“ oder „Objektorientierte Entwicklung“ ausgeliefert. Für die Ziele der vorliegenden Arbeit ist vor allem das Szenario „Einsatz von Fertigprodukten [Komponenten]“ interessant, welches auf der Idee beruht, den „Entwicklungsaufwand durch Nutzung bereits verfügbarer Bausteine (SW und HW) zu reduzieren“ (Dröschel und Wiemers (2000), S. SZ-11).

Das für Ende 2004 geplante „V-Modell XT soll den Aspekt der „Anwendbarkeit, Erweiterbarkeit und Anpassbarkeit“ (Rausch u. a. (2004)) durch die Bereitstellung eines „Baukastens“ aus so genannten „Vorgehensbausteinen“ (Prozesskomponenten) weiter verbessern (vgl. Abbildung 1.1).

Werkzeugunterstützung

Da die komplexen Abläufe und Produktdefinitionen sehr genau eingehalten werden müssen, ist das Modell ohne entsprechende Unterstützung durch Entwicklungs- und Managementwerkzeuge nicht zu realisieren. Aus diesem Grund beschäftigt sich einer der drei Bände der Prozessdokumentation ausschließlich mit den funktionalen Eigenschaften von in Frage kommenden Tools. Die Originaldokumentation inklusive umfassender Erläuterungen findet man bei Dröschel und Wiemers (2000) oder Industrieanlagen-Betriebsgesellschaft (1997)6. Weitere Bewertungen des Modells wurden u.a. von Balzert (1998), Noack und Schienmann (1999), Versteegen (2000) sowie Fettke und Loos (2002) vorgenommen.

3.2.2 OPEN Process

Genau wie der UP, welcher im folgenden Abschnitt ausführlich behandelt wird, stellt der Object-Oriented Process, Environment and Notation (OPEN)7 ein objektorientiertes Prozessrahmenwerk oder auch Metaprozessmodell dar, welches aus der Konsolidierung mehrerer objektorientierter Methoden entstanden ist (siehe auch Atkinson u. a. (2002)) und mit dessen Hilfe dann passende Prozessmodelle für die jeweilige Entwicklungsorganisation bzw. unterschiedliche Projekte instantiiert und implementiert werden können (siehe Abbildung 8).

Philosophie

Abgesehen von seinem generischen Charakter zeichnet sich OPEN Process durch sein vertragsgetriebenes Vorgehensmodell und seine anfangs „proprietäre“ Modellierungssprache OML (Object Modeling Language) aus, die mittlerweile durch die UML substituiert wurde (vgl. hierzu Henderson-Sellers und Unhelkar (2000)).

Terminologie

Bezeichnend für OPEN ist, dass die grob granularen Phasen hier Activities genannt werden, und die eigentlichen Aktivitäten, wie sie im Rahmen dieser Arbeit definiert wurden, als Tasks bzw. Subtasks tituliert werden. Techniken, Richtlinien und Standards werden alle unter „Tools and Techniques“ subsumiert (siehe auch Abschnitt 3.3.3). Das zugrunde liegende Prozessmetamodell von OPEN ist auch bei Henderson-Sellers (2003) noch einmal beschrieben.

Modellumfang

OPEN verfügt über eine äußerst umfangreiche Dokumentation des Prozessmodells, worin u.a. Instanzen von 30 Aktivitäten, 57 Tasks, 200 Techniken und 76 Rollen im Rahmen eines erweiterbaren Repository für Prozesskomponenten spezifiziert sind, und lässt sich daher ebenfalls bei den „Schwergewichten“ der Prozessmodelle einreihen (siehe hierzu Graham u. a. (1997), Henderson-Sellers u. a. (1998) sowie Firesmith und Henderson-Sellers (2002) und Henderson-Sellers (2003)). Wie auch beim V-Modell werden alle wesentlichen Phasen abgebildet, lediglich bei der Prozessabdeckung vermisst man einen expliziten Testprozess, Techniques zum Testen von Software sind jedoch spezifiziert (Graham u. a. (1997), S. 141ff.).

Prozessarchitektur

Die Architektur von OPEN besteht aus einem projektspezifischen und einem projektübergreifenden Teil, was in Abbildung 4.14 mittels der grauen Schattierung kenntlich gemacht wurde (vgl. Graham u. a. (1997), S. 46). In der Aktivität „Build-Phase“ wurde der evolutionäre Entwicklungsprozess, bestehend aus objektorientierter Analyse (OOA), Design (OOD) und Programmierung (OOP), sowie Verifikation und Validierung zusammengefasst (vgl. Abschnitt 3.4.2 bzw. 3.4.4). Akzeptanztests (User Reviews) und Konsolidierungsaktivitäten noch innerhalb der Build-Phase stellen auch einen Bezug zu früheren Projekten her und sollen wieder verwendbare Komponenten identifizieren. überdies sieht das Modell Aktivitäten zur Unternehmens- bzw. Domänenmodellierung vor, um die Anforderungsanalyse mit fachlichen Informationen zu unterstützen (vgl. dazu Abschnitt 2.3). Wie bereits weiter oben erwähnt, erfolgt die Prozesssteuerung bei OPEN über ein vertragsorientiertes Modell mit Voraussetzungen, Vervollständigungskriterien und Zeitbeschränkungen (Timeboxes, Graham u. a. (1997)).

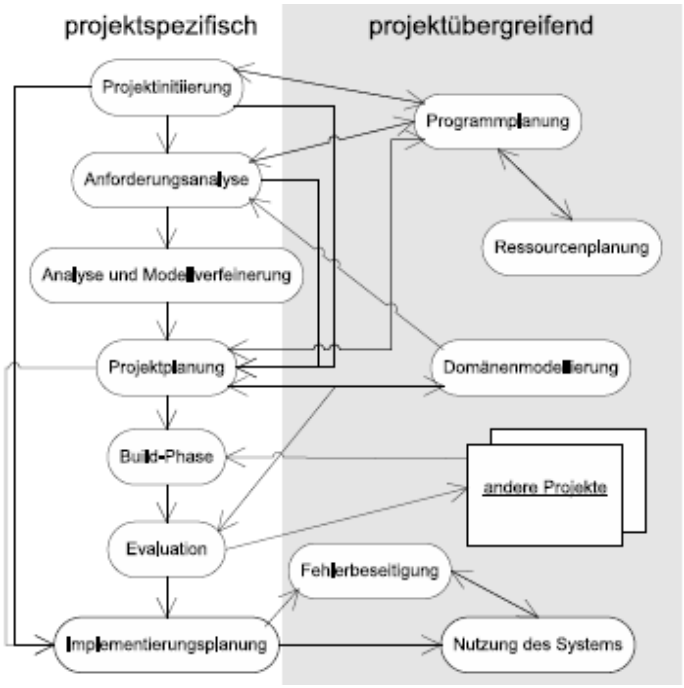

Abbildung 8: Das Lebenszyklusmodell von OPEN (Graham u. a., 1997)

Komponentenverständnis

Obwohl bei der Konsolidierung versucht wird, wieder verwendbare Komponenten zu identifizieren, ist der Komponentenaspekt bei Graham u. a. (1997) noch nicht explizit verankert. Dieses Defizit in puncto Wiederverwendungsprozess versuchen neuere Versionen von OPEN mittlerweile zu beheben (siehe dazu Firesmith und Henderson-Sellers (2002)).

Kollaborationspunkte

Zusätzlich zur Anforderungsanalyse werden die Anwender auch in der Build-Phase durch User Reviews mit in den Prozess eingebunden. Die Aktivität „Domänenmodellierung“ und der Abgleich mit früheren Projekten ermöglichen eine projektübergreifende Kollaboration in Form von asynchronem Wissenstransfer.

Adaptionsmöglichkeiten und **Werkzeugunterstützung**

Da es sich um ein flexibles Framework handelt, können Anpassungen durch geeignete Instantiierungen erfolgen. Außerdem sind Aktivitäten und Ergebnisse durch Modalitäten (optional und obligatorisch) gekennzeichnet und können daher zum Teil ausgelassen werden (vgl. Tailoring beim V-Modell). In Frage kommende CASE-Werkzeuge werden lediglich exemplarisch genannt. Intendiert sind marktübliche objektorientierte Analyse- und Entwurfswerkzeuge. Eine genaue funktionale Spezifikation der Werkzeuge wie beim V-Modell existiert nicht.

3.2.3 Unified Software Development Process

„*In fact, it is one of the constants of software development that the requirements change*” (Jacobson u. a. (1999), S. 9).

Eine der Grundideen beim Entwurf des Unified Software Development Process, oder auch nur Unified Process (UP), war das iterative Einbeziehen sich im Projektverlauf dynamisch ändernder Anforderungen an die zu erstellende Software (siehe Abschnitt 3.4.2). Wie auch bei OPEN handelt es sich beim UP um ein Metarahmenwerk zur Konstruktion bzw. Instantiierung von individuellen Prozessmodellen („Generic Process

Framework“, Jacobson u. a. (1999), S. 4). Zusammen mit der Unified Modeling Language (UML) entstammen die Ideen für das Prozessrahmenwerk der Zusammenführung (Unification) zahlreicher objektorientierter Methoden und Techniken Mitte der neunziger Jahre durch die „drei Amigos“ Grady Booch (Booch u. a. (1999)), Ivar Jacobson (Jacobson u. a. (1999)) und James Rumbaugh (Rumbaugh u. a. (1999)).

Der Rational Unified Process (RUP) wiederum stellt eine kommerzielle Instanz des UP dar, die teilweise immer noch Metamodellcharakter hat, und gleichzeitig die in der Praxis am häufigsten genutzte Variante ist (siehe Kruchten (2004)). Daher wird bei der folgenden Analyse kein Unterschied zwischen UP und RUP gemacht, außer wenn dies explizit erwähnt wird. Abbildung 4.15 zeigt eine mögliche Ausprägung des UP als inhaltlich gewichtetes Phasenmodell.

NIMSAD-Analyse

Die der eigentlichen Analyse vorgeschaltete NIMSAD-Betrachtung soll den generellen Charakter der komplexen UP-Methodik herausarbeiten und einen Rahmen für die anschließende Vergleichsanalyse bilden (vgl. Abschnitt 3.3.2 und Anhang A). In puncto Methodikkontext ist der UP laut eigenen Angaben für alle Anwendungssituationen in der SWT geeignet (Jacobson u. a. (1999)). Als Startpunkte für den Prozess gelten frühere Systeme und Pläne sowie die Anforderungen der späteren Systemanwender. Die Kontextbeschreibung findet über UML-basierte Unternehmens- und Domänenmodelle mit entsprechenden Geschäftsanwendungsfällen statt. Risiken bei der Analyse der Problemsituation bestehen in der zu genauen Beschreibung und in der Weitergabe des Methodenwissens der Prozessanwender (Forsell u. a. (1999)). Bei den Anwendern der UP-Methodik wird schon früh im Entwicklungszyklus technische Rationalität und ein hohes abstraktes Denkvermögen vorausgesetzt. Es werden ebenfalls umfassende Methodik- und Domänenkenntnisse erwartet, wobei vor allem erstere lange Einarbeitungszeiten und eine steile anfängliche Lernkurve zur Folge haben können. Die Methodik (Problemlösungsprozess) an sich bietet ein umfassendes Instrumentarium zur Problemformulierung und zum Lösungsentwurf. Zur Festlegung der Problemgrenzen bzw. Systemdiagnose sind zahlreiche Aktivitäten und Artefakte in den Disziplinen „Unternehmensmodellierung“ und „Anforderungen“ definiert, u.a. das Unternehmens- bzw. Domänenmodell und die Anforderungsspezifikation. Die Prognose für das zu erstellende System wird in der „Vision“ festgehalten und in ersten Anwendungsfällen und Architekturbeschreibungen konkretisiert. Beim Lösungsentwurf – sowohl logisch als auch physisch – kommt die volle Bandbreite der UML-Artefakte zum Einsatz, deren Erstellung über die Aktivitäten der Disziplin „Analyse und Entwurf“ koordiniert wird. Die Umsetzung der Entwürfe wird in den Arbeitsabläufen „Implementierung“, „Test“ sowie „Installation und Einsatz“ beschrieben, wobei die beiden letzteren hauptsächlich in der Phase „Transition“ (zum Kunden) ausgeführt werden (siehe Abbildung 4.15). Für die einzelnen zeitlichen Phasen sind jeweils a posteriori Evaluationskriterien zum Erreichen der Teilziele (Milestones) spezifiziert (siehe Kruchten (2004)).

Philosophie

Die drei Hauptmerkmale des UP sind, dass er anwendungsfallgetrieben, architekturzentriert, iterativ und inkrementell ist (siehe hierzu Abschnitt 3.4.2). Anwendungsfälle (Use-Cases) stellen von der Unternehmensmodellierung bis zum finalen Systemtest den „roten Faden“ des Prozessmodells dar (siehe auch Cockburn (2001)). Durch Geschäftsanwendungsfälle werden fachliche Anforderungen spezifiziert und mittels Systemanwendungsfällen in der Analyse- und Entwurfsphase auf IT-Ebene weiter konkretisiert (Oestereich (2003)). Testanwendungsfälle dienen dann der Verifikation und Validierung des Systems im Hinblick auf die zu erfüllenden Anforderungen. Des Weiteren zeichnet der UP sich dadurch aus, dass er sehr eng an die Konzepte und Notationen des UML-Standards gebunden ist und durch den Architekturfokus von vornherein auf

Wiederverwendung in Form von Komponenten ausgerichtet ist. Die in Abschnitt 4.3.3 beschriebene Model-Driven Architecture der OMG soll diesen Wiederverwendungsaspekt um weiter Abstraktions- und Generierungstechniken (MDD) erweitern.

Terminologie

Die von Jacobson u. a. (1999) verwendeten Bezeichnungen, wie beispielsweise Arbeiter (Worker) und Artefakt (Artifact) haben sich mittlerweile durch die große Verbreitung der Methodik zu De-facto-Standards in der SWT entwickelt (siehe hierzu auch Tabelle 4.1). Dies liegt nicht zuletzt auch an der engen Verbindung mit der UML, die im Bereich der objektorientierten Modellierungssprachen praktisch konkurrenzlos ist.

Modellumfang

Allein die HTML-Dokumentation des Rational UP misst in der aktuellen Version „2003.06.01“ knapp 25 MB an Datenvolumen. Es existieren zahlreiche Monografien, die versuchen, das Prozess-Framework zu beschreiben, doch scheint aufgrund der Komplexität und der mehrdimensionalen Prozessarchitektur Hypertext am besten geeignet zu sein. Der Original-UP wird bei Jacobson u. a. (1999) in Form einer umfangreichen Monografie vorgestellt und die wohl bekannteste Literatur zur Einführung des RUP stammt von Kruchten (2000, 2004). Die grundlegenden Phasen und Prozess-Querschnittsfunktionen werden vom UP/RUP lückenlos abgedeckt und spiegeln das Destillat der bewährten Praktiken (Best-Practices) aus der SWT wieder (siehe auch Prozessarchitektur und Abbildung 4.15). Der RUP expliziert folgende sechs etablierte Praktiken als essenziell: iterative Entwicklung, Anforderungsmanagement, Komponentenarchitekturen, visuelle Modellierung, kontinuierliche Qualitätsprüfung und änderungsmanagement (siehe RUP-HTML-Dokumentation). Auch das Rollenmodell ist mit 30 definierten Rollen in fünf unterschiedlichen Gruppen (Analysten, Entwickler, Tester, Manager und sonstige Worker) sehr umfassend (Noack und Schienmann (1999)).

Prozessarchitektur

Der Prozessrahmen gliedert sich beim UP in inhaltlich verschiedene Prozessabläufe (Workflows) und zeitlich geordnete Phasen, welche wiederum in einzelne Iterationen unterteilt sind (siehe 4.15). Bei den Prozessabläufen unterscheidet man die Kernprozessabläufe der eigentlichen Softwareentwicklung und die Kernunterstützungsabläufe, welche teilweise mit den in Abschnitt 3.2.7 beschriebenen Querschnittsfunktionen korrespondieren – Qualitätssicherung wird durch den zusätzlichen Kernprozessablauf „Test“ repräsentiert. In Abbildung 4.15 ist eine Instanz des UP, der bereits erwähnte RUP, dargestellt, welche die beschriebenen Architekturelemente beinhaltet und die Gewichtung der einzelnen Workflows in der Iterationsfolge anhand der jeweiligen Kurvenhöhe visualisiert. Diese Ausgestaltung der einzelnen Workflows kann projektindividuell angepasst werden. „Prinzipiell kann der RUP als eine Variante des Spiralmodells [...] verstanden werden, die inhaltlich detaillierter ausformuliert wurde und [...] optisch anders dargestellt ist“ (Fettke und Loos (2002), S. 36). Jeder Workflow wird beim RUP durch ein Aktivitätsdiagramm näher beschrieben. Diese genaue Spezifikation der Aktivitätenfolge hat einen aktivitätsorientierten Prozessfluss zur Folge (siehe Kruchten (2004) und HTML-Dokumentation). Ergebnisse (Artefakte) hingegen können den Aktivitäten nicht immer eindeutig zugeordnet werden (vgl. Fettke und Loos (2002)).

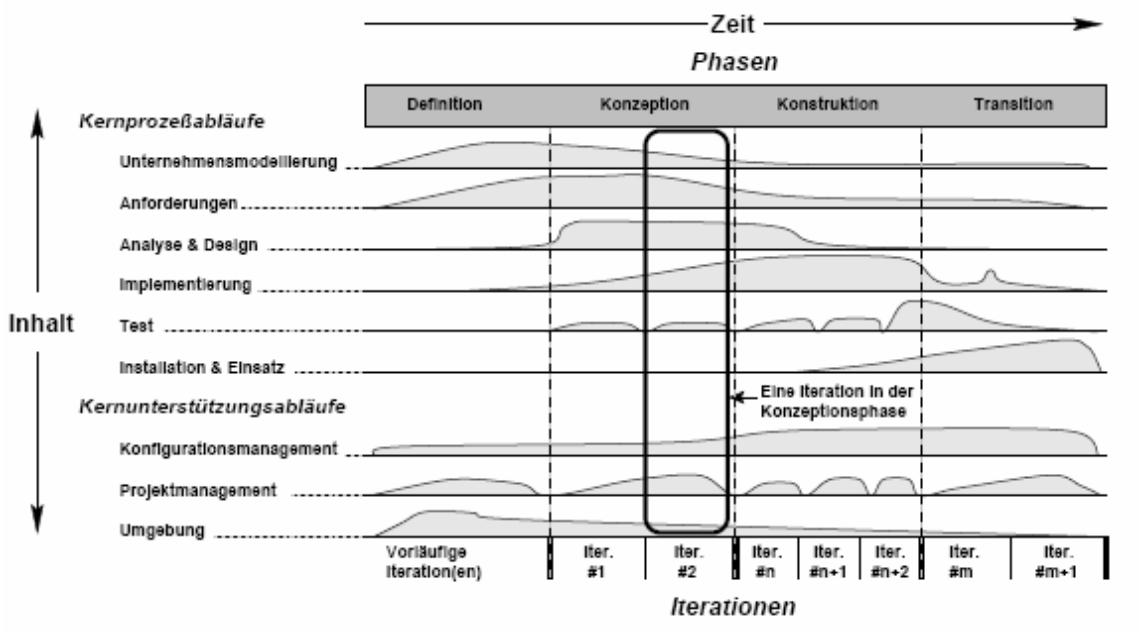

Abbildung 9: Beispielinstanz eines Unified Process (RUP, Kruchten, 2004)

Komponentenverständnis

In der aktuellen Version des RUP werden Laufzeitkomponenten (Runtime Components), Komponenten aus Entwicklersicht (Development Components) und Geschäftskomponenten (Business Components) unterschieden (siehe Kruchten (2000)). Mit der Dokumentation wird bereits eine Variante (Conceptual Roadmap) ausgeliefert, welche auf die KBSE hin angepasst wurde (Developing Component Solutions). Das Modell suggeriert eine Top-Down-Herangehensweise zur Komponentenbildung und es werden die Komponentenlebenszyklusphasen „Entwicklung“ (Process Workflows), „Wiederauffindung“ (Teil der Disziplin Projektmanagement) und „Komposition“ (Deployment) explizit abgedeckt (vgl. Abbildung 4.5).

Kollaborationspunkte

Durch die fein granularen Iterationen und die inkrementelle Vorgehensweise kommt zu einem häufig auftretenden Abgleich der Anforderungen mit dem Anwender. Dies hat eine bessere Integration der Auftraggeber und eine Minimierung des Projektrisikos zur Folge (siehe Spiralmodell in Abschnitt 3.4.2). überdies ermöglicht die genaue Spezifikation der Aktivitäten, Artefakte und Rollen generell eine methodische, arbeitsteilige Vorgehensweise. Unternehmensübergreifende Zusammenarbeit in Projekten wird nicht explizit durch entsprechende Workflows bzw. Aktivitäten unterstützt.

Adaptionsmöglichkeiten

Zusätzlich zu den bereits erwähnten, konzeptuellen Roadmaps als fertige Prozessvarianten, besteht die Möglichkeit, die Prozesselemente des UP zu erweitern und vor allem aber auch nicht relevante Teile auszulassen (Tailoring). Es ist wegen der hohen Komplexität der Zusammenhänge und Abhängigkeiten für die meisten Projekte sogar zwingend notwendig, den Standardprozess zu „beschneiden“. Das Prozessmodell skaliert daher gut für unterschiedlich große Projekte, aber auch der anfängliche Aufwand des Zurechtschneidens bei kleineren Projekten ist nicht zu unterschätzen.

Werkzeugunterstützung

Bereits aus der Hypertext-Dokumentation heraus, finden sich Verweise und Links zu den Werkzeugen der Firma IBM/Rational8. Rational XDE (früher: Rational Rose) ist eine auf der Open Source-Plattform Eclipse9 aufbauende integrierte Entwicklungsum-

gebung, die speziell auf den UP/RUP zugeschnitten und erweitert wurde. Wesentlicher Bestandteil ist die Unterstützung der Modellierungssprache UML mit den einzelnen Diagrammarten, Modellelementen und Notationen. Weitere Rational Werkzeuge sind beispielsweise Clearcase und Clearquest (siehe Piwecki und Schmidt (2004)) für das Konfigurations- und Änderungsmanagement und Requisite Pro für das Anforderungsmanagement im Entwicklungsprozess.

3.2.4 Perspective

Das Vorgehensmodell Perspective wurde ursprünglich 1994 von der Firma Select heute: Aonix10) entworfen und seitdem kontinuierlich weiterentwickelt. In der zweiten Version von Allen und Frost (1998) wurde die KBSE als zentrales Merkmal des Modells integriert und von Apperly u. a. (2003) weiter um Aspekte der serviceorientierten Entwicklung mit Web Services (Service-Oriented Architecture, SOA) ergänzt.

Philosophie

"*Perspective is a collection of industry best-practice modeling techniques that are applied and adapted using process templates within an architectural framework across a wide range of developments in a component-based setting*" (Allen und Frost (1998).

Charakteristisch für Perspective ist vor allem der starke Praxisbezug durch die Anwendung in bis dato zahlreichen Projekten und die starke Fokussierung auf die letztendliche Auslieferung der Softwarelösung („delivery-based approach", Apperly u. a. (2003), S. 120). Ebenfalls differenziert sich Perspective durch eine strikte Trennung der Prozesse zur Erstellung unternehmensspezifischer Softwaresysteme (Solutions bzw. USW) einerseits und der Komponentenentwicklung andererseits. Die Idee dahinter ist die strikte organisatorische Separation von Komponentenbereitstellung (Supply), Komponentenwiederverwendung (Consume) und den entsprechenden Managementaktivitäten (Supply-Manage-Consume-Modell, vgl. Apperly u. a. (2003)).

Terminologie

Bei der verwendeten Terminologie ist lediglich auffällig, dass Phasen als Stages und Aktivitäten als Tasks bezeichnet werden. Dokumentierte Richtlinien werden als Practical Guidelines tituliert (vgl. Tabelle 4.1).

Modellumfang

Mit der umfangreichen Hypertext-Dokumentation und der fast 500-seitigen Monografie von Allen und Frost (1998) lässt sich die Prozessbeschreibung durchaus als umfangreich und die Methodik als „schwergewichtig" klassifizieren. Die sechs grundlegenden Phasen des Softwarelebenszykluses werden durch die sieben Stages des Perspective Solution Process abgedeckt. Besonders zu erwähnen ist die Machbarkeitsstudie (Feasibility, siehe Abbildung 4.16) zu Beginn jedes Zykluses, die analog zum Risikomanagement im Spiralmodell durchgeführt wird (vgl. Abschnitt 3.4.2). Bei der Abdeckung des gesamten Prozesses fällt bezüglich der Querschnittsfunktionen auf, dass Perspective keine Aktivitäten für das änderungs- und Konfigurationsmanagement spezifiziert (siehe Allen und Frost (1998)). Teilweise wird die Konfigurationsverwaltung jedoch auf Komponentenebene unterstützt (vgl. Apperly u. a. (2003)). Der Teamarbeit kommt bei diesem Modell eine große Bedeutung zu, wobei innerhalb der Arbeitsgruppen immer ein ausgewogenes Verhältnis von technisch und fachlich ausgerichteten Rollen herrschen soll. Das Rollenmodell umfasst insgesamt 21 Rollen und unterteilt sich in die drei Bereiche Solution Process, Component Process und „technische Infrastruktur" (vgl. Allen und Frost (1998)). Perspective definiert dabei als einziges Prozessmodell vier Rollen speziell für Wiederverwendungsaktivitäten: Reuse Identifier, Reuse Librarian, Reuse Assessor und Reuse Architect.

Prozessarchitektur

Wie bereits erwähnt, teilt sich die Architektur von Perspective in den Solutions Process für die Anwendungsentwicklung und den Component Process für die Bereitstellung von Komponenten auf, wobei die Komponentenentwicklung als unabhängiges Prozessmodell betrachtet werden kann. Beide Prozessteile sind auf iterative und inkrementelle Entwicklung der Artefakte sowie eine möglichst parallele Ausführung der Aktivitäten ausgelegt. Im Komponentenentwicklungsprozess (Supplier) kommen eher generische USW-Anforderungen für wieder verwendbare Komponenten zum Tragen, wohingegen der Solution Process (Consumer) problemspezifischere Anforderungen abbilden soll (Apperly u. a. (2003)). Die „unteren" vier Phasen in Abbildung 4.16 sind bei System- und Komponentenerstellungsprozess analog, wobei ein Zyklus ausgehend von der Planung über „Entwurf und Bau", Akzeptanztest und „Einführung" einer Iteration im jeweiligen Prozessmodell entspricht. Nachdem bei der „Einschätzung" (Assessment) im Component Process überprüft wurde, welche Dienste von den Solution-Prozessen anderer Projekte nachgefragt werden, gibt es grundsätzlich zwei „Pfade" hin zur Planung der Komponentendienste: Upgrade existierender Komponenten (unterer Pfad) oder Komponentenneuerstellung (eingerahmte Phasen). Gesteuert werden die Abläufe sowohl von aktivitäts- als auch von ergebnisorientierte Aspekten. Zum einen wird jede Phase durch die durchzuführenden Aktivitäten und Unteraktivitäten beschrieben, andererseits werden aber zu jeder Aktivität auch die benötigten und zu erzeugenden Ergebnisse aufgeführt. Aufgrund der eindeutigen Zuordnung kann man in diesem Fall im Gegensatz zum UP/RUP von einer Mischform von Aktivitäts- und Ergebnisorientierung sprechen (siehe auch Noack und Schienmann (1999) bzw. Fettke und Loos (2002)).

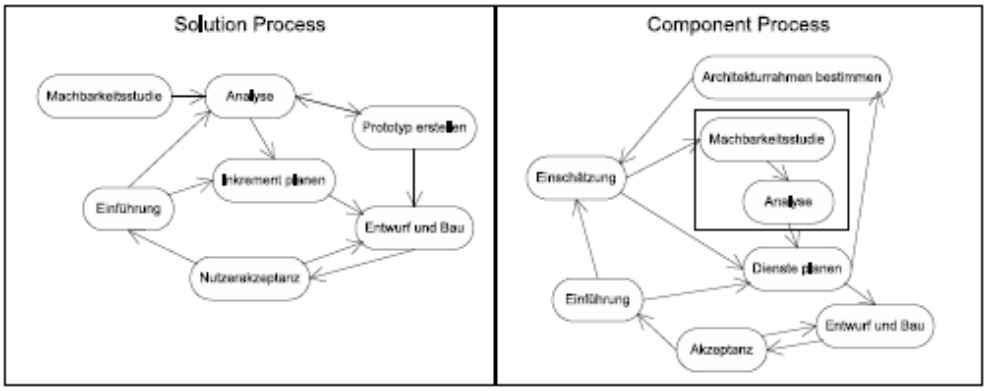

Abbildung 10: Prozessarchitektur von Perspective (Allen und Frost, 1998)

Komponentenverständnis

"*A component is an executable unit of code that provides physical black-box encapsulation of related services*" (Allen und Frost (1998), S. 4).

Bis auf die beiden Einschränkungen auf ausführbaren Code und reine Blackbox-Wiederverwendung entspricht der von Perspective betrachtete Komponentenbegriff im Wesentlichen dem dieser Arbeit zugrunde liegenden (vgl. Allen und Frost (1998) bzw. Abschnitt 4.2.1). Komponenten werden hier, ausgehend von zusammenhängenden Diensten, in einer Bottom-Up-Manier gebildet – ähnlich einem Clustering-Verfahren. Aus dem Komponentenlebenszyklus von Turowski (1999) (vgl. Abbildung 4.5) werden vor allem die Phasen „Komponentenentwicklung" (als eigenständiges Prozessmodell, Component Process), „Wiederauffindung" und „Komposition" explizit behandelt.

Kollaborationspunkte

Allen und Frost (1998) weisen darauf hin, dass sich das Vorgehensmodell für die parallele, und damit verteilbare, Entwicklung verschiedener Systeme oder Systemteile eignet und somit die Kollaboration bzw. „interaction between suppliers and consumers of

components“ (Apperly u. a. (2003), S. 11) ermöglicht. Dadurch kann dann eine Art „Komponentenlieferkette“ (vgl. Messerschmitt und Szyperski (2003)) oder „Community“ aus Komponentenlieferanten (Suppliers) und -konsumenten (Consumers) im Rahmen von Perspective „aufgespannt“ werden (Apperly u. a. (2003)).

Adaptionsmöglichkeiten

Laut Apperly u. a. (2003) ist Perspective „designed to fit most organizations“ (S. 13) und verfügt über vordefinierte Profile für unterschiedliche Projekttypen, wie z.B. Greenfield (völlige Neuentwicklung, d.h. „grüne Wiese“), Brownfield (inklusive Altlasten), Web-driven Development und Integration von Pauschallösungen (Packaged Solutions). Solution und Component Process können weitgehend unabhängig voneinander in verschiedenen Organisationen implementiert werden.

Werkzeugunterstützung

Wie auch beim RUP bietet die herausgebende Firma ein integriertes Softwarepaket zur Unterstützung des kompletten Vorgehensmodells an, und es werden dementsprechend viele Hinweise geliefert, „wie bestimmte Aktivitäten und Ergebnisse am besten mit den herstellereigenen Entwicklungswerkzeugen umgesetzt werden können“ (Noack und Schienmann (1999), S. 178). Die Firma Select bietet u.a. mit der Component Factory, dem Component Portal und dem Process Director zahlreiche Funktionen zur Umsetzung des Perspective-Prozesses als fertige Werkzeuglösungen an.

3.2.5 Catalysis

Der Ansatz von D’Souza und Wills (1999) unterscheidet sich grundsätzlich von allen bisher analysierten Prozessmodellen insofern, dass keinerlei konkrete Vorschriften über die Reihenfolge oder den Ablauf der zu durchlaufende Phasen bzw. Aktivitäten spezifiziert werden. Stattdessen stellen die Autoren so genannte „Prozessmuster“ (Process Patterns) vor, aus denen sich flexibel Prozesse für die KBSE zusammenstellen lassen. Wie auch beim UP basieren viele Aktivitäten auf einer besonders starken Einbeziehung der UML-Notationen.

NIMSAD-Analyse

Ähnlich auch dem UP, handelt es sich bei Catalysis mit seinen 48 Prozessmustern um ein sehr komplexes Vorgehensmodell und soll daher ebenfalls durch eine vorgeschaltete NIMSAD-Analyse strukturiert werden (vgl. Abschnitt 3.3.2). Vergleichbar zu Jacobson u. a. (1999) schreiben auch D’Souza und Wills (1999) ihrem Ansatz eine universelle Anwendbarkeit in allen Entwicklungssituationen zu. Als Startpunkte für die Erfassung der Problemsituation (Methodikkontext) werden sowohl Anforderungen der späteren USW-Nutzer als auch Ankündigungen für BPR-Maßnahmen aufgeführt. Den Geschäfts- und Domänenmodellen als Beschreibung des Kontexts für komponentenbasierte USW kommt hierbei eine große Bedeutung zu (vgl. D’Souza und Wills (1999), S. 543ff.). Risiken bei der Kontextbeschreibung und der letztendlichen Anwendung der Catalysis-Methodik werden von den Autoren hauptsächlich in der korrekten und eindeutigen Ermittlung der Anforderungen der Kunden gesehen. An die Methodiknutzer werden vor allem bei der Umsetzung der komponentenbasierten Modellierungstechniken hohe Anforderungen bezüglich ihres Abstraktionsvermögens gestellt, und ein fundiertes Domänenwissen, Erfahrung mit objektorientierter Modellierung sowie gute Methodikkenntnisse vorausgesetzt. Dies hat, ähnlich wie beim UP, eine anfänglich sehr steile Lernkurve für Erstanwender von Catalysis zur Folge. Bei der Problemformulierung soll zunächst die Problemsituation mittels Anwendungsfalldiagrammen, Kunden-Feedback und einem Verzeichnis (Dictionary) abgesteckt werden (Problem Boundaries). Zur Systemdiagnose schlagen D’Souza und Wills (1999) wiederum Anwendungsfälle, komponentenbezogene Type Models und System Context Models vor. Der Lö-

sungsentwurf besteht dann aus der Spezifikation der Abgrenzung von Menschen und Softwarekomponenten sowie der Komponenten untereinander (Komponentenspezifikation) und dem Entwurf der realisierenden Interna der einzelnen Komponenten (Komponentenrealisierung). Für die konkrete Umsetzung des Entwurfs (Implementierung) werden lediglich einige iterative Strategien genannt. Der Evaluationsaspekt fehlt bei Catalysis im Gegensatz zu UP/RUP gänzlich.

Philosophie

Catalysis charakterisiert sich vor allem durch den Aufbau aus Prozessmustern und die alleinige Ausrichtung auf die KBSE (vgl. Perspective). Es wird dabei auch schon von „Produktfamilien", d.h. unterschiedliche USW-Systeme mit teilweise identischen Geschäftskomponenten gesprochen (vgl. KobrA in Abschnitt 4.4.8). Des Weiteren lehnt sich der Prozess an den inkrementellen und iterativen Charakter des UP an und fordert eine vergleichsweise hohe formelle Qualität der System- und Komponentenspezifikation. Zudem ist eines der Grundprinzipien des Ansatzes die in Abschnitt 2.3 dieser Arbeit bereits erörterte Rückverfolgbarkeit bzw. Durchgängigkeit von Konzepten (Traceability) vom Unternehmensmodell bis hin zum lauffähigen Code (D'Souza und Wills (1999)).

Terminologie

Wie bereits angedeutet, existiert das Konzept der sachlogisch oder zeitlich strukturierten „Phasen" bei Catalysis nicht. Gleiches gilt für „Rollen", auch hierfür wird kein entsprechendes Konzept eingeführt. Die restliche Terminologie deckt sich im Wesentlichen mit der der anderen Prozessmodelle (vgl. Tabelle 4.1), was sich u.a. durch den engen Bezug zur UML, dem UP und der OMG erklärt.

Modellumfang

D'Souza und Wills (1999) liefern mit ihrer nahezu 800-seitigen Beschreibung des Prozessansatzes eine sehr umfangreiche und detaillierte Dokumentation (weitere Informationen sind über die Catalysis-Homepage11 abrufbar). Bei der Phasen bzw. Prozessabdeckung beschränkt sich das Modell ausschließlich auf Entwicklungstätigkeiten und geht auf Querschnittsfunktionen, wie beispielsweise Projektmanagement, nicht explizit ein. Ein ausdrückliches Rollenmodell wurde ebenfalls nicht spezifiziert, Aufgaben und Mitarbeiterprofile können aber teilweise aus den Prozessmusterbeschreibungen abgeleitet werden.

Prozessarchitektur

„Aufgrund der genannten Eigenschaften [Aufbau aus Prozessmustern] ist es nicht möglich, eine allgemeine Prozessarchitektur [von Catalysis] zu beschreiben" (Fettke und Loos (2002), S. 34). Catalysis setzt sich durch das fehlende Phasenkonzept von allen anderen analysierten Prozessen ab. Die einzelnen Prozessmuster werden u.a. durch die Attribute „Absicht" (Intent), „Kontext", „Erwägungen" (Considerations), „Strategien" und „Leistungen" (Benefits) näher beschrieben. Zur übersichtlicheren Darstellung der „Prozessarchitektur" lassen sich folgende vier Kategorien von spezifizierten Prozessmustern unterscheiden: Hauptprozessmuster (Main Process Patterns), Unternehmensmodellierungsmuster (Business Modeling Process Patterns), Komponentenspezifikationsmuster (Patterns for Specifying Components) und Komponentenentwurfsmuster (Component Design Patterns). Komponentenentwurfsmuster werden bei D'Souza und Wills (1999) weiter unterteilt in Muster zum „Designing to Meet a Specification" (Design-by-Contract, S. 639f.) und „Detailed Design Patterns" (S. 669f.), wobei diese Entwurfsmuster Ähnlichkeiten mit denen von Gamma u. a. (2002) aufweisen (siehe auch Abschnitt 4.1.3). Die Hauptprozessmuster beschreiben vier unterschiedliche Profile zur Projektgestaltung, nämlich „Objektentwicklung von Grund auf", „Reengineering", „Kurze Entwicklungszyklen" und „Paralleles Arbeiten". Die Aktivitäten zur

Unternehmensmodellierung und Komponentenspezifikation sind mit jeweils 13 Prozessmustern sehr ausführlich beschrieben.

Trotz dieses losen Aufbaus aus einzelnen Prozessmustern folgt der Ablauf aller Aktivitäten einem bestimmten gemeinsamen Grundgerüst, bestehend aus Modellierung, Entwurf, Implementierung und Test (Fettke und Loos (2002)), welches rekursiv über die drei spezifizierten Abstraktionsstufen Unternehmensmodell, Komponentenspezifikation und interner Komponentenentwurf (Realisierung, „Implementierung“ auf Modellebene) angewendet werden kann. Die unterschiedlichen Bezugsobjekte (Artefakte) innerhalb dieser Aktivitäten sind allerdings weder genau festgeschrieben noch eindeutig zugeordnet. So werden Artefakte wie beispielsweise Geschäfts- und Domänenmodelle, Systemarchitektur und Komponentenmodelle lediglich exemplarisch im Rahmen einer Fallstudie beschrieben (D’Souza und Wills (1999)). Aufgrund der oben dargestellten hohen Freiheitsgrade charakterisieren D’Souza und Wills (1999) den Catalysis-Prozess auch als „Nonlinear, Iterativ und Parallel“ (S. 512).

Wegen der prozessmusterorientierten Architektur gestaltet sich die Prozesssteuerung entscheidungsorientiert. Je nach Situation und Rahmenbedingungen muss von den Prozessbeteiligten entschieden werden, welche Arbeitsschritte durchzuführen sind bzw. wie die einzelnen Aktivitäten weiter verknüpft werden sollen. Es existieren sozusagen keine „Kanten“ im Prozessgraphen, an den Artefakte „entlang fließen“ oder Aktivitäten fest verbunden werden können.

Komponentenverständnis

Da Catalysis speziell und ausschließlich für die Entwicklung von und mit Komponenten entworfen wurde, stellen diese das zentrale Konzept des Ansatzes dar. Der verwendete Komponentenbegriff deckt sich weitgehend mit der in dieser Arbeit verwendeten Komponentendefinition (vgl. Abschnitte 4.2.1 und 4.2.3, sowie D’Souza und Wills (1999), S. 385ff.). Lediglich die Vermarktbarkeit wird bei D’Souza und Wills (1999) nicht explizit behandelt. Es werden sowohl Top-Down- als auch Bottom-Up-Aktivitäten zur Komponentenbildung beschrieben, im Gegensatz zu Perspective aber ausschließlich mit dem Ziel eine komponentenbasierte USW-Anwendung zu produzieren – die Komponentenerstellung wird also nicht losgelöst betrachtet. Das Vorgehensmodell bietet die breiteste Abdeckung des Geschäftskomponentenlebenszykluses von allen bisher analysierten Modellen. Die Standardisierung bzw. Spezifikation von Komponenten wird detailliert anhand von 13 unterschiedlichen Prozessmustern beschrieben („How to Specify a Component“, D’Souza und Wills (1999), S. 581ff.), ebenso wie die Entwicklung bzw. der Entwurf der Interna von Komponenten mit 18 unterschiedlichen Prozessmustern („How to Implement a Component“, D’Souza und Wills (1999), S. 639ff.). überdies wird die technische Anpassung von und die Komposition mit vorhandenen Komponenten mit jeweils zwei bzw. fünf Mustern spezifiziert. Generell operiert Catalysis auf drei unterschiedlichen Modellierungs- bzw. Betrachtungsebenen von Komponenten: der Geschäfts-/Domänenebene, die die äußere Problemsituation (Methodik- bzw. Komponentenkontext) modelliert, der Komponentenspezifikation, welche die Grenzen, Schnittstellen und Verantwortlichkeiten abbildet (Black Box) und der Komponentenentwurfsebene mit der internen Architektur (White Box, vgl. Abbildung 4.3).

Kollaborationspunkte

Da das Vorgehensmodell kein explizites Rollenmodell spezifiziert, müssen eigene Konzepte zur Aufteilung von Aufgaben und Verantwortlichkeiten eingebracht werden. Grundsätzlich zielen die Autoren von Catalysis auf kürzere Produkteinführungszeiten (Time-to-Market) ab und liefern mit den beiden Hauptprozessmustern Short-Cycle Development und Parallel Work Vorgaben dafür, wobei eine massive Parallelisierung von “largely independent [tasks]“ (D’Souza und Wills (1999), S. 541) befürwortet wird. Eine

Aufteilung und Integration von Aufgaben und Ergebnissen entlang der Komponentenarchitektur wäre denkbar, wird aber nicht expliziert.

Adaptionsmöglichkeiten

Catalysis bietet durch den flexiblen Einsatz und die Konfiguration von Prozessmustern die Möglichkeit zur Anpassung des „Vorgehensmodells“ an unterschiedliche USW-Typen, Softwareerstellungsorganisationen und Anwendungsdomänen. Für zwei typische Projektarten, zum einen „Entwicklung ohne Wiederverwendung“ (Build Route) und „Komposition von Komponenten“ (Assembly Route) werden grobe Prozessschablonen bereitgestellt (D’Souza und Wills (1999), S. 511f.). Die flexible Adaption durch Prozessmuster und -schablonen bringt allerdings auch den Nachteil mit sich, dass der Prozessingenieur, welcher das Tailoring (vgl. V-Modell) vornimmt, aus der Dokumentation sehr wenig systematische Unterstützung erhält (Fettke und Loos (2002)). Diese Problematik trat aber auch bereits bei OPEN und UP auf.

Werkzeugunterstützung

Durch die Einhaltung des UML-Standards bei der Modellierung von Komponenten und klar definierten Beziehungen zwischen Artefakten kann der Prozess durch „popular object-modeling UML tools“ (D’Souza und Wills (1999), S. 41) effektiv unterstützt werden und simple Verwendungsrichtlinien für diese Werkzeuge bereitstellen. Für Projekt-, Konfigurations- und Qualitätsmanagement können ebenfalls beliebige am Markt erhältliche Werkzeuge verwendet werden, da der Prozess für die Erfüllung der Querschnittsfunktionen keinerlei Vorgaben liefert. Denkbar wäre eine Ergänzung um die entsprechenden Workflows des RUP (vgl. Abschnitt 4.4.3).

3.2.6 UML Components

Bei dem komponentenbasierten Vorgehensmodell von Cheesman und Daniels (2001) handelt es sich, im Gegensatz zu den bisher analysierten Modellen, um ein sehr kompaktes und übersichtlich strukturiertes Prozessmodell, welches auf dem UML/OCL-Standard der Object Management Group (2003e) und dem RUP aufbaut.

Philosophie

Die Autoren unterscheiden bei der Softwareentwicklung Management sowie Entwicklungsaspekte, wobei sie sich von vornherein auf Letztere fokussieren, da UML Components (UMLC) letztendlich mit einer „variety of management processes“ (Cheesman und Daniels (2001), S. 26) einsetzbar sein soll. Innerhalb des Entwicklungsprozesses wird wiederum ein Schwerpunkt gesetzt, und zwar auf die Spezifikation der einzelnen Komponenten und der Komponentenarchitektur mit dem Ziel der Erstellung komponentenbasierter USW (vgl. Catalysis). Es handelt sich also um ein relativ eng gefasstes und spezielles Vorgehensmodell, das leicht handhabbar sein soll, klare methodische Vorgaben formuliert und sich an bestehenden, etablierten Standards orientiert.

Terminologie

Das Modell verwendet den Begriff „Phase“ nicht explizit, spricht aber von mehreren Iterationen, in denen die einzelnen Workflows (vgl. UP) durchlaufen und Artefakte inkrementell erstellt werden. Die Workflows wiederum sind in einzelne Tasks untergliedert, was den Aktivitäten in anderen Modellen, wie beispielsweise dem RUP, entspricht. Auch der restliche Sprachgebrauch orientiert sich im Wesentlichen an den Modellen von Jacobson u. a. (1999) und Kruchten (2000) (siehe auch Tabelle 4.1).

Modellumfang

Die Dokumentation des Vorgehensmodells wirkt mit weniger als 200 Seiten sehr kompakt und schlank, was nicht zuletzt durch die zahlreichen Querverweise auf den RUP

und die UML-Spezifikation ermöglicht wird (siehe Cheesman und Daniels (2001), sowie Apperly u. a. (2003)). Bis auf die Wartungsphase werden alle wesentlichen Abschnitte des allgemeinen Softwarelebenszykluses durch das Modell abgedeckt, wobei für „Anforderungsanalyse“ (Requirements), „Test“ sowie „Installation und Einsatz“ (Deployment) lediglich auf die entsprechenden Workflows beim RUP verwiesen wird. „Spezifikation“, „Beschaffung“ (Provisioning) und „Montage“ (Assembly) ersetzen andererseits die RUP-Prozessabläufe „Analyse und Design“ und „Implementierung“ (vgl. Cheesman und Daniels (2001) und Abschnitt 4.4.3). Wie bereits erwähnt, werden Management- bzw. Querschnittsfunktionen im Rahmen der Dokumentation nicht spezifiziert, genauso wenig wie Rollen bzw. ein eigenes Rollenmodell.

Prozessarchitektur

Die Architektur des Prozessmodells setzt sich aus den in Abbildung 4.17 dargestellten sechs Basisarbeitsabläufen (Workflows) zusammen. Der Prozess, der zu einem Modell der Unternehmensanforderungen führt, ist nicht Teil der Dokumentation von UMLC („Unternehmensmodellierung“, siehe Abschnitt 2.3). Besonderes Augenmerk liegt auf dem Spezifikations-Workflow, welcher sich weiter in die Unterarbeitsschritte Komponentenidentifikation, Komponenteninteraktion und Komponentenspezifikation gliedert. Es sollen dabei System- und Geschäftskomponenten getrennt identifiziert werden, und wieder verwendbare Komponenten sowie andere „Software Assets“ (Cheesman und Daniels (2001), S. 34) miteinbezogen werden (siehe Abbildung 4.17: „Wiederverwendungsarchiv“). Die daraus entstandene anfängliche Komponentenarchitektur wird im Schritt Komponenteninteraktion weiter um die Beziehungen zwischen den Komponenten verfeinert und die Details der USW-Systemstruktur festgelegt. Die eigentliche Komponentenspezifikation besteht dann aus der Definition des Interface Information Models (IIM), der Vor- und Nachbedingungen der Operationen und der Constraints der Schnittstelle. In den Workflows „Beschaffung“ und „Montage“ wird die Implementierung bzw. Bereitstellung („build or buy“, Cheesman und Daniels (2001), S. 28) und die Komposition von Komponenten knapp angerissen. Die Abfolge der Arbeitsabläufe wird durch den Evolutionsgrad der so genannten „Workflow Artifacts“ (Cheesman und Daniels (2001)) Konzeptmodell, Anwendungsfallmodell, Komponentenspezifikation und der eigentlichen Komponenten bestimmt. Das Prozessmodell ist daher in erster Linie ergebnisgesteuert.

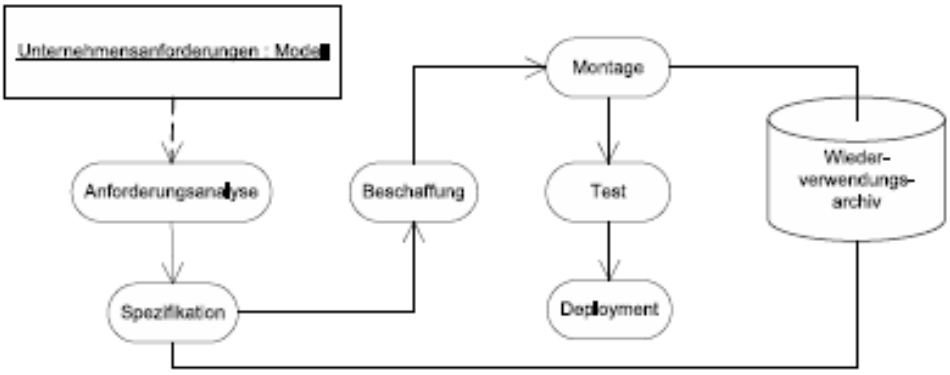

Abbildung 11: Die Prozessabläufe von UML Components

Komponentenverständnis

Das Komponentenmetamodell des UMLC-Ansatzes wurde bereits in Abschnitt 4.2.1 dieses Kapitel vorgestellt, da es sehr differenziert fünf unterschiedlich abstrakten Erscheinungsformen des Komponentenkonzepts im Entwicklungsprozess widerspiegelt: Komponentenschnittstelle, Komponentenspezifikation, Komponentenimplementierung, Komponenteninstallation und Komponentenobjekt (Instanz, vgl. Abbildung 4.3). Der Spezifikations-Workflow des UMLC-Prozessmodells beschreibt ein hierarchisches Top-

Down-Vorgehen bei der Identifikation der einzelnen Komponenten. Wie bereits angedeutet, deckt UMLC in erster Linie die Standardisierungsphase des Komponentenlebenszykluses ab, geht aber auch auf Aktivitäten der Entwicklung und Wiederauffindung („Beschaffung"), sowie Komposition („Montage") und Betrieb („Deployment") ein.

Kollaborationspunkte, Adaptionsmöglichkeiten und Werkzeugunterstützung

In puncto Kollaboration lässt sich auch über UMLC lediglich sagen, dass der komponentenbasierte Aufbau und die genaue Spezifikation von USW-Modellen eine Aufteilung der Arbeit entlang dieser Dekomposition ermöglichen könnte. Aspekte der organisationsübergreifenden Entwicklung wurden im Modell nicht expliziert. Des Weiteren muss bei den Adaptionsmöglichkeiten auf die aus dem RUP „importierten" Workflows und deren Anpassung verwiesen werden. UMLC konzentriert sich sehr stark auf die Komponentenspezifikation durch Design-by-Contract und bietet in diesem Bereich daher auch kaum Variationsmöglichkeiten an. Bei der Werkzeugunterstützung kommen UML-konforme Modellierungstools, wegen der engen Bindung an den RUP insbesondere diejenigen von IBM/Rational, in Betracht (siehe Abschnitt 4.4.3).

3.2.7 BOOSTER

Der Ansatz von Korthaus (2001), Business Object-Oriented Software Technology for Enterprise (Re-)Engineering (Booster), hat sich die „geschäftkomponentenbasierte Entwicklung betrieblicher Informationssysteme" (S. 140) zum Ziel gemacht und beschreibt dabei ein sehr komponentenzentriertes Vorgehensmodell, bei dem technische Aspekte wie Komponentenmodell und Verteilung ebenso wie fachliche Gesichtspunkte (Spezifikation von Geschäfts-/Fachkomponenten) beleuchtet werden. Das Vorgehensmodell subsumiert etablierte Verfahren und Techniken aus den Vorgängermodellen Baseballmodell (Coad und Nicola (1993), siehe Abschnitt 3.4.4), Unified Process, Perspective und Catalysis (siehe auch Korthaus (1998)). Die betriebswirtschaftlichen Überlegungen basieren nach eigenen Angaben des Autors in erster Linie auf dem Software Reuse-Ansatz von Jacobson u. a. (1997) und dem Business Component Approach von Herzum und Sims (1999).

Philosophie

Der Booster-Ansatz setzt ebenfalls auf der UML als Standardnotation auf und „erbt" damit die drei Hauptcharakteristika des UP, nämlich dass er anwendungsfallgetrieben, architekturzentriert und iterativ/inkrementell abläuft. Es handelt sich also um einen objektorientierten Ansatz, der versucht, eine „durchgängige Betrachtung des Geschäftskomponentenkonzepts" (Korthaus (2001), S. 140) zu ermöglichen (Traceability, vgl. Abschnitt 2.3 sowie Catalysis). Der Schwerpunkt liegt bei diesem Vorgehensmodell auf der Kombination von Business Engineering und Software Engineering. Zusätzlich zu rein softwaretechnischen Methoden legt Booster viel Wert auf Unternehmensmodellierung, insbesondere auf die Geschäftsprozessanalyse und -verbesserung (BPR, siehe Abschnitt 2.3.2 bzw. Jacobson u. a. (1997)). Die ebenfalls „durch den Ansatz propagierte modellbasierte Vorgehensweise" (Korthaus (2001), S. 139) ist ideologisch bereits ein Schritt in Richtung modellgetriebene Softwareentwicklung (MDD).

Terminologie

Das Prozessmodell unterscheidet eine Makroprozessebene und eine Mikroprozessebene. Aktivitäten werden daher hierarchisch in Makroprozess-Aktivitäten und Mikroprozess-Aktivitäten unterteilt (siehe Prozessarchitektur), Phasen existieren wie bei UMLC lediglich implizit durch den iterativen Charakter des Prozesses. Anstatt von Ergebnissen wird teilweise auch von „Produkten" gesprochen und Richtlinien werden vom Autor als „Handlungsanweisungen" bezeichnet (siehe auch Tabelle 4.1).

Modellumfang

Die ausführliche Beschreibung des Booster-Ansatzes findet in der Dissertationsschrift von Korthaus (2001) statt und gliedert sich in die beiden Hauptteile Booster*Core (Konzeptionsrahmen) und Booster*Process (Vorgehensmodell). Das eigentliche Vorgehensmodell baut auf dem durch den Konzeptionsrahmen bereitgestellten „Modellierungsinstrumentarium“ (Korthaus (2001), S. 140) auf. Zusammen mit einem exemplarischen Projekt (Prozessmodellinstanz) umfasst die Dokumentation über 200 Seiten und bietet zahlreiche Verweise auf die Primärquellen bestimmter beschriebener Techniken. Booster deckt die aus dem allgemeinen Softwarelebenszyklusmodell bekannten Phasen „Unternehmensmodellierung“, „Analyse“, „Entwurf“ und „Implementierung“ ab, wobei Korthaus (2001) ähnlich wie Allen und Frost (1998) einen Anwendungs- und einen Komponentenentwicklungsprozess (Makroaktivitäten) unterscheidet (vgl. Perspective, Abschnitt 4.4.4). „Tests“ sowie die beiden letzten Phasen des Zykluses „Abnahme“ und „Wartung“ werden nicht näher erläutert. Zudem verzichtet der Autor bewusst auf die Beschreibung von Querschnittsfunktionen wie Projekt- und Konfigurationsmanagement, sowie auf soziotechnische Aspekte der notwendigen „Schaffung einer Wiederverwendungskultur“ (Korthaus (2001), S. 139) im Unternehmen, um sich in seiner Arbeit auf das eigentliche Ziel, die geschäftskomponentenbasierte Modellierung von betrieblichen Anwendungssystemen (USW) zu fokussieren. Der Booster-Ansatz definiert kein eigenes Rollenmodell, orientiert sich aber an dem in Abschnitt 4.2.2 im Rahmen der EJB-Entwicklung vorgestellten, eher technischen Modell von Sun (siehe Abbildung 4.6), da das Java-Komponentenmodell die intendierte Zielplattform des Ansatzes darstellt.

Prozessarchitektur

Abbildung 4.18 zeigt die Makroprozessarchitektur von Booster*Process mit den entsprechenden Informationsflüssen zwischen den einzelnen Bereichen (Makroaktivitäten). Die ausgefüllten Pfeile deuten die Hauptflussrichtungen an, und in Verbindung mit den gestrichelten Pfeilen in der Gegenrichtung wird der iterative sowie inkrementelle Charakter auch auf der Makroebene deutlich. In Anlehnung an Jacobson u. a. (1997) formuliert Korthaus (2001) einen Mikroprozess zur Unternehmensmodellierung (Business Engineering) bestehend aus den Aktivitäten „Formulierung der Unternehmensvision“, „Reverse-Engineering“ und „Forward-Engineering“ (oder auch BPR), welche in der Prozessdokumentation sehr detailliert beschrieben und anhand von Beispielen aus unterschiedlichen Domänen verdeutlicht werden. Auch die Entwicklung der Anwendungsarchitektur lässt sich in drei Unteraktivitäten auflösen: „Anforderungsermittlung“ auf Basis des Unternehmensmodells, „Globale Analyse“ zur Bestimmung von Komponentenkandidaten, der eigentliche „Architekturentwurf“ und auch „Implementation und Test der entworfenen Schichtenarchitektur“ (Korthaus (2001), S. 130).

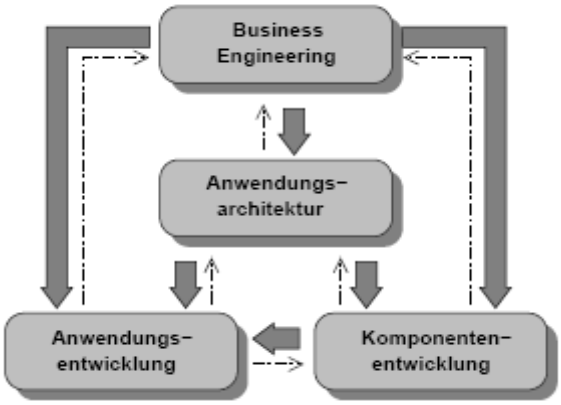

Abbildung 12: Die Booster-Prozessarchitektur (Korthaus, 2001)

Den Mikroprozess zur Anwendungsentwicklung zeigt Abbildung 4.19. Er basiert auf dem Baseballmodell von Coad und Nicola (1993) und ist auf die Wiederverwendung von Geschäftkomponenten hin erweitert worden – sowohl auf Spezifikations- als auch auf Implementierungsebene. Bei der Erläuterung der Aktivitäten zur Komponentenentwicklung stützt sich der Autor auf das Komponentenlebenszyklusmodell von Turowski (1999), wobei er hauptsächlich zwischen externem und internem Design einer Komponente unterscheidet (vgl. Catalysis). Es werden überdies ausgehend von Herzum und Sims (1999) erweiterte Heuristiken zur Identifikation und Abgrenzung von Geschäftskomponenten aufgestellt (Korthaus (2001), S. 137ff.). Der Booster-Ansatz beinhaltet eine „Vorgabe der in den jeweiligen Aktivitäten zu entwickelnden UML-Diagramme und Produkte“ (Korthaus (2001), S. 138) und ist daher hauptsächlich ergebnisgesteuert. Die Aktivitäten auf der Makro- und Mikroprozessebene bzw. deren Verknüpfungen sind nicht so genau spezifiziert, als dass man von einer aktivitätsorientierten Steuerung reden könnte.

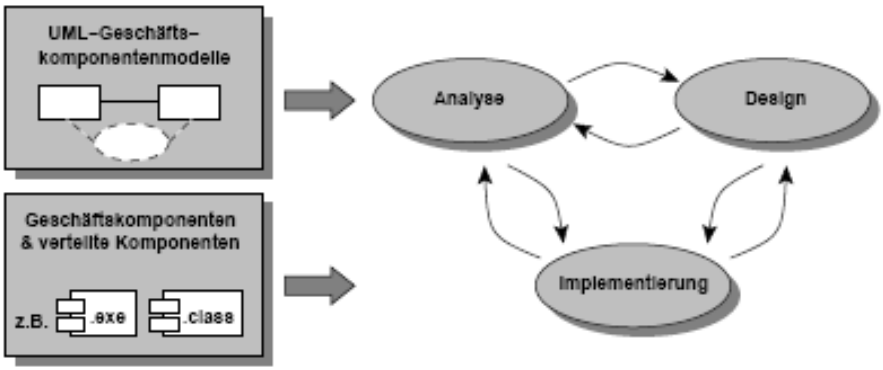

Abbildung 13: Wiederverwendungsorientierte Anwendungsentwicklung bei Booster

Komponentenverständnis

Aufbauend auf dem Konzept der Business Components von Herzum und Sims (1999) wird durch den technischen Konzeptionsrahmen Booster*Core ein mehrschichtiger und mehrdimensionaler Komponentenbegriff in Form eines UML-basierten Metamodells konstruiert (Korthaus (2001), S. 94). Hierbei werden u.a. Komponentengranularität und unterschiedliche Typen von Geschäftskomponenten (Prozess-, Entitäts- und Hilfs-Geschäftskomponenten) spezifiziert und daran anknüpfend ein UML-Profil für die Modellierung von Geschäftskomponenten innerhalb des Booster-Prozesses vorgestellt. Die Komponentenbildung kann bei Booster entweder top-down oder in Form einer Art Clusteranalyse bottom-up erfolgen, wobei hier nur auf die entsprechende Literatur verwiesen wird. Es wird zwar der gesamte Geschäftskomponentenlebenszyklus betrach-

tet, systematische Unterstützung wird aber lediglich für die Komponentenstandardisierung bzw. -spezifikation bereitgestellt.

Kollaborationspunkte

Bei der Besprechung der Limitationen des Modells wird bereits eine „Repository-basierte Dokumentation der Konzepte des Booster-Ansatzes“ (Korthaus (2001), S. 139) vorgeschlagen, welche eine teilweise modellbasierte, projekt- und gegebenenfalls organisationsübergreifende Kollaboration von Entwicklern durch ein Wiederverwendungsarchiv ermöglichen würde (vgl. dazu Abschnitt 4.1.5).

Adaptionsmöglichkeiten

Das Vorgehensmodell kann durch die Veränderung bzw. Erweiterung des vorgeschalteten Konzeptionsrahmens, d.h. durch Adaption des Metamodells und des UML-Profils (Booster*Core), an unterschiedliche Arten von USW bzw. Softwareunternehmen angepasst werden (vgl. Abschnitt 4.3.2). An dieser Stelle sei noch erwähnt, dass die OMG mit dem Standard „Enterprise Distributed Object Computing“ (EDOC) seit Anfang 2002 ein UML-Profil zur Erstellung von USW mittels verteilter Komponentenarchitekturen zur Verfügung stellt, welches einige der Konzepte von Booster*Core substituieren bzw. den Konzeptionsrahmen erweitern könnte (vgl. Object Management Group (2002c)). Hildenbrand und Korthaus (2004) argumentieren, dass somit nun die Möglichkeit bestünde, den Booster-Ansatz mit einem international anerkannten Industriestandard zu integrieren und auszubauen. Zu den Anpassungsmöglichkeiten bezüglich Anwendungsdomäne und Projektstruktur lassen sich anhand der Prozessbeschreibung keine direkten Aussagen treffen.

Werkzeugunterstützung

Konkrete Hinweise auf bestimmte Softwareentwicklungs- bzw. Softwaremanagementwerkzeuge werden bei Korthaus (2001) nicht gegeben, daher können zur Modellierung und Spezifikation von Geschäftkomponenten herkömmliche, UML-konforme Werkzeuge zu Einsatz kommen. Gleiches gilt für die Unterstützung der Prozessquerschnittsfunktionen (Projektmanagement, Konfiguration etc.), da diese überhaupt nicht Gegenstand der Dokumentation sind.

3.2.8 KobrA

Das letzte der acht hier analysierten Prozessmodelle stammt, wie auch der Booster-Ansatz, ebenfalls aus dem akademischen Umfeld. Der Ansatz zur „Komponentenbasierten Anwendungsentwicklung“ (KobrA) von Atkinson u. a. (2002) stellt überdies derzeit den State-of-the-Art der wiederverwendungsorientierten Vorgehensmodelle dar und integriert alle wesentlichen, bisher vorgestellten Wiederverwendungstechniken, wie Komponententechnologie, modellgetriebene Entwicklung mit der MDA, Frameworks und Produktfamilienkonzepte (vgl. Abschnitt 4.1).

Philosophie

Die grundlegende Zielsetzung bei der Entwicklung der KobrA-Methodik bestand in der Einfachheit, Systematik, Skalierbarkeit und Praktikabilität der Vorgehensweise (Atkinson u. a. (2002), S. 25ff.). Die Kernidee besteht in der Trennung von unterschiedlichen Belangen (Separation of Concerns, Atkinson u. a. (2002), S. 28), besonders der Produkten (Artefakten) und der Prozessen. Zu diesem Zweck werden drei orthogonale Dimensionen der Softwareentwicklung formuliert, welche mit jeweils unterschiedlichen Techniken bearbeitet werden können: Komposition (hierarchische Dekomposition), Abstraktionsebene (Embodiment, Implementierung) und generischer Charakter der Artefakte (Product Line Engineering). Jede der drei Dimensionen lässt sich auf eine der bereits vorgestellten Wiederverwendungstechniken zurückführen. Kompositionstechni-

ken bzw. Dekomposition sorgen für Komplexitätsbeherrschung durch eine rekursive Divide&Conquer-Strategie bei der Komponentenmodellierung, die Abstraktionsebenen entsprechen im Wesentlichen denen der MDA (CIM, PIM und PSM, siehe Abbildung 4.20) und Produktlinientechnik (Product Line Engineering) beschreibt die Variantenbildung mittels generisch erstellter Rahmenwerke und Architekturen (vgl. Abschnitt 4.1.3).

Terminologie

Da KobrA zahlreiche Techniken und Konzepte aus methodischen Vorgängermodellen subsumiert (OPEN, UP, Perspective, Catalysis und UMLC), lassen sich bei der verwendeten Terminologie keine Besonderheiten festmachen. Obwohl der iterative Charakter des Komponentenmodellierungsprozesses immer wieder betont wird, wird das Konzept der „Phase“ nicht expliziert, wobei man die rekursiv verschachtelten Komponentenspezifikations- und Realisierungstätigkeiten als Phasenkonzept interpretieren kann (vgl. Tabelle 4.1 sowie Atkinson u. a. (2002), S. 458ff.).

Modellumfang

Bei Atkinson u. a. (2002) wird die KobrA-Methodik sehr ausführlich und strukturiert beschrieben, und zahlreiche Anwendungsbeispiele tragen zum besseren Verständnis bei. Verglichen mit dem allgemeinen Softwarelebenszyklus liegt der Schwerpunkt der Dokumentation klar auf der Entwurfsphase, welche sich in der Komponentenmodellierung und optional beim Einsatz von Produktlinien in den Framework und Application Engineering-Aktivitäten abzeichnet. Aspekte der Unternehmensmodellierung (Enterprise Modeling) und der Analyse (Context Realization) werden von dem Modell nur rudimentär abgedeckt. Die Implementierung bzw. die Transformation der Modelle in Quellcode wird im Rahmen des so genannten Embodiment behandelt und kontinuierliche Tests sind Teil der Querschnittsfunktion „Qualitätssicherung“ (siehe Atkinson u. a. (2002), S. 379ff.). Wartung, änderungs- und Konfigurationsmanagement werden bei KobrA zu einem Maintenance-Prozess zusammengefasst. Ein großer Teil der Dokumentation, Project Monitoring and Control, erläutert überdies die Querschnittsfunktion „Projektmanagement“ prozessübergreifend. Abbildung 4.20 fasst noch einmal den Umfang der im Prozessmodell gegebenen Vorschriften, bezogen auf die unterschiedlich abstrakten Artefakte, zusammen. Atkinson u. a. (2002) definieren für die beschriebenen Aktivitäten kein eigenes Rollenmodell, sondern sorgen durch die hierarchische Struktur der Artefakte und der Aufgaben für eine mögliche Trennung von Verantwortlichkeiten.

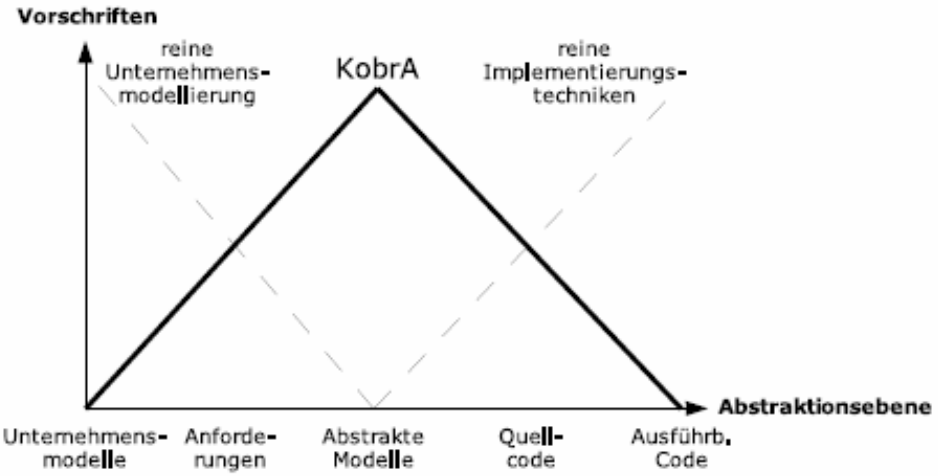

Abbildung 14: Umfang der Vorschriften bei KobrA (vgl. Atkinson u. a., 2002)

Prozessarchitektur

Charakteristisch für die Prozessarchitektur von KobrA ist die rekursive, komponentenbasierte und modellzentrierte Verfeinerung der Artefakte, wobei das zu erstellende Gesamtsystem als „Wurzelkomponente“ den Ausgangspunkt des Prozesses darstellt. Durch die rekursive Vorgehensweise entsteht am Ende eine stücklistenartige, baumförmige Komponentenhierarchie (Containment Tree) mit atomaren Komponenten an den „Blättern“. Nachdem zunächst bei der „Kontextrealisierung“ durch Unternehmensmodellierung und Anforderungsanalyse der betriebliche Kontext und die Systemanforderungen festgelegt wurden, können die Komponenten in rekursiven Modellierungsphasen kaskadenartig spezifiziert (äußere Ansicht, Specification) und ihr innerer Aufbau beschrieben werden (Realization) – und das auf mehreren verschachtelten Hierarchieebenen bis hin zu primitiven Komponenten. Dabei muss nicht jeder „Ast“ dieselbe Pfadtiefe haben (siehe Abbildung 4.22). Der Aufbau der Komponentenhierarchie und damit der Modellierungsprozesse folgt dem Composite-Muster von Gamma u. a. (2002). Charakteristisch für die Architektur des KobrA-Modells ist die strikte Separation von Artefakten und Prozessaspekten bei der Beschreibung der einzelnen Aktivitäten (vgl. Abbildung 4.21). Dem oben skizzierten Top-Down-Verfahren bei der Neuentwicklung von komponentenbasierten Systemen steht die Einbindung von wieder verwendbaren Komponenten (Component Reuse) gegenüber. Es entfällt dann jeweils die Realisierungsphase und es müssen Conformance Maps bzw. Semantic Maps zum Angleichen der Komponentenspezifikationen definiert werden (siehe Atkinson u. a. (2002), S. 256ff.). Je mehr Komponenten in den Embodiment-Aktivitäten (Implementierung) wieder verwendet werden, desto mehr bekommt KobrA einen zusätzlichen Bottom-Up-Charakter der Anwendungsentwicklung.

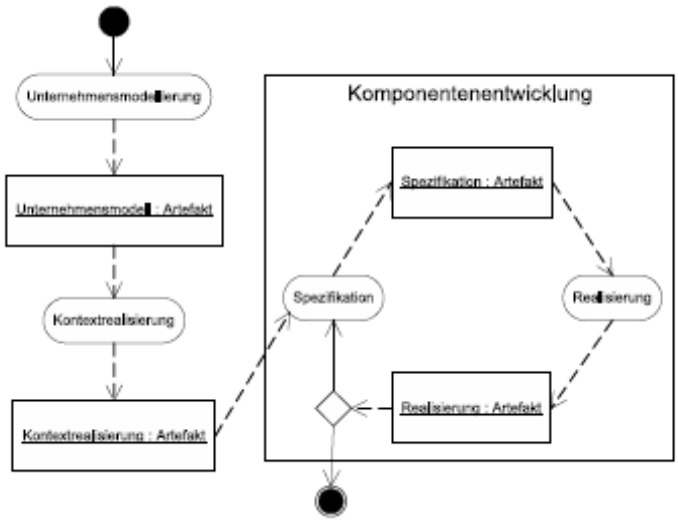

Abbildung 15: KobrA-Artefakte und -Prozesse

Die bisher beschriebene Prozessarchitektur ist ein Spezialfall, welcher Produktlinienaspekte außer Acht lässt und keine Variabilitäten definiert. Das KobrA-Prozessmodell beschreibt aber auch Aktivitäten zur Entwicklung von generischen Komponentenhierarchien bzw. Softwaresystemen, aus denen dann ähnlich wie bei C++ mit Templates (generische Programmierung) konkrete Anwendungen instantiiert werden können (Framework Engineering, vgl. Atkinson u. a. (2002)). Die generischen Elemente sind in den Framework-Modellen mit dem UML-Stereotyp <<variant>> gekennzeichnet und können mit Hilfe von Entscheidungstabellen (Decision Models) im Hinblick auf die speziellen Projekt- bzw. Unternehmensanforderungen konkretisiert werden. Auch die Prozesssteuerung wird von der rekursiven Vorgehensweise bei der Zerlegung des Systems in Komponenten geprägt. Bezogen auf die einzelne Komponente gestaltet sich

das Vorgehensmodell bei der Spezifikation eher aktivitätsorientiert, die Realisierung der USW jedoch hat dann mehr einen vertragsgetriebenen Charakter, da sie an die Spezifikation gebunden ist (Design-by-Contract).

Komponentenverständnis

Atkinson u. a. (2002) formulieren für ihr Prozessmodell ein spezielles „KobrA-Komponentenmodell“ (S. 67ff.). Im Gegensatz zu Autoren wie Szyperski u. a. (2002) werden Komponenten nicht nur als ausführbare Codeeinheiten angesehen, sondern ihre Eigenschaften als logisches Konzept hervorgehoben, welches Komponenteninstanzen sowohl während der Entwicklungszeit, als auch während der Laufzeit beschreibt (vgl. UP, UMLC und Booster). Um diesen erweiterten Komponentenbegriff zu verdeutlichen wird in der englischsprachigen Dokumentation der Begriff „Komponent“ statt „Component“ verwendet. Wie schon bei der Prozessarchitektur erwähnt, setzen sich KobrA-Komponenten hauptsächlich aus modellbasierten Spezifikations- und Realisierungsartefakten zusammen. Atkinson u. a. (2002) liefern für die Komponentenmodellierung einige Grundprinzipien, die die Handhabung dieser Aufgabe im Rahmen des Prozessmodells erleichtern sollen (S. 80ff.).

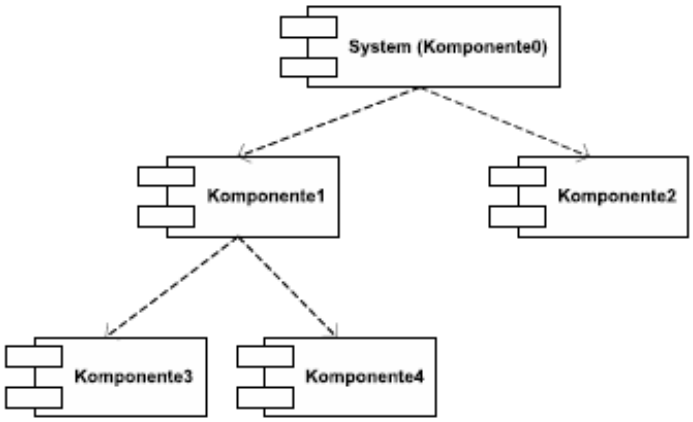

Abbildung 16: Beispielhafte KobrA-Komponentenhierarchie

Komponenten können wiederum Unterkomponenten haben und somit eine Baumstruktur bilden. Innerhalb dieses Baums können die Komponentenknoten unterschiedliche Beziehungen zueinander aufweisen, nämlich Client/Server- (Clientship), Kompositions- und Besitzverhältnisse (Ownership). Die Komponentenbildung erfolgt streng hierarchisch (top-down), außer wenn Commercial-Off-The-Shelf -Komponenten (COTS-Komponenten) in die Entwicklung und somit von „unten“ in den Komponentenbaum einfließen. Bezogen auf das Komponentenlebenszyklusmodell von Turowski (1999) (vgl. Abbildung 4.5) fokussieren Atkinson u. a. (2002) insbesondere die modellbasierte Standardisierung und Entwicklung von Komponenten, Implementierung im eigentlichen Sinn und Codegenerierung nach dem MDA-Ansatz werden ebenfalls beschrieben, wenn auch nicht sehr ausführlich. Die Wiederauffindung von Komponenten wird nur beiläufig erwähnt, wohingegen mit den Decision Models und Conformance bzw. Semantic Maps detaillierte Techniken zur Anpassung von Komponenten geliefert werden. Auch die Komposition mit den oben gezeigten, unterschiedlichen Beziehungen der Komponenten zueinander wird von dem Modell abgedeckt. Die Phase „Betrieb“ (Deployment) und damit auch die Verteilung der Komponenten auf die Infrastruktur des Anwenderunternehmens wird ebenfalls ausführlich behandelt.

Kollaborationspunkte

Der starke Einfluss der architektonischen Zerlegung des Softwaresystems und die genaue Spezifikation der einzelnen Komponenten sollte eine beliebig granulare, arbeits-

teilige Erstellung von USW erlauben. Zu diesem Zweck wäre ein vordefiniertes Rollenmodell mit einer Zuordnung der Verantwortlichkeiten und Anforderungen an die Mitarbeiter für einzelne Aktivitäten, sowie eine umfassende Werkzeugunterstützung dieses sehr komplexen Prozesses zur besseren Koordination und Synchronisation der Aktivitäten wünschenswert.

Adaptionsmöglichkeiten

Durch die optionale Einbindung von Produktlinienkonzepten bietet KobrA sehr gute Möglichkeiten zur Anpassung des Prozessmodells an bestimmte USW-Typen und auch an USW-Familien innerhalb einer Anwendungsdomäne. KobrA skaliert durch seinen rekursiven Prozessaufbau sehr gut für kleinere und größere Projekte, wobei jedoch die Einführung des Prozesses in eine bestehende Organisationsstruktur wegen der zahlreichen strikten Modellierungsvorschriften (siehe Abbildung 4.20) auf soziotechnischer Ebene problematisch sein könnte (siehe Abschnitt 4.4.10 bzw. 4.5).

Werkzeugunterstützung

Zur durchgängigen Unterstützung des Prozessmodells ist derzeit noch kein dediziertes Werkzeug verfügbar. Allerdings kann die Verwendung von UML-2.0-konformen Modellierungstools die Abbildung von hierarchisch gegliederten Artefakten und die Modellierung von komplexen Komponentensystemen mit den unterschiedlichen Beziehungen der Komponenten untereinander wesentlich vereinfachen (siehe auch Abschnitt 4.2.4). KobrA stellt in Sachen Wiederverwendung momentan den State-of-the-Art der analysierten Prozessmodelle dar, da das Modell sowohl Kompositions- als auch Generierungstechniken bei der Erstellung von USW integriert. Es existieren bereits weiterführende Arbeiten von Atkinson, welche den bisher nur angerissenen MDD-Charakter weiter ausbauen (siehe Atkinson und Muthig (2002) bzw. Atkinson und Groß (2003)).

3.2.9 Bewertung der fortgeschrittenen Vorgehensmodelle

Die Reihenfolge, in der die Vorgehensmodelle analysiert wurden, wurde bewusst chronologisch jeweils nach dem Jahr der ersten Veröffentlichung der analysierten Version gewählt. Es lässt sich innerhalb dieser Folge eine klare Tendenz hin zu mehr Wiederverwendung im Softwareentwicklungsprozess festmachen. Angefangen mit ersten Komponenten- und Modulkonzepten beim V-Modell und dem UP, über dediziert komponentenorientierte Prozessmodelle (Perspective und Catalysis), weisen Booster und KobrA bereits den Weg hin zur Entwicklung der (Geschäfts-)Komponenten auf einer höheren, primär modellbasierten Abstraktionsebene. Dies soll nach Aussagen von Atkinson u. a. (2002) letztendlich die eigentliche Implementierung in einer rein textbasierten Programmiersprache substituieren.

Abschließend sollen in diesem Abschnitt noch einmal die wichtigsten Ergebnisse in Form von Tabellen und vergleichenden Erläuterungen dargestellt werden, um einen besseren überblick über die existierenden Methodiken zu erhalten und weitere wertvolle Erkenntnisse für das konstruktive Ziel dieser Arbeit zu identifizieren. In der verwendeten Terminologie der Vorgehensmodelle spiegeln sich häufig auch die jeweiligen Prozessarchitekturen wider – beispielsweise fehlt bei Catalysis das Phasenkonzept, da das Modell lose aus Prozessmustern „aufgebaut“ ist. Bei den Modellelementen „Aktivität“, „Ergebnis“ und „Rolle“ (siehe Prozess-Metamodell, Abbildung 3.1) gibt es keine nennenswerten Unterschiede bei den Modellen. Die Elemente „Technik“ und „Richtlinie“ wurden innerhalb der Modelldokumentationen oft unterschiedlich und inkonsistent bezeichnet, so dass in Tabelle 4.1 lediglich der häufigste Bezeichner eingetragen ist.

Modellelement	V-Modell	OPEN	UP/RUP	Perspective	Catalysis	UMLC	BOOSTER	KobrA
Phase	Phase	*Activity/Stage*	Phase	*Stage*	–	(Phase)	Phase	*Recursion*
Aktivität	Aktivität	*Task*	*Activity*	*Task*	*Activity*	*Task*	Aktivität	*Activity*
Ergebnis	Produkt	*Work Product*	*Artifact*	*Deliverable*	*Deliverable*	*Artifact*	Produkt	*Artifact*
Rolle	Rolle	*Producer*	*Worker*	*Team Role*	–	–	Rolle	–
Technik	Elementarmeth.	*Tools/Techn.*	*Step*	*Technique*	*Technique*	*Technique*	Technik	*Technique*
Richtlinie	Handbuchsamml.	*Tools/Techn.*	*Guideline*	*Guideline*	–	*Guideline*	Handl.-anw.	*Rules*
Notation	–	*Language*	*Language*	Notation	Notation	Notation	Notation	Notation

Tabelle 2: Übersicht über die Terminologie fortgeschrittener Modelle

Da zu jedem Vorgehensmodell mindestens eine Monografie und zahlreiche Arbeitspapiere existieren, war die Dokumentation im Allgemeinen sehr umfangreich. Bei der Abdeckung der Softwarelebenszyklusphasen gab es bezüglich systematischer Unterstützung einzelner Phasen doch größere Unterschiede (siehe Tabelle 4.2). UML Components beispielsweise bietet keinerlei Unterstützung bei der Unternehmensmodellierung, was die Methodik in dieser Form für die Erstellung von USW zunächst ungeeignet macht. Booster andererseits hat seine Schwerpunkte u.a. in der betrieblichen Analyse und der Geschäftskomponentenmodellierung, vernachlässigt aber Aspekte des Testens und der Übergabe an den Kunden („Transition Phase“, vgl. UP).

Phase	V-Modell	OPEN	UP/RUP	Perspect.	Catal.	UMLC	BOOSTER	KobrA
Untern.-mod.	X	(X)	X	X	X	–	X	X
Analyse	X	X	X	X	X	X	X	X
Entwurf	X	X	X	X	X	X	X	X
Implement.	X	X	X	X	X	X	X	X
Test	X	(X)	X	X	X	X	–	(X)
Abn./Einf.	X	X	X	X	(X)	X	–	(X)
Wart./Pflege	X	X	(X)	(X)	–	–	–	X

Tabelle 3: Übersicht über die Phasenabdeckung fortgeschrittener Modelle

Bei der Abdeckung des gesamten Softwareentwicklungsprozesses, einschließlich der Querschnittfunktionen, bilden die analysierten Modelle klar zwei Cluster: zum einen diejenigen, die die eigentlichen Entwicklungstätigkeiten und zusätzlich Projekt-, Konfigurations-, änderungs- und Qualitätsmanagement als integriertes Vorgehensmodell unterstützen, und andererseits Catalysis, UMLC und Booster, welche sich lediglich auf den Kernprozess der komponentenbasierten Softwareentwicklung spezialisieren (vgl. Tabelle 4.3). Bei den drei zuletzt genannten Modellen wird dann an entsprechender Stelle auf unabhängige Softwaremanagementverfahren und -werkzeuge verwiesen.

Prozessteil	V-Modell	OPEN	UP/RUP	Persp.	Catal.	UMLC	BOOSTER	KobrA
Softwareentwicklung	X	X	X	X	X	X	X	X
Projektmanagement	X	X	X	X	–	–	–	X
Konfig.-/Änderungsmgt	X	X	X	(X)/–	–	–	–	X
Qualitätsmanagement	X	(X)	X	X	–	–	–	X

Tabelle 4: Die Unterstützung von Querschnittfunktionen bei fortgeschrittenen Modellen

Die beiden Tabellen 4.4 und 4.5 sollen noch einmal die wichtigsten Punkte aus den Analysekategorien zusammenfassen und vergleichend darstellen. Es ist auffällig, dass keine zwei Prozessmodelle exakt die gleiche Philosophie vertreten, was sich schon allein dadurch ergibt, dass die Formulierung eines neuen Vorgehensmodells auch durch neuartige Ideen gerechtfertigt sein sollte und sich gegenüber den anderen Modellen differenzieren muss. Trotzdem lassen sich die drei Kategorien

1. (objektorientierte) Prozessrahmenwerke (V-Modell, OPEN und UP),
2. rein komponentenorientierte Prozessmodelle (Perspective und Catalysis) und
3. Vorgehensmodelle zur Komponentenmodellierung und -spezifikation (UMLC, Booster und KobrA)

erkennen, wobei Catalysis auch schon sehr ausführlich die Komponentenspezifikation mit der UML beschreibt und daher auch der dritten Kategorie zugeordnet werden könnte. Erhebliche Unterschiede ließen sich bei der Beschreibung der Prozessarchitektur und den Möglichkeiten zur Umsetzung bzw. Anpassung in konkreten Projekten feststellen. Während die Prozessrahmenwerke Metaprozessmodelle zur individuellen Instantiierung (vgl. Abbildung 1.1) zur Verfügung stellen, bietet Catalysis einzelne, lose Prozessbausteine an. Booster und KobrA wiederum beschreiben zu diesem Zweck einen veränderlichen Konzeptrahmen (Komponentenmetamodell) bzw. Produktlinienansätze.

Modellaspekt	V-Modell	OPEN	UP/RUP	Perspective
Philosophie	Qualität	Vertragsprinzip	Anwendungsfälle	Auslieferung
Modellumfang	sehr umfangreich	umfangreich	sehr umfangreich	umfangreich
Prozesssteuerung	aktiv.-/ergebnisor.	vertragsor.	aktivitätsor.	aktiv.-/ergebnisor.
Komponentenbildung	*top-down*	–	*top-down*	*bottom-up*
Kollaborationspkt.	wenige	mehrere	viele	mehrere
Adaptionsmögl.	*Tailoring*	Instantiierung	Instantiierung	Profile
Werkzeugunterst.	funkt. Anf.	–	kommerziell	kommerziell

Tabelle 5: Übersicht über die analysierten Kategorien (Teil 1)

Modellaspekt	Catalysis	UMLC	BOOSTER	KobrA
Philosophie	Prozessmuster	Komponentenspez.	Geschäftskompon.	Rekursion
Modellumfang	sehr umfangreich	gering	umfangreich	sehr umfangreich
Prozesssteuerung	entscheidungsor.	ergebnisor.	ergebnisor.	aktivi.-/vertragsor.
Komponentenbildung	*top-down/bottom-up*	*top-down*	*top-down/bottom-up*	*top-down*
Kollaborationspkt.	mehrere	wenige	mehrere	mehrere
Adaptionsmögl.	Muster/*Routes*	s. UP/RUP	Metamodell	Produktlinien
Werkzeugunterst.	UML-Tools	UML-Tools	UML-Tools	UML-Tools

Tabelle 6: Übersicht über die analysierten Kategorien (Teil 2)

3.3 Aktuelle Vorgehensmodelle

Der RUP in seiner aktuellsten Fassung von 2004 sowie das neue V-Modell XT, welches erst Anfang 2005 endgültig veröffentlicht wurde, bieten gegenüber den bereits analysierten Vorgängerversionen zahlreiche Neuerungen. Da diese sich hauptsächlich auf die komponentenbasierte Softwareerstellung und die Durchführung großer, verteilter Projekte beziehen, soll im Folgenden auf den momentanen State-of-the-Art in Sachen Softwareprozessmodelle eingegangen werden.

3.3.1 Rational Unified Process 2004

Im Wesentlichen entspricht der *Rational Unified Process 2004* (RUP'04) dem in Abschnitt 3.2.3 bereits analysierten *Unified Software Development Process* (Jacobson u. a. (1999), Kruchten (2004)).

Philosophie und Terminologie

Die Philosophie und Terminologie hat sich in Bezug auf den UP bzw. die der letzten RUP-Version (Kruchten (2000)) nicht verändert (siehe hierzu Abschnitt 3.2.3). Die Entwicklungsmethodik des RUP charakterisiert sich im Wesentlichen durch

(a) iterative Entwicklungszyklen,

(b) einen architekturzentrierten Prozess und

(c) eine anwendungsfallgetriebene Vorgehensweise.

Modellumfang

Die Prozessdokumentation umfasst neben der sehr umfangreichen HTML-Beschreibung zahlreiche Monografien wie beispielsweise das Standardwerk von Kruchten (2004). Der RUP deckt alle wesentlichen Phasen des Softwarelebenszyklus sowie der entsprechenden Querschnittsfunktionen ab. Zusätzlich werden zahlreiche „Best Practices“ und so genannte „Road Maps“ für unterschiedliche Entwicklungsprojekte mitgeliefert. Das Rollenmodell umfasst 30 definierte Rollen in sechs unterschiedlichen Kategorien, z.B. Analystenrollen, Entwicklerrollen und Managementrollen (Kruchten (2004), S. 273ff.).

Prozessarchitektur und –steuerung

Wie auch der UP besteht der RUP’04 aus unterschiedlichen inhaltlichen Disziplinen, die iterativ durchlaufen werden. Die Artefakte werden dabei inkrementell bearbeitet und erweitert. Abbildung 9 zeigte bereits bei der Analyse des UP die Disziplinen und Phasen des RUP. Der RUP gibt die **Kern-Workflows**

1. Unternehmensmodellierung (Business Modeling),
2. Anforderungen (Requirements Engineering),
3. Analyse und Design (Analysis and Design),
4. Implementierung (Implementation),
5. Test sowie
6. Installation und Einsatz vor.

Des Weiteren werden die **unterstützenden Disziplinen** („Querschnittsfunktionen“)

1. Konfigurationsmanagement,
2. Projektmanagement und
3. Umgebung beschrieben (vgl. Kruchten, 2004).

Abbildung 17 stellt alle genannten Kern- und unterstützenden Disziplinen im Überblick dar. Ordnet man diese in der typischen RUP-Darstellungsweise mit einer zeitlichen Achse für die einzelnen Phasen (Definition, Konzeption, Konstruktion und Transition) an, kann eine zeitliche Gewichtung der einzelnen Disziplinen vorgenommen werden (vgl. Abbildung 9). Wie in Abschnitt 3.2.3 bereits erwähnt, entspricht die RUP-Prozessarchitektur im Wesentlichen einem Spiralmodell (siehe Abschnitt 3.1.2).

Abbildung 17: Überblick über die Disziplinen des RUP 2004 [Quelle: IBM-Rational]

Im Folgenden sollen kurz die Kerndisziplin „Analyse und Entwurf“ sowie die unterstützende Querschnittsfunktion „Projektmanagement“ exemplarisch anhand der jeweiligen Workflows vorgestellt werden. Abbildung 18 zeigt exemplarisch den Workflow der Kerndisziplin „Analyse und Design“ in Form eines UML-Aktivitätsdiagramms.

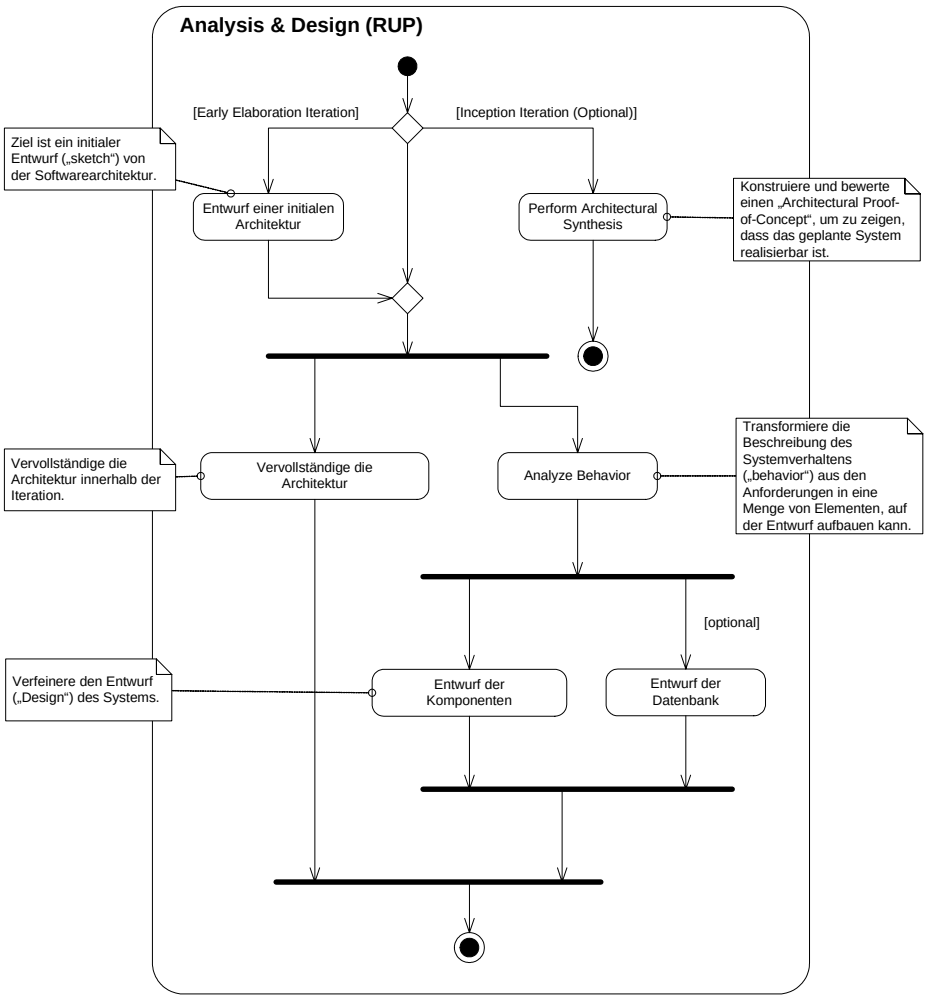

Abbildung 18: RUP-Disziplin "Analyse und Entwurf" mit Erläuterungen

Neben der Anforderungsanalyse (Requirements Engineering) handelt es sich dabei um die zentralen Aktivitäten im Rahmen des RUP. Der Ablauf wird von der Tatsache be-

stimmt, ob es sich um die erste oder eine der früheren Iterationen handelt. Ziel dieses Vorgehens ist die Erstellung, Verfeinerung und Validierung einer (komponentenbasierten) Softwarearchitektur (vgl. nächster Abschnitt). Die in Abbildung 18 dargestellten Schritte werden in der RUP-Dokumentation noch weiter verfeinert und mit Rollen assoziiert.

Abbildung 19 hingegen zeigt den Workflow der unterstützenden Disziplin „Projektmanagement“. Auch hier wird wiederum unterschieden, ob es sich um die erste Iteration bzw. den Projektstart oder um einen der folgenden Durchläufe handelt. Anhand der dargestellten Aktivitäten wird die Ähnlichkeit zum Spiralmodell von Boehm (1981, 1986, vgl. Abschnitt 3.1.2) sichtbar. Nach jeder Iteration bzw. Phase werden wie im Spiralmodell die Erfüllung der Aufgaben, der Handlungsspielraum sowie die Risiken bewertet und auf dieser Basis über den Fortgang des Projekts entschieden.

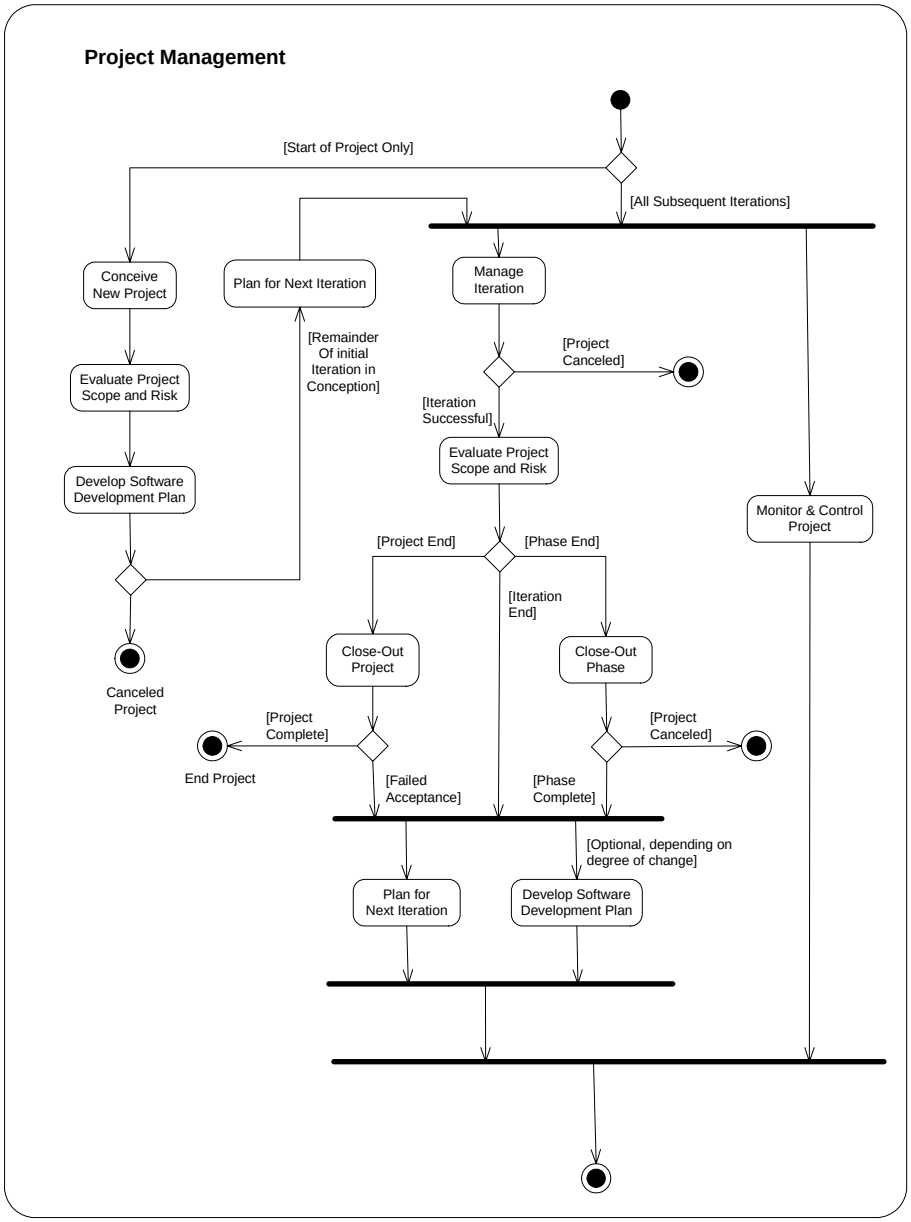

Abbildung 19: RUP-Disziplin "Projektmanagement" als Aktivitätsdiagramm

Komponentenverständnis

Im Gegensatz zu den vorhergehenden Versionen des RUP und dem ursprünglichen UP bietet der RUP eine verstärkte Unterstützung der komponentenbasierten Softwareerstellung. Der Ausdruck „Komponente“ wird als „*encapsulated part of a system, ideally a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture*“ definiert und die RUP-HTML-Dokumentation unterscheidet

- Designkomponenten (Subsysteme, Klassen und Pakete in Entwurfsform) und
- Implementationskomponenten (Quelltext, der die Design-Komponente umsetzt).

Von Kruchten werden zusätzlich Laufzeitkomponenten (Runtime Components), Komponenten aus Entwicklersicht (Development Components) und Geschäftskomponenten (Business Components) unterschieden (Kruchten (2004), S.94). Kruchten definiert Komponenten im Rahmen des RUP’04 als „nontrivial piece of software, a module, a package, or a subsystem“ mit den oben bereits genannten Eigenschaften (Kruchten (2004), S. 27).

Abbildung 20 zeigt die Aktivität „Komponentenentwurf“ aus dem Workflow „Analyse und Entwurf“ (vgl. Abbildung 18) in einer detaillierten eigenen Darstellung. Der Prozess sieht dabei vor, dass der Designer einzelne Klassen zu Subsystemen mit entsprechenden Schnittstellen aggregiert, die dann wiederum als Realisierung der zuvor spezifizierten Anwendungsfälle eingesetzt werden können. Neben dem Entwurf von Komponenten stellt auch die Nachbearbeitung und Überprüfung (*Review*) der Entwurfmodelle einen entscheidenden Baustein der RUP-Methodik dar.

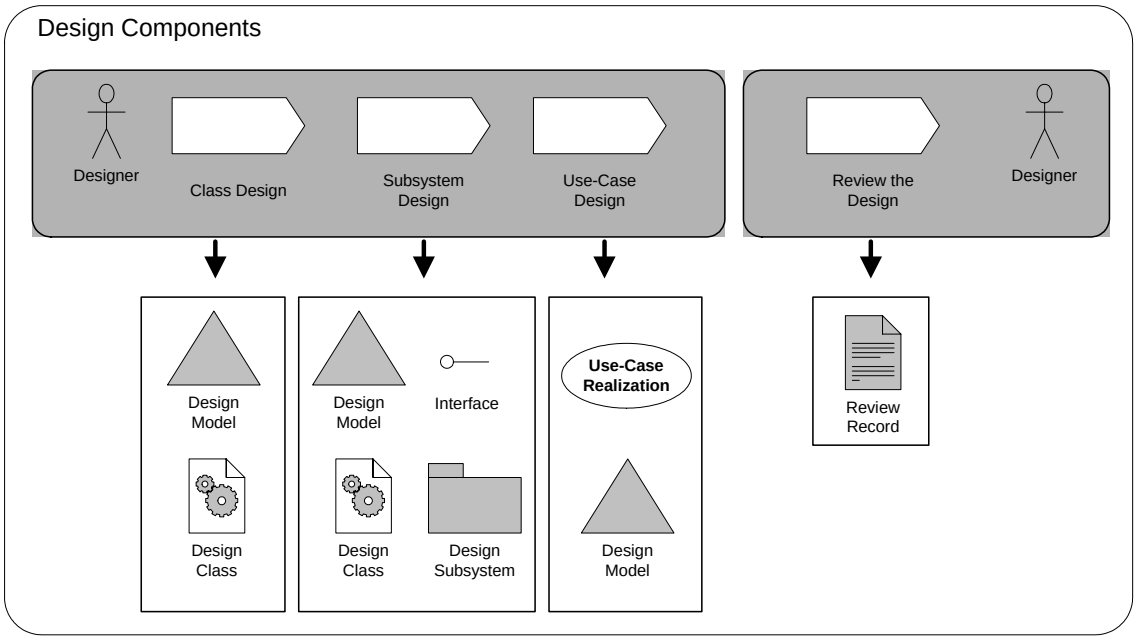

Abbildung 20: Der Komponentenentwurf beim RUP

Auch die grundlegenden Eigenschaften des RUP zeichnen ihn für ein komponentenorientiertes Vorgehen aus. So erleichtert der iterative Ansatz die schrittweise Identifikation und Verfeinerung von Komponenten. Der starke Fokus auf die Architektur der Lösung hilft zudem bei der Artikulierung der Anwendungsstruktur und Konzepte wie Pakete, Subsysteme und Klassen erleichtern die Formalisierung. Des Weiteren sieht der RUP’04 das inkrementelle Testen einzelner Komponenten bis hin zum gesamten System vor (vgl. Kruchten (2004)).

Kollaborationspunkte

Potenzielle Kollaborationspunkte beim RUP stellen alle Aktivitäten dar, die von zwei oder mehreren Rollen durchgeführt werden sollen. Dadurch, dass die einzelnen Workflows sowie deren Verfeinerungen immer im Wesentlichen durch die Basiselemente Aktivität (Activity), Rolle (Role) und Artefakt (Artifact) definiert werden (siehe Abbildung 1), eignet sich der RUP im Vergleich zu den bisher analysierten Modellen relativ gut, um kollaborative Softwareerstellungsprozesse abzubilden und zu unterstützen.

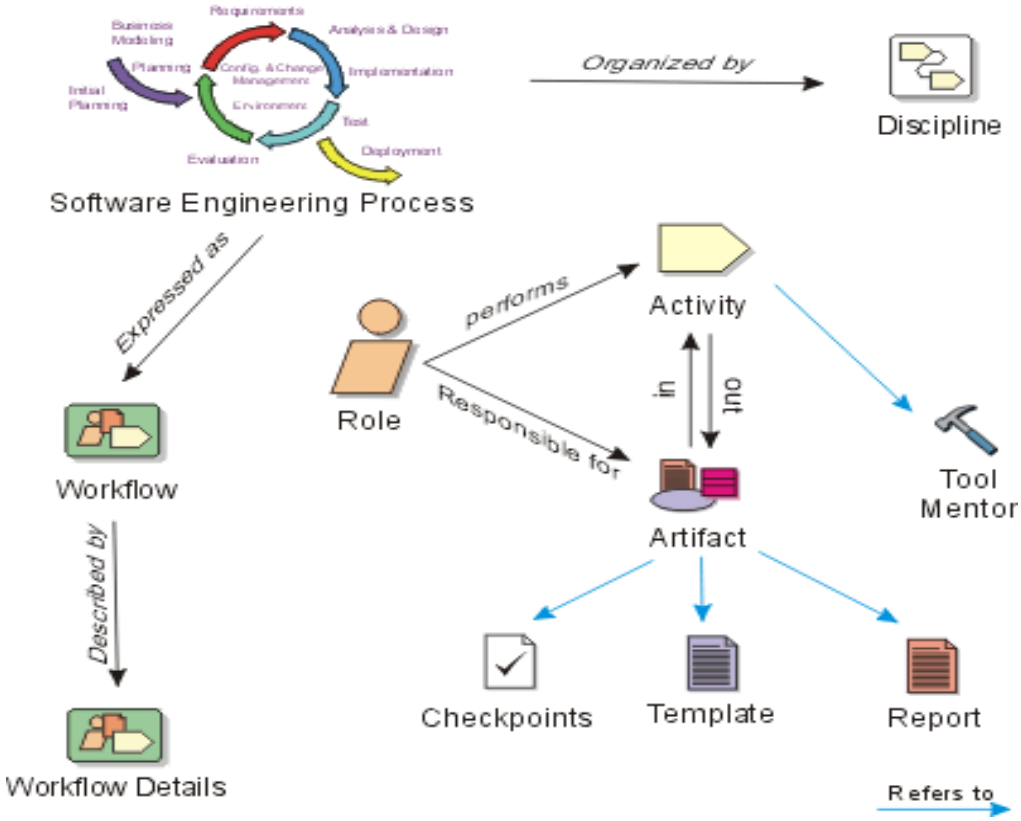

Abbildung 21: Die Basiselemente des RUP 2004 [Quelle: IBM-Rational]

Den einzelnen Aktivitäten können, anders als in anderen Vorgehensmodellen, immer auch Mitarbeiterrollen zugeordnet werden. Allerdings wurden die kollaborativen Aspekte der Softwareerstellung bei den vordefinierten Prozessbausteinen (Workflows) per se noch recht wenig berücksichtigt. Der RUP bietet somit keine eigenen Ansätze zur kollaborativen Softwareerstellung sondern bietet vielmehr einen Rahmen zur Gestaltung und Modellierung solcher Prozesse.

Adaptionsmöglichkeiten und Werkzeugunterstützung

Zur Prozesskonfiguration bietet der RUP grundsätzlich zwei Möglichkeiten: zum einen können die vordefinierten Workflows in ihren unterschiedlichen Detaillierungsebenen umgesetzt werden und zum anderen bietet das Vorgehensmodell fünf verschiedene „*Conceptual Roadmaps*“, die Standardkonfigurationen für unterschiedliche Szenarien bieten. Im Einzelnen bietet der RUP Konfigurationen für

- komponentenbasierte Lösungen (*Develop Component Solutions*),
- E-Business-Lösungen (*Develop E-Business Solutions*),
- benutzerfreundliche Lösungen (*Usability Engineering, User-Centered Design*),
- kleine Softwareprojekte (*Tailoring a Process for a Small Project*) und

- agile Entwicklungsmethoden (*Agile Practices and RUP*, vgl. Abschnitt 4.4).

Da der RUP parallel mit den Werkzeugen der IBM Rational Suite gestaltet wurde und immer weiter entwickelt wird, bietet dieses Vorgehensmodell als wahrscheinlich einziges eine durchgängige Werkzeugunterstützung. Den unterschiedlichen Benutzern der RUP-HTML-Dokumentation wird durch den so genannten „Tool Mentor" rollenbasierte Unterstützung bei der Auswahl der einzusetzenden Werkzeuge gegeben und auf die entsprechenden Lösungen der Firma IBM (ehemals Rational) verwiesen. Diese werden ausführlich in dem Arbeitspapier „Werkzeuge zur kollaborativen Softwareerstellung – Stand der Technik" von Geisser et al. (2006) beschrieben. Zusätzlich existieren mit dem RUP Modeler, einem Rational XDE Plugin zur Prozessmodellierung, dem RUP Organizer zur Verwaltung und Erstellung von Inhalten sowie dem RUP Builder zum Zurechtschneiden des RUP drei dedizierte Werkzeuge zur Modelladaption.

Zusammenfassung

Auf eine NIMSAD-Analyse des RUP wird an dieser Stelle verzichtet, da dies bereits in Abschnitt 3.2.3 für den USDP durchgeführt wurde. Auf der Abstraktionsebene der NIMSAD-Betrachtung (vgl. Anhang 1) wirken sich die Änderungen und Erweiterungen des RUP'04 gegenüber dem USDP nicht aus. Des Weiteren bietet der Rational unter der Bezeichnung „IBM Rational Method Composer" (RMC) bereits aktuell eine Nachfolgerversion des RUP an[4]. Diese konnte aufgrund Ihrer Aktualität im Rahmen dieser Arbeit nicht mehr berücksichtigt werden.

3.3.2 V-Modell XT

Das V-Modell XT (eXtreme Tailoring, VMXT) ist ein Prozessmodell und eine Projektmanagementmethode zur Planung und Implementierung von IT-Projekten. Es beschreibt alle benötigten und durchzuführenden Tätigkeiten eines Softwareprojekts. Es kann somit als eine Richtlinie für die IT-Projektleitung genutzt werden. Die Verwendung des VMXT soll die Transparenz des Projektverlaufs steigern, das Projekt selbst überschaubarer machen und die Projekterfolgsquote signifikant steigern[5]. Im Vergleich zum Vorgänger, dem V-Modell'97, kommt dem Projektmanagementaspekt eine größere Bedeutung zu.

Philosophie und Terminologie

Das V-Modell XT unterscheidet sich von den bisher untersuchten Prozessmodellen auf drei verschiedene Arten:

- es bietet mehrere Prozessmodelle bzw. Projekttypen statt nur einem,
- es fördert und unterstützt die Maßanfertigung der Prozessmodelle auf die spezifischen Bedürfnisse der Organisation und des Projekts hin, und
- es deckt sowohl produktorientierte Prozesse als auch Projektleitungsprozesse ab.

Die unterstützten Prozessmodelle sollen drei verschiedenen **Projekttypen** dienen:

1. Systementwicklungsprojekt eines Auftraggebers,
2. Systementwicklungsprojekt eines Auftragnehmers und

---

[4] siehe http://www.ibm.com/software/news/n/sdsd6hahqa (2005-12-09) bzw.

http://www-306.ibm.com/software/swnews/swnews.nsf/n/sdsd6hahqa (2005-12-09)

[5] vgl. http://ftp.uni-kl.de/pub/v-modell-xt/Release-1.1/Dokumentation/pdf/V-Modell-XT-Komplett.pdf (2005-12-09)

3. Einführung und Pflege eines organisationsspezifischen Prozessmodells.

Es wird zwischen „Auftraggeber“ (Anwenderunternehmen) und „Auftragnehmer“ (Systemintegrator oder Softwarehaus) unterschieden, da zu jedem Softwareentwicklungsprojekt ein Auftraggeber und ein Auftragnehmer gehören. Prozessmodelle, auch das V-Modell’97, haben sich bisher auf (Software-)produktorientierte Prozesse konzentriert. Diese Prozesse werden auf der Auftragnehmerseite ausgeführt. Jedoch sind Projektmanagementprozesse, die auf beiden Seiten unternehmensübergreifend durchgeführt werden, kritisch für den Projekterfolg. Indem es Prozessunterstützung sowohl für die Anbieter- als auch für die Käuferseite anbietet, löst das VMXT das Problem der einseitigen Betrachtung der Auftragnehmerseite. Außerdem schreibt es vor, wie die notwendige Interaktion und Zusammenarbeit zwischen Systemintegrator und Anwender durchgeführt werden sollten. Auf diese Weise kann die Erfolgsrate der Projekte signifikant gesteigert werden.

Modellumfang, Adaptionsmöglichkeiten und Werkzeugunterstützung

Das Maßschneidern der Prozessmodelle („*Tailoring*“) wird auf unterschiedliche Arten gefördert und unterstützt. Das VMXT bietet zum einen 18 unterschiedliche Prozessbausteine, die benutzt werden können, um spezifische Prozessmodelle zu entwerfen. Bei der Verwendung von projekteigenen Profilen ist bereits festgelegt, welche Bausteine zwingend und welche optional sind. Zusätzlich wird das Projektprofil benutzt, um den Bestand der unterstützten Modelle einzuengen. Zudem werden *Open Source* Software-Werkzeuge angeboten, um die Maßanfertigung (VMXT Editor) und das Projektmanagement (VMXT Projektassistent) zu unterstützen. Außerdem liefert das VMXT eine Methodik zur Implementierung und Aufrechterhaltung der speziell angefertigten Prozessmodelle. Allein das Hauptdokument der Version 1.1 des VMXT umfasst 533 DinA4-Seiten.

Prozessarchitektur und –steuerung

Das VMXT besteht aus 18 unterschiedlichen Prozessbausteine

1. Projektmanagement
2. Qualitätssicherung
3. Problem- und Änderungsmanagement
4. Konfigurationsmanagement
5. Kaufmännisches Projektmanagement
6. Messung und Analyse
7. Einführung und Pflege eines organisationsspezifischen Vorgehensmodells
8. Anforderungsfestlegung
9. Auftragsvergabe, Projektbegleitung und –abnahme
10. Angebotserstellung und Vertragserfüllung
11. Systemerstellung
12. Softwareentwicklung
13. Hardwareentwicklung
14. Benutzbarkeit und Ergonomie
15. Systemsicherheit
16. Weiterentwicklung und Migration von Altsystemen

17. Evaluierung von Fertigprodukten

18. Logistikkonzeption

Diese Bausteine sollen alle potentiellen Probleme abdecken, die in einem Softwareentwicklungsprojekt auftreten können – ganz egal, ob es sich um ein Auftragnehmer- oder Auftraggeberprojekt handelt. Sie beschreiben sehr detailliert standardisierte Tätigkeiten, durchzuführende Arbeiten sowie die Rollen einschließlich ihrer Verantwortlichkeiten. Auf diese Weise soll sichergestellt werden, dass kein wichtiger Vorgang vergessen wird. Da es die Prozesse mit so vielen Details beschreibt, ist das VMXT sehr viel umfassender als die meisten – wenn nicht sogar alle – anderen Prozessmodelle, was sich unter anderem an der mehrere tausend Seiten umfassenden Dokumentation manifestiert. Dennoch ist es nicht monolithisch. Vielmehr ist das VMXT dadurch, dass es Bausteine anbietet, die durch zusätzliche Bausteine ergänzt oder die angepasst werden können, sehr flexibel.

Komponentenverständnis und Kollaborationspunkte

Die Unterstützung der komponentenbasierten Softwareerstellung stellt eine der wesentlichen Neuerungen des VMXT gegenüber dem Vorgänger VM'97 dar. Eine Komponente wird als „eine eigenständig einsetzbare Einheit mit Schnittstellen nach außen, die Entwurf und Implementierung kapselt und mit anderen Komponenten verbunden werden kann", verstanden. „Sie ist sowohl fachlich als auch technisch unabhängig und besitzt eine gewisse Größe im Sinne eines wirtschaftlichen Werts. Allgemein werden von einer Komponente folgende Eigenschaften verlangt: Verfügbarkeit klarer, sauber definierter Schnittstellen, Kommunikation mit der Außenwelt (zum Beispiel mit anderen Komponenten) ausschließlich über die definierten Schnittstellen, Anpassung an bestimmte Anwendungsumgebungen (Customizing) nur über die Schnittstellen und Realisierungsspezifika bleiben dem Benutzer verborgen (Blackbox-Sichtweise)" (VMXT (2005)).

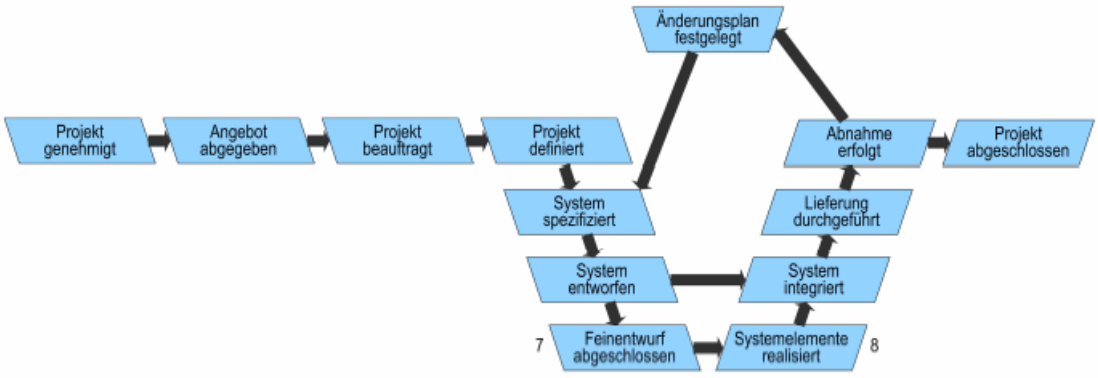

Abbildung 22: Projektdurchführungsstrategie für KBSE im VMXT

Abbildung 22 fasst noch einmal die Projektdurchführungsstrategie des VMXT für die komponentenbasierte Softwareerstellung zusammen. Der charakteristische „V-Knick" ist dabei klar erkennbar.

Das VMXT bietet somit detaillierte Richtlinien für Softwareentwicklungsprojekte. Es deckt sowohl produktorientierte Prozesse als auch Projektmanagementprozesse ab. Die Verwendung des VMXT macht es einfacher, Softwareentwicklungsprojekte zu planen. Das liegt hauptsächlich an der detaillierten und strukturierten Beschreibung der benötigten Aktivitäten, durchzuführenden Arbeiten, Rollen und Verantwortlichkeiten. Bei Verwendung des Modells VMXT ist es unwahrscheinlich, notwendige Tätigkeiten oder durchzuführende Arbeiten bei der Planung eines Projekts zu versäumen. Die Transparenz wird erhöht und der Bedarf, Pläne nachträglich zu korrigieren, wird signifi-

kant reduziert. Außerdem bietet das VMXT explizit Beschreibungen der Tätigkeiten und durchzuführenden Arbeiten für die Projektplanung. Diese können als eine Planungsrichtlinie benutzt werden. Da das VMXT Auftraggeber- und Auftragnehmersicht integriert ist es insbesondere für die Kollaboration auf Unternehmensebene geeignet.

3.3.3 Bewertung und Gegenüberstellung von RUP'04 und VMXT

Beim RUP werden, im Gegensatz zum VMXT, die einzelnen Aktivitäten des Vorgehensmodells den definierten Rollen zugeordnet. Beim VMXT hingegen werden diese den zu erstellenden und bearbeitenden Produkten zugeordnet. Will man Kollaborationsbeziehungen zwischen unterschiedlichen Mitarbeitern und Organisationseinheiten abbilden, eignet sich ein Aktivitäten/Rollen-Modell prinzipiell besser als eine produktbezogene Variante. Die Stärke des V-Modells liegt in der Unterstützung von Projektmanagementaktivitäten, was es grundsätzlich für große, zwischenbetriebliche Softwareprojekte qualifiziert.

Tabelle 7 fasst noch einmal die wesentlichen Unterschiede der beiden Vorgehensmodelle entlang der Dimensionen des hier gewählten Vergleichsrahmens zusammen. Der unterschiedliche und teilweise komplementäre Charakter manifestiert sich bereits an der Gegenüberstellung der zu Grunde liegenden Philosophien. Während der RUP versucht, etablierte Konzepte aus dem Software Engineering konsequent umzusetzen, zielt das VMXT primär auf Qualitätsverbesserung in Prozess und Produkt ab. Des Weiteren soll das VMXT in der aktuellen Version für sehr unterschiedliche Arten von Softwareprojekten einsetzbar und anpassbar sein. Sowohl in der Abdeckung des Softwarelebenszyklus und der Querschnittsfunktionen als auch bezüglich der Dokumentation des Vorgehensmodells sind beide Modelle als umfangreich und schwergewichtig einzustufen, wobei das VMXT eher noch die Querschnittsfunktionen Projekt- und Qualitätsmanagement hervorhebt. Obwohl Terminologie und Metamodell (vgl. Abbildung 23) auf den ersten Blick sehr ähnlich wirken, besteht der größte Unterschied zwischen RUP und VMXT in der Prozesssteuerung: Während der RUP entlang der definierten Workflows und deren Aktivitäten abläuft, stehen beim VMXT die Artefakte (Ergebnisse) im Zentrum der Betrachtung.

Vergleichskriterium	RUP 2004	V-Modell XT (2005)
Philosophie	iterativ, architekturzentriert und anwendungsfallgetrieben	qualitätsorientiert und anpassbar
Modellumfang	umfangreiche Prozessabdeckung und Dokumentation	umfangreiche Prozessabdeckung, aber Fokus auf Projektmanagement
Prozessarchitektur-/steuerung	zeitliche Gewichtung unterschiedlicher Workflows (Spiralmodell), aktivitätsorientierte Prozesssteuerung	eher ergebnisorientierte Steuerung
Komponentenverständnis	KBSE fester Bestandteil der Methodik	Wiederverwendung und Integration von „Fertigprodukten" möglich
Kollaborationspunkte	Zuordnung Aktivitäten-Rollen gut geeignet	auf Organisationsebene (Auftraggeber/Auftragnehmer), produktzentrierte Sicht sonst eher ungeeignet
Adaptionsmöglichkeiten	Tailoring wird methodisch und softwaretechnisch unterstützt	Tailoring wird methodisch und softwaretechnisch unterstützt
Werkzeugunterstützung	RUP-Anpassungswerkzeuge und Rational Suite über Tool Mentor	VMXT-Editor/-Projektassistent

Tabelle 7: Vergleich des RUP'04 und des VMXT

Das Komponentenkonzept verinnerlichen beide Modelle, wobei beim RUP die KBSE auf Grund der Entstehung im OMG-Umfeld stärker verwurzelt ist. Das VMXT kennt so genannte „Fertigprodukte“ und diese können standardmäßig integriert werden. In puncto Unterstützung der kollaborativen Softwareerstellung zeigen beide Modelle unterschiedlichen Stärken und Schwächen: Durch die Möglichkeit, jeder Aktivität im RUP eine oder mehrere Rollen zuzuordnen, wird hier die Kollaboration auf Projekt- bzw. Teamebene besonders gut unterstützt. Die Stärke des VMXT besteht andererseits darin, dass ein integriertes Vorgehensmodell für Auftraggeber und Auftragnehmer bereitgestellt wird, was die zwischenbetrieblichen Aspekte der Kollaboration besser abbildet. Beide Methodiken stellen sowohl Richtlinien als auch Werkzeuge zur Anpassung (Tailoring) der vorgegebenen Prozessbausteine und Workflows zur Verfügung. Darüber hinaus bietet der RUP eine nahtlose Integration mit der umfangreichen Produktpalette an Softwareentwicklungswerkzeugen von IBM Rational.

Abbildung 23 zeigt noch einmal einen Vergleich der dem RUP und dem VMXT zu Grunde liegenden Prozessmetamodelle (vgl. auch Abbildung 1). Hieran lassen sich die jeweiligen Terminologien sowie die bereits analysierte Prozessarchitektur und –steuerung veranschaulichen.

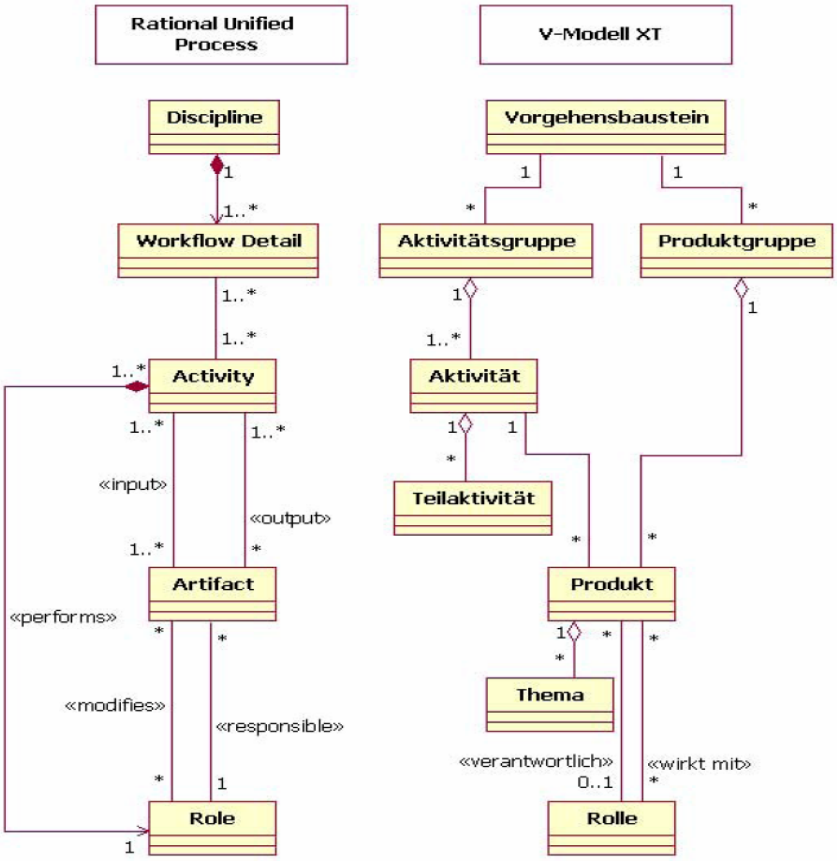

Abbildung 23: Vergleich der Metamodelle von RUP und VMXT (Grundmann (2005), S. 38)

Der Zusammenhang zwischen Disziplinen, Workflow-Details und Aktivitäten im RUP kann als analog zu dem der Vorgehensbausteine, Aktivitätsgruppen und Aktivitäten betrachtet werden. Auch werden Artefakte bzw. Produkte in beiden Modellen mit entsprechenden Rollen assoziiert. Der bereits erwähnte Unterschied besteht darin, dass beim RUP auch die Aktivitäten mit mindestens einer Rolle verbunden sind, was die Modellierung kollaborativer Abläufe, an denen mehrere Entwickler und/oder Anwender beteiligt sind, zulässt.

Aus der Integration dieser beiden Ansätze könnte sich das Prozessmodell der Zukunft formieren. Erste Ansätze hierzu finden sich bei Grundmann (2005) [6].

4 Weitere methodische Ansätze

Ansätze zur kollaborativen Softwareerstellung (KSE) können für unterschiedliche organisatorische Ausmaße bzw. Verteilungsszenarien konzipiert sein. Im Folgenden werden exemplarisch unterschiedliche, aus der Literatur bekannte Ansätze, die sich nicht als „echtes" Vorgehensmodell qualifizieren, vorgestellt und analysiert:

Zum einen werden spezielle prozessorientierte Ansätze betrachtet, die sich auf einzelne Phasen bzw. Disziplinen des SE beziehen. Auf der anderen Seite werden agile Entwicklungsmethodiken, hier insbesondere das Extreme Programming (XP), zur intensiven Zusammenarbeit auf Teamebene, sowie etablierte Entwicklungsmethodiken aus dem Open Source-Umfeld (große, global verteilte Projekte in unterschiedlichen Domänen) diesem gegenübergestellt. Die prozessorientierten Ansätze betrachten weniger soziotechnische Fragestellungen, sondern legen mehr Wert auf einen formal strukturierten Softwareerstellungsprozess, bspw. in Form eines Prozess- oder Vorgehensmodells. Zur Strukturierung dienen hierbei in erster Linie die einzelnen Phasen bzw. Disziplinen des allgemeinen Softwarelebenszyklus, welcher sich entlang vertikaler Wertschöpfungsstufen von den Anforderungen hin zur fertigen Software erstreckt. Als gemeinsamer Nenner aus unterschiedlichen Vorgehensmodellen werden Arbeiten mit kollaborativem Charakter aus den Bereichen (1) Anforderungsanalyse, (2) Entwurf und Modellierung sowie (3) Implementierung, Test und Wartung unterschieden. In der SE-Literatur findet man auch erste Ansätze zur Unterstützung der horizontalen Zusammenarbeit innerhalb einzelner Phasen, wohingegen bisher keine der etablierten Entwicklungsmethodiken und Prozessmodelle die vertikal durchgängige Zusammenarbeit behandelt (Hildenbrand und Korthaus (2004), vgl. auch Analyse der modernen und aktuellen Vorgehensmodelle in den vorhergehenden Abschnitten).

4.1 Kollaborative Methoden in der Anforderungsanalyse

Neue Formen der Zusammenarbeit wurden auf methodischer Ebene bisher vor allem im Requirements Engineering, d.h. bei Anforderungserhebung, -spezifikation und –analyse, entwickelt und erprobt. Da in dieser frühen Phase von Softwareprojekten die Interessen vieler Anspruchsgruppen (Stakeholder) koordiniert werden müssen (Sommerville (2004)), existieren im Bereich „Collaborative Requirements Engineering" einige Ansätze, die sich im Wesentlichen in den Verteilungsdimensionen sowie der Werkzeugunterstützung unterscheiden. Speziell für die verteilte Anforderungserhebung existieren zahlreiche empirische Studien, die die Vorteilhaftigkeit der geografischen Verteilung und dabei die Unterschiede zwischen synchroner und asynchroner Anforderungserhebung untersuchen (DaEb (2000), Damian (2001), Lloyd u. a. (2002), Damian et al. (2003)).

Basierend auf dem Harvard-Konzept der Verhandlungstechnik wurde von Boehm und Ross die Theory W entwickelt (Boehm und Ross (1989)). Hierbei sollen in erster Linie asymmetrische Win-Lose-Situationen zwischen einzelnen Stakeholdern durch systematische Verhandlungs- und Moderationstechniken vermieden werden, um das Ge-

[6] Es handelt sich hierbei um eine Diplomarbeit, die an der Hochschule Reutlingen in Zusammenarbeit mit IBM-Rational entstand:

ftp://ftp.uni-kl.de/pub/v-modell-xt/Release-1.1/Infomaterial/Prozessmodellintegration.pdf

samtrisiko eines Softwareprojekts zu minimieren. Eine Anwendung dieser Theorie und gleichzeitig Weiterentwicklung des Spiralmodells (Boehm (1981)) stellt das WinWin-Spiralmodell dar, welches zunächst als „*Next Generation Process Model*“ bezeichnet wurde (Boehm und Bose (1994)). Dabei werden die sich verändernden Interessen unterschiedlicher Stakeholder sowie deren Koordination in jeder Phase über mehrere Iterationen hinweg berücksichtigt.

Die EasyWinWin-Methodik von Boehm, Briggs und Gruenbacher stellt die mittlerweile vierte Generation des WinWin-Ansatzes zur kollaborativen Anforderungserhebung dar (Gruenbacher (2000), Briggs und Gruenbacher (2002)). Die Methodik baut auf dem WinWin-Spiralmodell auf und integriert weitergehende Erkenntnisse aus den Bereichen „Interpersonelle Beziehungen“ und „Erfolgsmanagement“. EasyWinWin zeichnet sich u.a. durch einen flexiblen, iterativen Erhebungsprozess aus, der es ermöglicht, neue Stakeholder im laufenden Prozess zu integrieren. Außerdem wird der Aufbau von gegenseitigem Vertrauen durch physische Treffen und moderierte Diskussionen gefördert. Allerdings ist die Methodik komplex, teilweise nicht sehr intuitiv und trotz entsprechender Groupware primär nicht für ein verteiltes Umfeld konzipiert.

Groupware-basierte Werkzeugunterstützung für EasyWinWin ermöglicht mittlerweile Komplexitätsreduktion, in Ansätzen verteiltes Arbeiten und asynchronen Informationsaustausch, beispielsweise durch die Protokollierung von Diskussionen (Boehm und Gruenbacher (2001), Boehm et al.(2001)). Die webbasierte Variante ARENA erlaubt ausschließlich asynchrones sowie verteiltes und mobiles Zusammenarbeiten bei der Anforderungserhebung. Allerdings ist ARENA nicht mit ursprünglichen WinWin-Groupware kombinierbar (Gruenbacher und Braunsberger (2003), Seyff u.a (2004), Geisser und Hildenbrand (2005)).

4.2 Kollaborative Entwurfs- und Modellierungsmethoden

Kollaborative Ansätze innerhalb der Entwurfs- und Modellierungsphase wurden aus anderen Ingenieurdisziplinen übertragen (z.B. Computer Aided bzw. Concurrent Design Prozesse im Automobilbau). Bisher befinden sich die meisten dieser Ansätze im SE, u.a. aufgrund der speziellen Eigenschaften von Software, in einem prototypisch-experimentellen Stadium.

Joint Application Design (JAD) beschreibt eine Methodik für rechnergestützte Treffen und Workshops mit der Zielsetzung, aus den teilweise unterschiedlichen Anforderungen der Stakeholder zu einem gemeinsamen ersten Entwurf der Systemarchitektur zu kommen. Charakteristisch für JAD-Treffen sind u.a. ein fest definierter Zeitrahmen, eine strukturierte Umgebung mit visuellen Hilfsmitteln, ein Moderator und ein Protokollant (WoSi (1989), CaNu (1992), Avison und Fitzgerald (2003)). Im Gegensatz zu JAD ist der Participatory Design-Ansatz (PD) weniger ergebnisorientiert und fokussiert primär die intensive synchrone Zusammenarbeit zwischen Entwicklern und späteren Systemanwendern beim Entwurf der Software. Dabei wird nicht geografisch verteilt gearbeitet und es sind keine bestimmten Werkzeuge vorgegeben (Carmel et al. (1993), Altmann (1999)).

Concurrent Software Engineering (CCSE) wurde ursprünglich durch den Ansatz von Dewan und Riedl geprägt (Dewan und Riedel (1993)). Im Gegensatz zu JAD und PD sollen beim Concurrent Software Engineering möglichst viele Entwicklungsaktivitäten parallel bzw. überlappend ablaufen. Das bedeutet, dass dabei nicht notwendigerweise synchron zusammengearbeitet wird (Altmann (1999), Graham et al. (1999), Hildenbrand und Korthaus (2004)). Dean et al. hingegen betrachten mit ihrer Collaborative Software Engineering Methodology (CSEM) ebenfalls die nebenläufige, asynchrone Zusammenarbeit unterschiedlicher Stakeholder im Rahmen des Softwareentwurfspro-

zesses. Insbesondere Nutzer sollen in drei Stufen eingebunden werden: Zunächst einige wenige Repräsentanten, dann kleinere Benutzergruppen und schließlich die gesamte Nutzergemeinschaft. Im Gegensatz zum CCSE sieht CSEM eine hohe Intensität der Zusammenarbeit im Rahmen von regelmäßigen physischen Treffen zum Abgleich der Ergebnisse vor (Dean et al.(1998)).

Nebenläufige Entwurfsprozesse werden auch immer stärker in UML-basierten Methoden zur modellgetriebenen Softwareentwicklung (Model-Driven Development, MDD) integriert. Wenn auch bisher nur wenige fundierte Methodiken existieren, ermöglichen moderne Entwicklungswerkzeuge, wie beispielsweise Poseidon for UML Enterprise (Gentleware (2005)), Konesa (CanyonBlue (2005)) und Rational XDE (IBM (2005), IBM (2005a)), nebenläufige, verteilte und modellgetriebene Softwareerstellungsprozesse. Neben Vorgehensweisen zur Entwurfserstellung existieren des Weiteren werkzeuggestützte Ansätze zur Evaluation von Softwarearchitekturen. Hierbei sind in erster Linie die Arbeiten von Babar et al. zu nennen, die Architekturevaluationsverfahren für die verteilte Zusammenarbeit weiterentwickeln (Distributed Software Architecture Evaluation Process, (Babar (2004), Babar und Gorton (2004))).

4.3 Kollaborative Implementierung, Test und Wartung

Bei den Ansätzen zur kollaborativen Implementierung, Test und Wartung herrscht im Allgemeinen eine höhere Werkzeugorientierung vor als in den beiden bisher behandelten Disziplinen. Zusammenarbeit in der Implementierungsphase kann sowohl synchron, in Form von Shared Workspaces bzw. Application Sharing (z.B. Collaborative Editing), als auch asynchron, gepuffert über Versionsmanagementsysteme (Code Repositories), unterstützt werden (Dewan und Riedl (1993). Da auch hier häufig methodische Aspekte fehlen, soll auf diese und weitere Werkzeuge detaillierter in den nachfolgenden Abschnitten eingegangen werden.

Mit der CAIS-Methode (Collaborative Asynchronous Inspection of Software, (Mashayekhi et al. (1993), Mashayekhi et al. 1994)) existiert ein verteilter Ansatz zur kollaborativen Qualitätssicherung als Teil der Postimplementierungsphase (Stein et al. (1997)). Ebenso in diese Kategorie fallen Ansätze zur verteilten und asynchronen Wartung bereits entwickelter Software (Collaboration in Software Maintenance) (Lougher und Rodden (1993)). Andere Arbeiten versuchen, die Arbeitsabläufe in der Softwareentwicklung durch formale Modelle, z.B. Petrinetze, abzubilden und zu optimieren (Oberweis (1994), Oberweis u.a (1994)). Viele Autoren unterscheiden hierbei prozessorientierte und produktorientierte Ansätze zur kollaborativen Softwareerstellung (Altmann und Weinrich (1998), Ballarini et al.(2003)), welche im folgenden Abschnitt im Rahmen der werkzeugzentrierten Ansätze behandelt werden.

4.4 Agile Softwareentwicklung

Als Gegenbewegung zu den bisher analysierten, stark vorschreibenden Methodiken, wie z.B. dem UP und Catalysis, schlossen sich 17 Softwaretechniker zusammen, um das „Manifesto for Agile Software Development“ als Ausgangspunkt der agilen Softwareentwicklung (ASE) zu verfassen (Agile Alliance (2001)). Unter diesen Leuten befinden sich Wissenschaftler und Autoren wie Kent Beck, Alistair Cockburn, Martin Fowler und Steve Mellor. Beim agilen Softwareentwicklungsansatz handelt es sich nicht, wie oft fälschlicherweise angenommen, um eine „Anti-Methodik“ (Fowler und Highsmith (2001)), sondern um eine „leichtgewichtige“ Herangehensweise, welche auf Kundenzufriedenheit durch frühe und kontinuierliche Auslieferung von brauchbaren Softwarekomponenten abzielt – Aspekte methodischer Wiederverwendung werden dabei in der Regel nicht berücksichtigt. Zur Gruppe der agilen Entwicklungsansätze werden neben

XP, als bekanntestem Vertreter, die Crystal-Methodiken, SCRUM, Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Feature-Driven Development (FDD), Pragmatic Programming und Agile Modeling (AM) gezählt.

4.4.1 Idee

Die Grundidee der agilen Softwareentwicklung besteht in einer „(Rück-)Besinnung“ auf bestimmte Grundwerte der SWT (Hruschka (2003)). Zu diesem Zweck wurde von der Agile Alliance eine „Umgewichtung“ der Bedeutsamkeit von Softwareentwicklungskonzepten in Form von vier Value Statements vorgenommen: Menschen und Zusammenarbeit vor Prozessen und Werkzeugen, lauffähige Software vor umfangreicher Dokumentation, Zusammenarbeit mit Auftraggebern vor Vertragsverhandlungen und Reagieren auf Änderungen vor strikter Befolgung eines Plans (vgl. Agile Alliance (2001) sowie Fowler und Highsmith (2001)).

Aufbauend auf diesem Wertesystem werden 13 grundlegende Prinzipien der agilen Softwareentwicklung formuliert, u.a. dass Kunde bzw. Auftraggeber und Entwickler eng zusammenarbeiten, viel Wert auf persönliche Kommunikation gelegt wird, selbst organisierende Teams gebildet werden und häufig über die angewandte Methodik reflektiert bzw. diese angepasst werden soll (Fowler und Highsmith (2001) und Cockburn (2003)). Nachfolgend werden zwei agile Methodiken vorgestellt, welche diese Prinzipien implementieren und erweitern.

4.4.2 Extreme Programming und Refactoring

Die beiden agilen Konzepte XP (Beck (2000)) und Refactoring (Fowler und Beck (1999)) sind eng miteinander verzahnt und haben in der SWT-Literatur und -Praxis in den letzten Jahren großes Aufsehen erregt. Bei den Refactorings handelt es sich, ergänzend zu der XP-Programmiermethodik, um kombinierbare Operatoren bzw. standardisierte Transformationsregeln, mit deren Hilfe man Quellcode unterschiedlicher Herkunft, aber in derselben objektorientierten Programmiersprache, schrittweise in eine einheitliche, semantisch aussagekräftigere Form bringen kann. Somit lassen sich vorhandene Codefragmente nachbearbeiten und umstrukturieren sowie Implementierungskonventionen für neuen Quellcode festlegen (Fowler und Beck (1999)).Im Folgenden soll die XP-Methodik kurz mit dem in dieser Arbeit entworfenen Vergleichsrahmen analysiert werden, um den Kontrast zu den anderen Prozessmodellen aus Kapitel 4 zu verdeutlichen, wobei besonderes Augenmerk dabei auf dem Aspekt „Kollaborationspunkte“ liegt.

4.4.3 Einordnung in den Vergleichsrahmen

Obwohl es sich bei den Methoden und Ansätzen der ASE nicht um Vorgehensmodelle im eigentlichen Sinn handelt, lässt sich doch der in dieser Arbeit verwendete Vergleichsrahmen (vgl. Abschnitt 2.1.3) sehr gut anwenden.

Philosophie: Beck (2000) führt die gesamte Methodik von XP auf die vier Werte Kommunikation, Einfachheit, Feedback und Mut zurück. Kommunikation und Feedback, vor allem durch häufige Tests, verdeutlichen besonders den kollaborativen Charakter von XP. Einfachheit soll ausdrücken, dass immer die Frage „Wie gut ist gut genug?“ mitschwingt, was dem allgemeinen agilen Prinzip „maximizing the amount of work not done“ (Fowler und Highsmith (2001), S. 5) entspricht und das in der SWT häufig beobachtbare Overengineering verhindern soll. Mut zu eigenständigen Entscheidungen, sowie Respekt und Interesse an Projekt und Teamkollegen wird ebenfalls vorausgesetzt. Da mit „Kommunikation“ auch hauptsächlich der persönliche Austausch von Informationen gemeint ist, zeigt sich XP sehr „people-centric“ (Beck (2000)).

Modellumfang: Neben Beck (2000) und Beck und Fowler (2001) existieren aufgrund des hohen Bekanntheitsgrads zahlreiche Dokumentationen und Internetseiten, die die XP-Methodik beschreiben. Es werden alle wesentlichen Phasen des Softwarelebenszyklus eingebunden, wobei jedoch der Schwerpunkt eindeutig auf der codezentrierten Implementierung und entsprechenden Tests liegt. Querschnittsfunktionen, wie „Projektmanagement“ und „Qualitätssicherung“, werden auch nicht vernachlässigt. Beck (2000) formuliert überdies ein kompaktes Rollenmodell, bei dem vor allem die Rolle des „Coach“, als „Trainer“ und Betreuer der „Entwicklungsmannschaft“, charakteristisch ist.

Prozessarchitektur: Die „Prozessarchitektur“ ist bei XP gemäß dem agilen Ansatz nicht strikt vorgegeben, orientiert sich aber an den fünf Grundprinzipien „unmittelbares Feedback“, „Einfachheit anstreben“, „inkrementelle Veränderung“, „Veränderung wollen“ und „Qualitätsarbeit“ (Beck (2000), S. 37ff). Es ergibt sich daraus ein zyklischer Prozess mit den Hauptaktivitäten Programmieren, Testen, Zuhören (den Geschäftsleuten/Auftraggebern) und Designentwurf (Beck (2000), S. 50). Loses Abbruchkriterium dieser Iteration ist die Erfüllung aller erdenklichen Komponenten- und Systemtests. Man kann XP daher als testgetriebenes Modell klassifizieren, wobei die Prozesssteuerung teilweise auch durch die freie Interpretation der Prinzipien entscheidungsorientiert erfolgen kann (vgl. hierzu Catalysis in Abschnitt 4.4.5).

Komponentenverständnis und **Kollaborationspunkte:** Da XP nicht auf methodische Wiederverwendung abzielt, werden Softwarekomponenten lediglich als Mittel zur arbeitsteiligen Entwicklung und zur Strukturierung durch einzelne Komponententests betrachtet (vgl. Szyperski u. a. (2002)). XP wurde vornehmlich für den Einsatz in lokalen Teams und eher kleineren Projekten konzipiert und schlägt daher Praktiken zur regelmäßigen und intensiven persönlichen Kommunikation aller Projektbeteiligten vor – inklusive der Vertreter des Auftraggebers. Beispiele für solche Praktiken (Techniken) sind: Programmieren in Paaren (Pair Programming), Gemeinsame Verantwortlichkeit und Kunde vor Ort (Beck (2000), S. 66ff.).

Die Vergleichskategorien „Terminologie“, „Adaptionsmöglichkeiten“ und „Werkzeugunterstützung“ spielen bei der Analyse von XP an dieser Stelle keine besondere Rolle und wurden daher ausgelassen. Man kann an den zuvor diskutierten Aspekten erkennen, dass XP grundsätzlich viel weniger Vorschriften zur Prozessgestaltung macht als die analysierten wiederverwendungsorientierten Vorgehensmodelle (vgl. Abschnitt 4.4).

4.4.4 Agile Modellierung

Weniger codezentriert als bei XP, versucht Ambler (2002) die Prinzipien der Agile Alliance auf ein eher modellzentriertes Szenario zu übertragen. Wesentliches Ziel ist dabei, den „Modeling Sweet Spot“ (Ambler (2002)) zu finden, d.h. den Punkt, an dem das Modell noch so einfach wie möglich, aber bereits so ausführlich wie nötig ist (Einfachheitsprinzip). Agile Modeling (AM) erweitert die vier Grundwerte von XP (siehe Abschnitt 5.2.1) um den Aspekt „Bescheidenheit“ (Humility), was in diesem Zusammenhang bedeutet, dass kein einzelner AM-Entwickler sich anmaßen darf, alles zu wissen. Dies deckt sich teilweise auch mit dem von XP geforderten Respekt für die anderen Teammitglieder. Bei AM handelt es sich nicht um einen eigenständigen Prozess, sondern lediglich um agile Richtlinien, die die effiziente, kollaborative Modellierung unterstützen sollen. Ambler (2002) beschreibt daher exemplarisch, wie man AM wahlweise in XP oder RUP einbetten kann. Bei Ambler (2003) werden die AM-Prinzipien und -Techniken noch weiter in Richtung des MDD-Ansatzes ausgebaut (Agile Model-Driven Development, AMDD), mit dem Ziel automatischer Codegenerierung aus Modellen anstatt manueller Implementierung, d.h. Programmieren auf Modellebene (vgl. Abschnitt 4.3).

4.4.5 Fazit

Zusammenfassend lässt sich zu den agilen Prinzipien und Techniken sagen, dass sie eher auf die kollaborative Erstellung von Software bei kleinen bis mittleren Teamgrößen zugeschnitten sind, da sie versuchen, quasi eine Jeder-mit-Jedem-Kommunikation herzustellen, bis hin zu den Mitarbeitern des auftraggebenden Unternehmens. Per se sind agile Methodiken nicht hinreichend zur Unterstützung der Kollaboration in komplexen USW-Projekten mit mehreren verteilten Teams, möglicherweise aus unterschiedlichen Unternehmen, geeignet (z.B. innerhalb einer bestimmten Anwendungsdomäne), da sie geografische Kollokation erfordern. Zudem ignorieren Methodiken wie XP bewusst die Möglichkeiten der methodischen Wiederverwendung von Softwareartefakten und damit der industriellen „Produktion“ von USW. Bis auf erste modellgetriebene Gehversuche wie AM und AMDD sind die agilen Ansätze im Wesentlichen noch sehr codezentriert und auf die manuelle Implementierung fokussiert – lassen also auch Aspekte der Wiederverwendung durch Generierung außer Acht. Agile Methoden und Techniken sind daher für kollaborative Mikroprozesse innerhalb lokal arbeitender Teams geeignet, müssen aber für die Ziele dieser Arbeit von umfassenderen Makroprozessen zur team- und projektübergreifenden Kollaboration gekapselt und gesteuert werden.

4.5 Open Source Softwareentwicklung

Gemessen an der organisatorischen Ausdehnung und dem Grad der Aufgabeverteilung spiegeln die meisten Open Source-Projekte genau das Gegenteil typischer XP-Projekte wider – die kollaborativen Entwicklungsprozesse laufen zumeist domänenübergreifend oder -unabhängig ab, und die Entwickler sitzen nicht selten über den gesamten Erdball verteilt (vgl. Global Software Development, Herbsleb und Moitra (2001)).

4.5.1 Philosophie der quelloffenen Softwareerstellung

Die Ideale des Open Source Software Development (OSSD) und der damit verbundenen Bewegung basieren hauptsächlich auf den Grundsätzen der 1984 von Richard Stallman gegründeten Free Software Foundation (FSF) bzw. dem daraus resultierenden GNU-Projekt. Weitere Mitbegründer der Bewegung sind Linus Torvalds (Torvalds und Diamond (2002)) und Eric Raymond (Raymond (2001)), der Gründer der Open Source Initiative (OSI). Obwohl teilweise Uneinigkeit über die Verwendung der Begriffe free software und open software herrscht, und die FSF sich mehr an die Ideale von „freedom and collaboration“ bzw. „free as in freedom“ (Ravella und Lum (2004), S. 2) hält, besteht Einigkeit darüber, dass der Quelltext von Softwareprodukten frei zugänglich sein muss. Sowohl Goldberg (2002) als auch Szyperski und Spinellis (2004) vergleichen Softwareentwicklung daher mit der Literaturwissenschaft, bei der es ihrer Meinung nach ebenso unvorstellbar wäre, den eigenen „Quelltext“ anderen Autoren nicht frei zur Verfügung zu stellen – das bedeutet in diesem Fall nicht, dass dies ohne entsprechendes Entgelt geschehen muss.

4.5.2 Vergleich zu kommerziellen Projekten

Es existiert eine Reihe wesentlicher Unterschiede von Open Source-Projekten zu herkömmlichen kommerziellen Projekten im USW-Umfeld (Augustin u. a. (2002)). Diese bestehen im Wesentlichen in der großen Anzahl meist freiwilliger Entwickler sowie der freien Auswahl der Aufgaben durch die Community-Mitglieder („Ressourcenmobilität“, siehe Augustin u. a. (2002)). Innerhalb von Open Source Communities herrscht in der Regel eine selbstverständlichen Kultur des Teilens („Knowledge Sharing“) und der

Wiederverwendung von sowohl Expertise als auch Code (Augustin u. a. (2002)). Bei Open Source-Projekten sind Entwickler häufig auch gleichzeitig Nutzer (z.B. bei der Weiterentwicklung der Eclipse IDE, siehe auch Fogel und Bar (2002)). Des Weiteren existiert zumeist nur ein rudimentäres Systemdesign und minimale Projektplanung. Weitere zentrale Unterschiede sind die elektronischen Archivierung fast aller Informationsflüsse (z.B. aus Mailinglisten) und der ständigen Kritik und Anerkennung durch die anderen Mitglieder der Community („peer processes“ bzw. „peerage“, Augustin u. a. (2002), S. 561).

Dennoch, oder gerade deshalb, sind die Ergebnisse von Open Source-Projekten meist äquivalent und teilweise sogar denen der traditionellen Methodiken überlegen, wie z.B. im Falle des HTTP-Servers Apache, der mittlerweile über den größten Marktanteil bei den Web Servern verfügt (Mockus u. a. (2002)). Paulson u. a. (2004) konnten in einer empirischen Vergleichsstudie von Open und Closed Source Software-Projekten anhand von Softwaremetriken zeigen, dass Open Source-Projekte mehr Kreativität fördern und generell weniger Defekte aufweisen, da diese schneller gefunden und behoben werden. Als weitere weltweit erfolgreiche OSSD-Projekte, die ihrer kommerziellen Konkurrenz immer mehr Marktanteile abnehmen, lassen sich u.a. noch: das Betriebssystem Linux, der HTTP- und Mail-Client Mozilla und die integrierte, erweiterbare Entwicklungsplattform Eclipse nennen.

Die nächsten Schritte im Rahmen dieser Arbeit bestehen darin, die Quintessenz aus den im Open Source-Umfeld verwendeten Prinzipien, Methoden und Techniken zu extrahieren und möglicherweise auf die kollaborative Erstellung von USW zu übertragen: „[...] *the Open Source community seems to be a model for enterprise development practices of the future*“ (Augustin u. a. (2002), S. 562).

4.5.3 Der Open Source Softwareentwicklungsprozess

Anders als bei den Prozessmodellen aus Kapitel 4 und den agilen Methodiken, existiert keine einheitliche „Schablone“ zur Durchführung von Open Source-Projekten. Im Folgenden sollen einige wiederkehrende Prozessmuster, Methoden, Techniken und dazugehörige Werkzeuge vorgestellt werden, welche in bereits erfolgreichen Projekten zum Einsatz kommen (Best-Practices bzw. Best-of-Breed-Techniken).

Die kollaborativen Entwicklungsprozesse innerhalb der Open Source Communities waren schon seit dem Aufkommen der Bewegung Anfang der neunziger Jahre sehr werkzeugzentriert (Zeller und Krinke (2004)). Abbildung 5.2 zeigt die rudimentäre technische Infrastruktur eines typischen Open Source-Projektes. Das CVS-Repository, als zentrales Element, übernimmt hierbei eine Doppelfunktion für Konfigurationsmanagement (Koordinationsinstrument) und die öffentliche Bereitstellung der Quellen (daher auch Open Source, siehe Fogel und Bar (2002)). Als Kommunikationswerkzeuge kommen häufig E-Mail-Anwendungen bzw. Mailinglisten zum Einsatz. Auf dieser Basisplattform bauen fast alle der heute auch kommerziell eingesetzten CSD-Plattformen auf (vgl. Abschnitt 5.1.2).

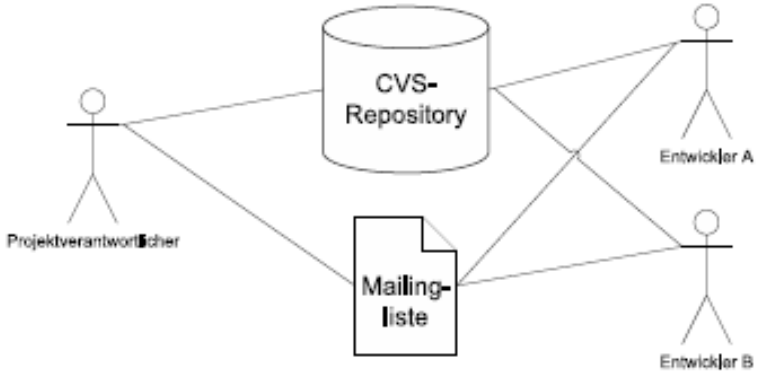

Abbildung 24: Stark vereinfachte Architektur eines typischen Open Source-Szenarios

Da in den Community-Prozess letztendlich nur fertige Codesegmente als Artefakte einfließen, bleiben die Aktivitäten im Rahmen der Analyse, des Designs und der Implementierung bis auf einige Rahmenvorgaben (z.B. Code Conventions und Refactorings, siehe Abschnitt 5.2.1) weitgehend den einzelnen Entwicklern überlassen (Methodenunabhängigkeit). Umso wichtiger sind daher aber ausgereifte Koordinations- und Entscheidungsprozesse, die die Integrierung von Beiträgen oder Änderungsvorschlägen regeln. Der Initiator eines Projekts fungiert aus diesem Grund dann auch häufig als so genannter „Betreuer“, der durch seine Entscheidungsgewalt und Autorität versucht, die Beiträge der einzelnen Entwickler zu koordinieren – d.h. Vorschläge anzunehmen oder abzulehnen (Fogel und Bar (2002)). Dabei muss der Betreuer sowohl die fachliche Relevanz der Beiträge als auch die technische Qualität des Quelltextes beurteilen können. Wird ein einzelner Betreuer im Projektverlauf mit wachsender Mitgliederzahl und Komplexität der Software durch die Koordinationsaufgaben überfordert, hat sich die Formierung eines Betreuerkomitees mit gleichen Stimmrechten und schreibendem Zugriff auf die Quelltexte im CVS-Repository (Commit) bewährt (Fogel und Bar (2002)).

Im Lauf der kurzen Geschichte der Open Source Communities haben sich drei wesentlichen Organisationsformen herausgebildet, welche sich auch auf die Community-Prozesse auswirken (Henkel (2003)). Abbildung 5.3 stellt diese unterschiedlichen Formen noch einmal vergleichend dar. In Abbildung 5.3a) ist die ursprüngliche Form einer Community dargestellt, wie sie beispielsweise anfangs bei der Entwicklung von Linux zu beobachten war. Der dargestellte Fall mit einem Betreuer (eingerahmter Punkt) ist ein Spezialfall der Organisationsform mit einem Betreuerkomitee. Abbildung 5.3b) beschreibt den Fall, dass ein oder auch mehrere Unternehmen Ressourcen, Quelltexte und/oder Man-Power in den Prozess einbringen, wie das z.B. Netscape und IBM bei Mozilla bzw. Eclipse getan haben (siehe Mockus u. a. (2002) bzw. Wolfe (2003)). Wie Henkel (2003) anhand der Domäne der eingebetteten Systeme bzw. embedded Linux untersucht hat, kann aber auch der dritte Fall einer „kommerziellen“ Community bzw. Professional Community (Abbildung 5.3c)), bestehend aus teilweise konkurrierenden Unternehmen, die zur Weiterentwicklung einer gemeinsamen Quelltextbasis beitragen und dabei in keinem festen Vertragsverhältnis stehen, wirtschaftlich Sinn machen („[...] collaborative software development by competing firms without a contractual base [...]“, Henkel (2003), S. 16). Henkel kam in seiner Studie zu dem Ergebnis, dass die beteiligten Unternehmen in diesem Fall die Diffusion von möglicherweise strategischem Wissen für nachweisbar kürzere Innovationszyklen und geringere Fehlerhäufigkeit in Kauf nehmen (siehe auch Paulson u. a. (2004)).

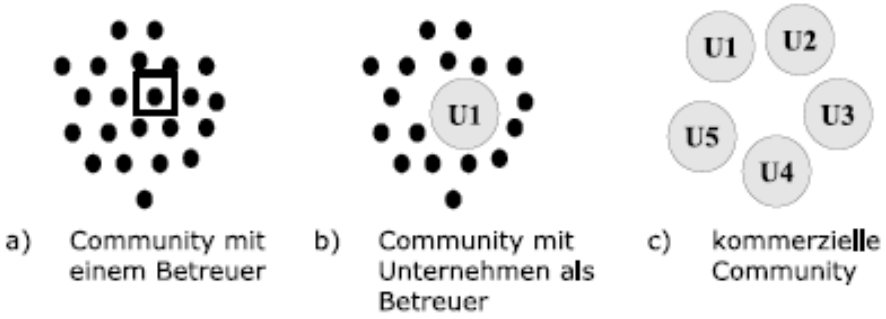

Abbildung 25: Unterschiedliche Formen von Open Source Communities (Henkel, 2003)

4.5.4 Fazit

In der von Henkel (2003) untersuchten Domäne waren die befragten Unternehmen der Meinung, dass die Freigabe von geistigem Eigentum und der Verlust von potentiellen strategischen Wettbewerbsvorteilen durch den Effizienzgewinn im Entwicklungsprozess mehr als kompensiert würden. Dies unterstreicht wiederum die anfangs aufgeführte These von Augustin u. a. (2002), dass Praktiken aus Open Source Communities als Modell für die zukünftige Erstellung von USW dienen könnten. Eine erste Entwicklung in diese Richtung zeigt sich durch die Anpassung und Vermarktung von ehemals im Open Source-Kontext eingesetzten Werkzeugen und Kollaborationsplattformen, wie dies beispielsweise bei SourceForge bzw. SourceForgeEnterprise und Eclipse bzw. WebSphere Studio der Fall war (siehe Kiss (2003) bzw. Lentzsch (2002)). Szyperski und Spinellis (2004) sprechen in diesem Zusammenhang sogar von einem möglichen „paradigm shift in the way we develop software“ (S. 33) – Clemens Szyperski ist Software Architekt bei Microsoft, dem bisweilen größten Verfechter von proprietärer, nicht quelloffener Software (siehe auch Messerschmitt und Szyperski (2003)).

5 Fazit

Dieses letzte Kapitel soll noch einmal die wesentlichen Ergebnisse der hier vorgestellten analytischen Arbeit zusammenfassen, gegenwärtige methodische Defizite im Hinblick auf die kollaborative Softwareerstellung aufzeigen, Implikationen für entsprechende Unterstützungswerkzeuge ableiten und einen Ausblick auf künftige Forschungsfragen geben.

5.1 Zusammenfassung

Abbildung 26 zeigt noch einmal die Zusammenhänge und Einflüsse zwischen den einzelnen Modellen auf[7]. Dem V-Modell kann anhand seiner eher nationalen Verbreitung in Deutschland kein direkter Einfluss auf nachfolgende, international angewandte Methodiken nachgewiesen werden, obwohl einige Nachfolger ähnliche Techniken, beispielsweise zur Verifikation und Validierung, übernommen haben. Aus dem Schaubild und der vorangegangenen Analyse wird deutlich, dass KobrA sowie der RUP’04 und das VMXT momentan die modernsten Ansätze im Bereich der wiederverwendungsorientierten Vorgehensmodelle darstellen und durch sie viele Techniken früherer Modelle subsumiert und erweitert werden, wobei KobrA dem Vergleich zu RUP und VMXT be-

[7] das VMXT wird in dieser Darstellung noch als „V-Modell 200x“ bezeichnet, da es sich dabei um den ursprünglichen Arbeitstitel vor der offiziellen Publikation handelte

züglich der Prozessabdeckung und vor allem der entsprechenden Werkzeugunterstützung nicht standhalten kann (vgl. Atkinson u. a. (2002)).

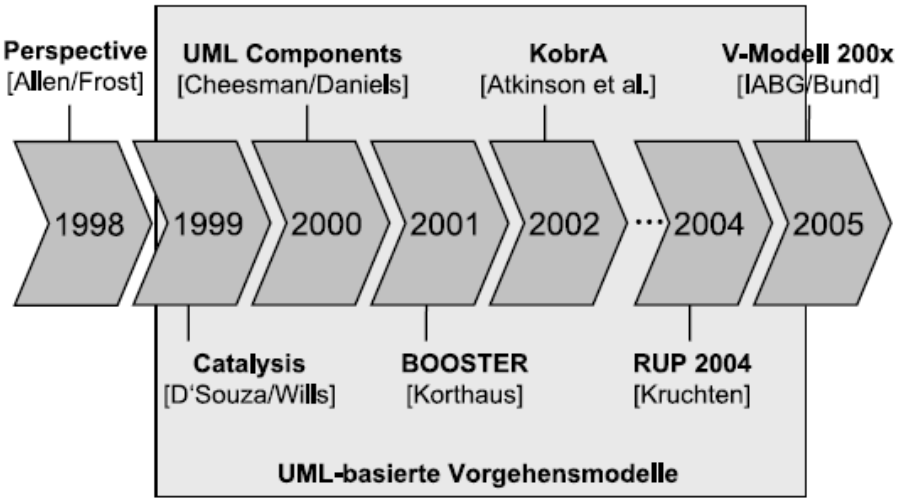

Abbildung 26: Chronologische Entwicklung ausgewählter Vorgehensmodelle

Die umfassende Analyse der existierenden Methodiken und Ansätze ergab jedoch als wesentliches Ergebnis, dass die Kollaboration unterschiedlicher Stakeholder aus Entwickler- und Anwenderunternehmen im Rahmen des Softwareerstellungsprozesses nur rudimentär unterstützt wird. Insbesondere finden neuere Konzepte wie beispielsweise die kollaborative, verteilte Anforderungserhebung mit der EasyWinWin-Methode oder Entwicklungsmethodiken aus dem Open Source-Bereich bisher im Rahmen von Prozessrahmenwerken und Vorgehensmodelle keinerlei Berücksichtigung.

5.2 Defizite der existierenden methodischen Ansätze

"Tools are integral to the process" (Jacobson u. a. (1999), S. 28). Vor allem bei der Unterstützung der Durchführung und der Koordination der teilweise sehr komplexen Prozesse durch geeignete Werkzeuge lassen sich bei vielen Modellen Schwachpunkte ausmachen (vgl. Jacobson u. a. (1999)). Eine Ausnahme bilden dabei der UP von Rational und Perspective von Select Business Solutions, da diese beiden Modelle jeweils in Verbindung mit den entsprechenden, kommerziellen Werkzeugen ausgeliefert und parallel weiterentwickelt werden. Auffällig wird dieses Defizit bei komplexen Modellen, wie z.B. Booster und vor allem KobrA, bei denen das zusätzliche Potenzial zur Produktivitätssteigerung durch modellgetriebene Wiederverwendung wegen des Mangels an passenden MDD-Werkzeugen und -Standards noch nicht ausgeschöpft werden kann (siehe Abschnitt 4.3). Ein weiteres Problem der betrachteten Methodiken sind die teilweise sehr unklar beschriebenen Methoden und Techniken zur Komponentenidentifikation. Die rekursive Zerlegung des Systems in eine baumartige Komponentenstruktur, wie von Atkinson u. a. (2002) dargestellt, erscheint dabei für das dieser Arbeit zugrunde liegende Komponentenverständnis am besten geeignet. Allerdings ist bei KobrA das „Stoppkriterium" für die rekursive Zerlegung, d.h. die Definition von atomaren Komponenten, auch nicht eindeutig geklärt. Zudem werden parallele bzw. nebenläufige Entwicklungsaktivitäten (vgl. Abschnitt 3.4.3), z.B. bei Komponentenentwurf und -implementierung, bisher in keinem der Modelle systematisiert (vgl. Abschnitt 3.4.3), obwohl es sich dabei um ein Prozessmuster handelt, welches in der Praxis schon häufiger beobachtet werden konnten (siehe Cain und Coplien (1996)). Da es sich bei den untersuchten Prozessmodellen, außer bei UMLC, um sehr „schwergewichtige" und umfangreiche Prozessbeschreibungen handelt, wird man bei der Umsetzung solcher

Prozesse in bestehenden Softwareorganisationen immer auf soziotechnische Probleme, wie beispielsweise mangelnde Akzeptanz, treffen. Ein möglicher Ansatzpunkt für diese Problematik wären spezielle, prozessunterstützende Werkzeuge und entsprechende Mitarbeiterschulungen (vgl. Balamuralikrishna u. a. (2000)). Ebenfalls ein teilweise soziotechnisches Problem ist die Schaffung einer Wiederverwendungskultur im Unternehmen (Korthaus (2001)) und die Etablierung organisationsübergreifender Wiederverwendungsprozesse, wie beispielsweise der kollaborativen Erstellung von USW für einzelne Anwendungsdomänen. Diese projekt- bzw. unternehmensübergreifende Zusammenarbeit bei der Softwareentwicklung wurde in den bisherigen Modellen noch kaum explizit behandelt und schon gar nicht systematisch unterstützt (siehe nächstes Kapitel). OPEN, Perspective, Catalysis und Booster bieten erste Ansätze zur domänenorientierten Archivierung und Wiederverwendung von Artefakten, beschreiben die hierfür benötigen Prozesse und Technologien aber nur rudimentär.

5.3 Implikationen für die Werkzeugunterstützung

Aus den methodischen Defiziten der existierenden Entwicklungsmethodiken ergeben sich auch Implikationen für die entsprechende Werkzeugunterstützung. Die Beteiligung mehrerer, fachlich sehr unterschiedlicher Stakeholder führt zu speziellen Anforderungen an die jeweilige Werkzeugunterstützung der relevanten Aktivitäten im Prozess. Es müssen daher sowohl die einzelnen Disziplinen und/oder Aktivitäten mit entsprechenden Werkzeugen individuell unterstützt werden als auch eine zentrale „Kollaborationsplattform“ zur Projektkoordination bereit stehen. Im Open Source-Bereich wird diese Funktion im Wesentlichen von Projektplattformen wie SourceForge bzw. den entsprechenden Komponenten für Versionskontrolle (Subversion, CVS etc.) und Gruppenkommunikation (Mailinglisten, Issue Tracker, Foren etc.) erfüllt. Auch Hersteller von kommerziellen Entwicklungswerkzeugen, wie beispielsweise IBM Rational und Borland, bieten immer mehr Teamfunktionalitäten an und unterstützen damit vermehrt kollaborative Softwareerstellungsprozesse. Die Frage, welche Funktionen für welche Aktivitäten geeignet bzw. essenziell sind ist bisher hingegen noch weitestgehend ungeklärt.

5.4 Ausblick

Die Analyse existierender Vorgehensmodelle und erster Ansätze zur kollaborativen Softwareerstellung hat zwei wesentliche Ergebnisse hervorgebracht. Zum einen findet die kollaborativen, verteilten und auch zwischenbetrieblichen Aspekte im Prinzip keinerlei Beachtung in den existierenden Vorgehensmodellen. Sogar neuere Modelle wie der RUP 2004 und das VMXT gehen auf diese Art von Prozessrahmenbedingungen nicht ein. Die weiteren methodischen Ansätze in Kapitel 4 hingegen sind teilweise speziell für kollaborative (z.B. XP und EasyWinWin) und auch verteilte Szenarien (bspw. OSSD) konzipiert. Allerdings sind diese wiederum nur eingeschränkt in und zwischen Unternehmen einsetzbar. Spezialisierte Methoden für Anforderungsanalyse, Modellierung und auch Implementierung, Test und Wartung sowie agile Methoden bieten nicht den nötigen Prozessumfang für große Projekte in Unternehmen und müssen somit durch eine übergeordnete Methodik oder Prozessframework ergänzt werden.

Daraus ergeben sich zahlreiche offene Fragestellungen und zukünftige Forschungsfelder: Beispielsweise ist die Frage, welches die richtige Methodik für ein verteiltes, zwischenbetriebliches Szenario, bisher nicht hinreichend erforscht. Agile Methoden und OSSD bieten dazu erste Ansätze, die jedoch wiederum nicht ohne weiteres auf große kommerzielle Projekte übertragbar sind. Will man etablierte Methodiken und Frameworks bezüglich deren Unterstützung kollaborativer Prozesse erweitern, ergeben sich

ebenfalls zahlreiche weitere Forschungsfrage. Um letztendlich zu einer „optimalen“ Lösung zu kommen, können beide Wege zum Ziel führen und sich ergänzen. Wie bereits in Abschnitt 5.3. erwähnt wurde, ergeben sich aus den methodischen Aspekten bestimmte Anforderungen an deren Unterstützung durch Entwicklungswerkzeuge. Diese Thematik wird in einem gesonderten Arbeitspapier wieder aufgegriffen werden (siehe Geisser et al. (2006)). Da in geografisch verteilten Szenarien, also auch bei der zwischenbetrieblichen Softwareerstellung, die Nutzung Internet-basierter Werkzeuge eine große Rolle spielt, kommt deren Gestaltung und Weiterentwicklung parallel zur Entwicklung der Methodik eine äußerst hohe Bedeutung zu (vgl. Jacobson et al. 1999).

Literaturverzeichnis

[Abran und Moore 2001] Abran, A. (Hrsg.) ; Moore, J.D. (Hrsg.): Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, May 2001

[Agile Alliance 2001] Agile Alliance: Agiles Manifest. 2001. – URL http://www. agile-manifesto.org (14.04.2004)

[Allen und Frost 1998] Allen, P.R.; Frost, S.: Component-Based Development for Enterprise Systems : Applying the Select Perspective. Cambridge University Press, 1998

[Altmann 19999] Altmann, J.: Kooperative Softwareentwicklung – Rechnerunterstützte Koordination und Kooperation in Softwareprojekten. Dissertation, Universitätsverlag Rudolf Trauner, Linz,1999.

[Altmann und Weinrich 1998] Altmann, J.; Weinreich, Rainer: An Environment for Cooperative Software Development Realization and Implications. In: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences, Band 1, 1998, S. 27-37.

[Ambler 2002] Ambler, S.W.: Agile Modeling – Effective Practices for Extreme Programming and the Unified Process. Wiley Computer Publishing, 2002 (Programming and Software Development)

[Ambler 2003] Ambler, S.W.: Agile Model Driven Development (AMDD). 2003. – URL http://www.agilemodeling.com/essays/amdd.htm(14.04.2004)

[AS4651-2004] Australian Standard: Standard Metamodel for Software Development Methodologies. Standards Australia International, 2004.

[Apperly et al. 2003] Apperly, H. ; Hofman, R. ; Latchem, S. ; Maybank, B. ; McGibbon, B. ; Piper, D. ; Simons, C.: Service- and Component-based Development: Using the Select Perspective and UML. Addison-Wesley, 2003 (Component Software Series)

[Atkinson et al. 2002] Atkinson, C. ; Bayer, J. ; Bunse, C. ; Kamsties, E. ; Laitenberger, O. ; Laqua, R. ; Muthig, D. ; Paech, B. ; Wüst, J. ; Zettel, J.: Component Based Product Line Engineering with UML. Addison-Wesley, 2002 (Component Software Series)

[Atkinson und Groß 2003] Atkinson, C. ; Groß, H.-G.: Model Driven Component-Based Development. In: Barbier, F. (Hrsg.): Business Component-Based Software Engineering. Kluwer International, 2003

[Atkinson und Muthig 2002] Atkinson, C. ; Muthig, D.: Model-Driven Product Line Architectures. In: Proceedings of the 2nd International Software Product Line Conference, 2002

[Augustin et al. 2002] Augustin, L. ; Bressler, D. ; Smith, G.: Accelerating Software Development Through Collaboration. In: Proceedings of the 24th International Conference on Software Engineering (ICSE-02), ACM Press, 2002, S. 559–566

[Avison und Fitzgerald 1995] Avison, D.E. ; Fitzgerald, G.: Informations Systems Development: Methodologies, Techniques and Tools. 2. Auflage. Blackwell Scientific Publications, 1995 (Information Systems Series)

[Avison und Fitzgerald 2003] Avison, D. & Fitzgerald, G.: Information Systems Development: Methodologies, Techniques and Tools. McGraw-Hill, 2003

[Babar et al. 2004] Babar, M.A.; Kitchenham, B.; Zhu, L.; Jeffery, R.: An Exploratory Study of Groupware Sup-port for Distributed Software Architecture Evaluation Process.

In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), Busan, Korea, 2004.

[Babar und Gorton 2004] Babar, M.A.; Gorton, Ian: Comparison of Scenario-Based Software Architecture Evaluation Methods. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), Busan, Korea, 2004, S. 600-607.

[Balamuralikrishna u. a. 2000] Balamuralikrishna, R. ; Anthinarayanan, R. ; Song, X.: The Relevance of Concurrent Engineering in Industrial Technology Programs. In: Journal of Industrial Technology 16 (2000), Nr. 3

[Ballarini et al. 2003] Ballarini, Daniele; Cadoli, Marco; Gaetta, Matteo; Mancini, Toni; Mecella, Massimo; Ritro-vato, Pierluigi; Santucci, Giuseppe: Modeling Real Requirements for Cooperative Software Development: A Case Study. Working Paper, 2003.

[Balzert 1998] Balzert, H.: Lehrbuch der Software-Technik – Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung. Bd. 2. Spektrum, 1998

[Balzert 2000] Balzert, H.: Lehrbuch der Software-Technik – Software-Entwicklung. Bd. 1. 2. Auflage. Spektrum, 2000

[Beck 2000] Beck, K.: Extreme programming : die revolutionäre Methode für Softwareentwicklung in kleinen Teams. Addison-Wesley, 2000

[Beck und Fowler 2001] Beck, K. ; Fowler, M.: Extreme programming planen. Addison-Wesley, 2001

[Bennington 1956] Bennington, H.D.: Production of Large Computer Programs. In: Proceedings of the ONR Symposium on Advanced Programming Methods for Digital Computers, Juni 1956, S. 15–27

[Boehm 1981] Boehm, B.W.: Software Engineering Economics. Prentice Hall, 1981

[Boehm 1986] Boehm, B.W.: A Spiral Model of Software Development and Enhancement. In: ACM SIGSOFT (1986), August, S. 14–24

[Boehm 1988] Boehm, B.W.: A Spiral Model of Software Development and Enhancement. In: IEEE Computer (1988), May, S. 61–72

[Boehm et al. 1995] Boehm, B.W. ; Bose, P. ; Horowitz, E. ; Lee, M.-J.: Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach. In: Proceedings of the 17th International Conference on Software Engineering, IEEE, April 1995, S. 243–254

[Boehm et al. 2001] Boehm, B.W.; Gruenbacher, P.; Briggs, R.O.: EasyWinWin: A Groupware-Supported Methodology For Requirements Negotiation. In: 23rd International Conference on Software Engineering, IEEE Computer Society, 2001.

[Boehm und Bose 1994] Boehm, B.W.; Bose, P.: A Collaborative Spiral Software Process Model Based on Theory W. In: Proceedings of the 3rd International Conference on the Software Process, Applying the Software Process, 1994.

[Boehm und Gruenbacher 2001] Boehm, Barry W.; Gruenbacher, P.; Briggs, R.O.: Developing Groupware for Re-quirements Negotiations: Lessons Learned. In: IEEE Software, 18 (2001) 3, S. 46-55.

[Boehm und Ross 1989] Boehm, B.W.; Ross, R.: Theory W Software Project Management: Principles and Examples. In: IEEE Transaction of Software Engineering, 1989, S. 902-916.

[Briggs und Gruenbacher 2002] Briggs R.O. ; Gruenbacher, P.: EasyWinWin: Managing Complexity in Requirements Negotiation with GSS, In: Proceedings of the 35th Hawaii International Conference on System Sciences, 2002.

[Booch et al. 1999] Booch, G. ; Rumbaugh, J. ; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999

[Broy und Rausch 2005] Broy, M. ; Rausch, A. : Das neue V-Modell Xt – Ein anpassbares Modell für Software und System Engineering. In: Informatik Spektrum, 28 (2005) 3, S. 220-229

[Budde et al. 1992] Budde, R. ; Kantz, K. ; Kuhlenkamp, K. ; Z¨ullighoven, H.: Prototyping – An Approach to Evolutionary System Development. Springer Verlag, 1992

[Cain und Coplien 1996] Cain, B.G. ; Coplien, J.O.: Social Patterns in Productive Software Development Organizations. In: Annals of Software Engineering (1996), Dezember

[CanyonBlue 2005] CanyonBlue: Collaborative UML Development. http://www.canyonblue.com/CanyonBlue.pdf, abgerufen am 10.10.2005.

[Carmel et al. 1993] Carmel, E.; Whitaker, R.D.; George, J.F.: PD and Joint Application Design: A Transatlantic Comparison. In: Communcations of the ACM, 36 (1993) 6, S. 40-48.

[Checkland 1981] Checkland, P.B.: Systems Thinking, Systems Practice. John Wiley and Sons Ltd., 1981

[Cheesman und Daniels 2001] Cheesman, J. ; Daniels, J.: UML Components : A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2001 (Component Software Series)

[Chrissis et al. 2003] Chrissis, M.; Konrad, M. & Shrum, S.: Guidelines for Process Integration and Product Improvement Addison-Wesley, 2003

[Coad und Nicola 1993] Coad, P. ; Nicola, J.: Object Oriented Programming. 2. Auflage. Prentice Hall, 1993. – Baseball Model

[Cockburn 2001] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, 2001 (The Crystal Collection for Software Development)

[Cockburn 2003] Cockburn, A.: Agile Software-Entwicklung. Verlag Moderne Industrie, 2003

[Damian 2001] Damian, Daniela: An empirical study of requirements engineering in distributed software projects: is distance negotiation more effective? In: Proceedings of the Asia Pacific Software Engineering Conference (APSEC) (2001)

[Damian und Eberlein 2000] Damian, Daniela; Eberlein, Armin; Shaw, Mildred; Gaines, Brian: Studies in Distributed Software Requirements Engineering. In: Proceedings of the 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW'99).

[Damian et al. 2003] Damian, D. E.; Eberlein, A.; Shaw, M.L.G.; Gaines, B.R.: An Exploratory Study of Facilitation in Distributed Requirements Engineering. In: Requirements Engineering Journal: Special Issue on Selected Papers from RE'01, 8 (2003) 1, S. 23-41.

[Dean et al. 1998] Dean, D.L.; Lee, J.D.; Pendergast, M.O.; Hickey, A.M.; Nunamaker, J.F.: Enabling the Effective Involvement of Multiple Users: Methods and Tools for Collaborative Software Development. In: Journal of Mangement Information Systems, 14 (1998).

[Dewan und Riedel 1993] Dewan, P.; Riedl, J.: Toward Computer-Supported Concurrent Software Engineering. In: IEEE Computer, 26 (1993) 1, S. 17-27.

[Dröschel und Wiemers 2000] Dröschel, W. (Hrsg.) ; Wiemers, M. (Hrsg.): Das V-Modell 97: Der Standard f¨ur die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Oldenbourg Verlag, 2000

[Dröschel et al. 1998] Dröschel, W. (Hrsg.) ; Heuser, W. (Hrsg.) ; Midderhoff, R. (Hrsg.): Inkrementelle und objektorientierte Vorgehensweise mit dem V-Modell 97. Oldenbourg Verlag, 1998

[D'Souza und Wills 1999] D'Souza, D.F. ; Wills, A.C.: Objects, Components, and Frameworks with UML : The Catalysis Approach. Addison-Wesley, 1999 (Object Technology)

[Dymond 2002] Dymond, K.M.: CMM Handbuch: Das Capability Maturity Model f¨ur

Software. Springer Verlag, 2002 (Xpert.press)

[ECMA 1994] ECMA: Mapping of the PCTE to the ECMA/NIST Frameworks Reference Model / European Computer Manufacturers Association. 1994 (TR/66). – Forschungsbericht

[ECMA 1997] ECMA: Portable Common Tool Environment (PCTE) / European Computer Manufacturers Association. 1997 (ECMA-149). – Abstract Specification

[Eriksson et al. 2004] Eriksson, H.-E. ; Penker, M. ; Lyons, B. ; Fado, D.: UML 2 Toolkit. 2. Auflage. Wiley, 2004

[Fettke und Loos 2002] Fettke, P. ; Loos, P.: Komponentenorientierte Vorgehensmodelle im Vergleich. In: Turowski, K. (Hrsg.): 4. Workshop komponentenorientierte betriebliche Anwendungssysteme, 2002

[Firesmith und Henderson-Sellers 2002] Firesmith, D.G.; Henderson-Sellers, B.: The OPEN Process Framework: An Introduction. Addison-Wesley Professional, 2002

[Fitzgerald 1996] Fitzgerald, G.: Formalized Systems Development Methodologies: A Critical Perspective. In: Information Systems Journal 6 (1996), Nr. 1, S. 3–23

[Fitzgerald 1998a] Fitzgerald, G.: An Empirical Investigation into the Adoption of Systems Development Methodologies. In: Information & Management 34 (1998), S. 317–328

[Fitzgerald 1998b] Fitzgerald, G.: An Empirically-Grounded Framework for the IS Development Process. In: Hirschheim, R. (Hrsg.) ; Newman, M. (Hrsg.) ; deGross, J. (Hrsg.): Proceedings of the 19th International Conference in Information Systems, 1998, S. 103–114

[Floyd 1984] Floyd, C.: A Systematic Look at Prototyping. In: Budde, R. (Hrsg.) ; Kuhlenkamp, K. (Hrsg.) ; Mathiessen, L. (Hrsg.) ; Z¨ullighoven, H. (Hrsg.): Approaches to Prototyping. Springer Verlag, 1984

[Fogel und Bar 2002] Fogel, K. ; Bar, M.: Open Source-Projekte mit CVS. 2. Auflage. MITP Verlag, 2002

[Forsell et al. 1999] Forsell, M. ; Halttunen, V. ; Ahonen, J.: Evaluation of Component-Based Software Development Methodologies. In: Proceedings of the Fenno-Ugric Symposium of Software Technology (FUSST'99), Tallin Technical University, 1999, S. 53–63

[Fowler und Beck 1999] Fowler, M. ; Beck, K.: Refactoring : improving the design of existing code. Addison-Wesley, 1999

[Fowler und Highsmith 2001] Fowler, M. ; Highsmith, J.: The Agile Manifesto. In:Software Development Magazine (2001), August

[Gamma et al. 2002] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, 2002 (Professional Computing)

[Geisser und Hildenbrand 2005] Geisser, Michael; Hildenbrand, Tobias: Eine Methode zur kollaborativen Anforderungserhebung und entscheidungsunterstützenden Anforderungsanalyse. Arbeitspapiere in der Wirtschaftsinformatik, Universität Mannheim, 2005.

[Geisser et al. 2006] Geisser, Michael; Hildenbrand, Tobias; Klimpke, Lars; Rashid, Asarnusch: Werkzeuge zur kollaborativen Softwareerstellung – Stand der Technik. Arbeitspapiere in der Wirtschaftsinformatik, Universität Mannheim, 2006.

[Gentleware 2005] Gentleware: Poseidon for UML (Produktbeschreibung). http://www.gentleware.de/index.php?id=30, Abruf am 2005-04-28.

[Graham et al. 1997] Graham, I. ; Henderson-Sellers, B. ; Younessi, H.: The OPEN Process Specification. ACM Press, 1997 (The OPEN Series)

[Graham et al. 1999] Graham, T.; Stewart, H.; Kopaee, A.; Ryman, A.; Rasouli, R.: A World-Wide-Web Architecture for Collaborative Software Design. In: Proceedings of the Conference on Software Tech-nology and Engineering Practice, Pittsburgh, Pennsylvania, USA, 1999, S. 22-29.

[Gruenbacher 2000] Gruenbacher, Paul: Collaborative Requirements Negotiation with EasyWinWin. In: Proceedings of the 11th International Workshop on Database and Expert Systems Applications , IEEE, 2000.

[Gruenbacher und Braunsberger 2003] Gruenbacher, P.; Braunsberger, P.: Tool Support For Distributed Requirements Negotiation. In: Cooperative Methods and Tools for Distributed Software Processes, Franco Angeli, Milano, Italy, 2003.

[Grundmann 2005] Grundmann, Markus: Prozessmodellintegration am Beispiel einer RUP-Erweiterung durch das V-Modell XT. Diplomarbeit an der Hochschule Reutlingen, Fachbereich Informatik, Januar 2005.

[Hammer und Champy 2001] Hammer, M. ; Champy, J.: Reengineering the Corporation – A Manifesto for Business Revolution. 2. Auflage. HarperCollins, 2001 (Harper Business)

[Heinrich und Häntschel 2000] Heinrich, Lutz J.; Häntschel, I.: Evaluation und Evaluationsforschung in der Wirtschaftsinformatik: Handbuch für Praxis, Lehre und Forschung. Oldenbourg Verlag, 2000.

[Henderson-Sellers 2003] Henderson-Sellers, B.: Method Engineering for OO Systems Development. In: Communications of the ACM 46 (2003), Nr. 10

[Henderson-Sellers und Edwards 1990] Henderson-Sellers, B. ; Edwards, J.M.:The Object-Oriented Systems Life Cycle. In: Communications of the ACM 33 (1990), Nr. 9

[Henderson-Sellers und Unhelkar 2000] Henderson-Sellers, B. ; Unhelkar, B.: OPEN Modeling with UML. Addison-Wesley, 2000

[Henkel 2003] Henkel, J.: Software Development in Embedded Linux – Informal Collaboration of Competing Firms. In: Tagungsband der Internationalen Tagung Wirtschaftsinformatik, 2003

[Herbsleb und Moitra 2001] Herbsleb, J.D. ; Moitra, D.: Global Software Development. In: IEEE Software 18 (2001), Nr. 2, S. 16–20

[Herzum und Sims 1999] Herzum, P. ; Sims, O.: Business Component Factory – A Comprehensive Overview of Component-Based Development for the Enterprise. Wiley, 1999

[Hildenbrand und Korthaus 2004] Hildenbrand, T. ; Korthaus, A.: A Model-Driven Approach to Business Software Engineering. 2004

[Hirschheim und Klein 1989] Hirschheim, R. ; Klein, H.K.: Four Paradigms of Information Systems Development. In: Communications of the ACM 32 (1989), Nr. 10, S. 1199–1216

[Hirschheim et al. 1995] Hirschheim, R. ; Klein, H.K. ; Lyytinen, K.: InformationSystems Development and Data Modeling: Conceptual and Philosophical Foundations. Cambridge University Press, 1995

[Hruschka 2003] Hruschka, P.: Agility – (Rück-)Besinnung auf Grundwerte in der Softwareentwicklung. In: Informatik Spektrum 6 (2003), Dezember, S. 397–401

[Iivari et al. 2001] Iivari, J. ; Hirschheim, R. ; Klein, H.K.: A Dynamic Framework for Classifying Information Systems Development Methodologies and Approaches. In: Journal of Management Information Systems 17 (2001), Nr. 3, S. 179–218

[IBM 2005] IBM: Rational Software Development Tools. http://www.ibm.com/software/rational/, Abruf am 2005-04-28.

[IBM 2005a] IBM Customer Success: Florida Department of Health Relies on RUP and IBM Rational Tools for Long-Term Projects and Rapid Crisis Response. http://www3.software.ibm.com/ibmdl/pub /software/rational/web/success/fl_dept_health.pdf, Abruf am 2005-04-28.

[Industrieanlagen-Betriebsgesellschaft 1997] Industrieanlagen-Betriebsgesellschaft: Allgemeiner Umdruck Nr. 250: Vorgehensmodell: Planung und Durchführung von IT-Vorhaben, Entwicklungsstandard für IT-Systeme des Bundes. 1997. – URL http://www.v-modell.iabg.de/(17.05.2004)

[Jacobson et al. 1997] Jacobson, I. ; Griss, M. ; Jonsson, P.: Software Reuse – Architecture, Process and Organization for Business Success. Addison-Wesley, 1997 (ACM Press)

[Jacobson et al. 1999] Jacobson, I. ; Booch, G. ; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999

[Jayaratna 1994] Jayaratna, M.: Understanding and Evaluating Methodologies: NIMSAD, a Systematic Framework. McGraw-Hill, 1994 (Information Systems, Management and Strategy Series)

[Kieback et al. 1992] Kieback, A. ; Lichter, H. ; Schneider-Hufschmidt, M. ; Züllighoven, H.: Prototyping in industriellen Software-Produkten. In: Informatik Spektrum 15 (1992), S. 65–77

[Kiss 2003] Kiss, F.: Tools für Teams. In: Javamagazin (2003), Nr. 10

[Korthaus 1997] Korthaus, A.: Business Objects as Constituents of Future Distributed Object-Oriented Business Information Systems / Universität Mannheim. 1997. – Forschungsbericht

[Korthaus 1998] Korthaus, A.: Using UML for Business Object-Based Systems Modeling. In: The Unified Modeling Language – Technical Aspects and Applications. Physica Verlag, 1998

[Korthaus 2001] Korthaus, A. ; Gaul, W. (Hrsg.) ; Schader, M. (Hrsg.): Informationstechnologie und Ökonomie. Bd. 20: Komponentenbasierte Entwicklung computergestützter betrieblicher Informationssysteme. Peter Lang, 2001

[Kruchten 2000] Kruchten, P.B.: The Rational Unified Process : An Introduction. 2. Auflage. Addison-Wesley, 2000 (Object Technology Series)

[Kruchten 2004] Kruchten, P.B.: The Rational Unified Process : An Introduction. 3. Auflage. Addison-Wesley, 2004 (Object Technology Series)

[Kühne 2003] Kühne, T.: Automatisierte Softwareentwicklung mit Modellcompilern. In: Thema Forschung 1 (2003)

[Kumar und Welke 1992] Kumar, K. ; Welke, R.J.: Methodology Engineering: A Proposal for Situation Specific Methodology Construction. In: Cotterman, W.W. (Hrsg.) ; Senn, J.A. (Hrsg.): Challenges and Strategies for Research in Systems Development. John Wiley & Sons, 1992, S. 257–269

[Leavitt 1972] Leavitt, H.: The Volatile Organization: Everthing Triggers Everything Else. University of Chicago Press, 1972 (Managerial Psychology)

[Lentzsch 2002] Lentzsch, K.: Licht ins Dunkel. In: Javamagazin (2002), Nr. 2

[Lloyd et al. 2002] Lloyd, Wesley James; Rosson, Mary Beth; Arthur, James D.: Effectiveness of Elicitation Tech-niques in Distributed Requirements Engineering. In: Proceedings of the IEEE Joint International Conference on Requirements Engineering , 2002.

[Lougher und Rodden 1993] Lougher, Robert; Rodden, Tom: Supporting Long-term Collaboration in Software Maintenance. In: Proceedings of the Conference on Organizational Computing Systems (COCS '93), New York USA, 1993.

[Mashayekhi et al. 1993] Mashayekhi, Vahid; Drake, Janet M.; Tsai, Wei-Tek; Riedl, John: Distributed, Collaborative Software Inspection. In: IEEE Software, 10 (1993) 5, S. 66-75.

[Mashayekhi et al. 1994] Mashayekhi, Vahid; Feulner, Chris and Riedl. John: CAIS: Collaborative Asynchronous Inspection of Software. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94), S. 21-34.

[Messerschmitt und Szyperski 2003] Messerschmitt, D.G. ; Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press, September 2003

[Mockus et al. 2002] Mockus, A. ; Fielding, R. ; Herbsleb, J.D.: Two Case Studies of Open Source Software Development: Apache and Mozilla. In: ACM Transactions on Software Engineering and Methodology 11 (2002)

[Müller-Ettrich 1999] Müller-Ettrich, G.: Objektorientierte Prozessmodelle: UML einsetzen mit OOTC, V-Modell, Objectory. Addison-Wesley, 1999 (Professionelle Softwareentwicklung)

[Noack und Schienmann 1999] Noack, J. ; Schienmann, B.: Objektorientierte Vorgehensmodelle im Vergleich. In: Informatik-Spektrum 22 (1999)

[Oberweis 1994] Oberweis, A.: Workflow Management in Software Engineering Projects. In: Proceedings of the 2nd International Conference on Concurrent Engineering and Electronic Design Automation, Bournemouth, UK, 1994.

[Oberweis et al. 1994] Oberweis, A.; W., Thomas; Stucky, W.: Teamwork Coordination in a Distributed Software Development Environment. In: Proceedings of the IFIP'94 Workshop FG 9: Communication and Coordination in Distributed Corporate Application Systems, Hamburg, Germany, 1994.

[Object Management Group 2002b] Object Management Group: Software Process Engineering Metamodel Specification, Version 1.0. Specification formal/02-11-14. November 2002

[Object Management Group 2002c] Object Management Group: UML Profile for Enterprise Distributed Object Computing (EDOC). Adopted Specification ptc/02-02-05. Februar 2002

[Object Management Group 2003e] Object Management Group: Unified Modeling Language (UML) Specification, Version 1.5. Specification formal/03-03-01. März 2003

[Oestereich 2001a] Oestereich, B. (Hrsg.): Erfolgreich mit Objektorientierung : Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung. 2. Auflage. Oldenbourg, 2001

[Oestereich 2001b] Oestereich, B.: Objektorientierte Softwareentwicklung : Analyse und Design mit der Unified Modeling Language. 5. Auflage. Oldenbourg, 2001

[Oestereich 2003] Oestereich, B.: Objektorientierte Geschäftsprozessmodellierung mit der UML. dpunkt, 2003

[Paulson et al. 2004] Paulson, J.W. ; Succi, G. ; Eberlein, A.: An Empirical Study of Open-Source and Closed-Source Software Products. In: IEEE Transactions on Software Engineering 30 (2004), Nr. 4, S. 246–256

[Piwecki und Schmidt 2004] Piwecki, M. ; Schmidt, R.: Aufräumarbeiten - Systemwartung mit IBM Rationals Clearquest. In: iX Magazin für professionelle Informationstechnik (2004), Juni, Nr. 6

[Project Management Institute 2000] Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK Guide). American National Standard ANSI/PMI 99-001-2000.

[Rausch et al. 2004] Rausch, A. ; Hammerschall, U. ; Vogel, S.: Das V-Modell 200x – ein modulares Vorgehensmodell. 2004. – URL http://www4.in.tum.de/ modellierung2004/ tutorials/rausch abstract.html (13.02.2004)

[Ravella und Lum 2004] Ravella, J. ; Lum, R.: Free as in Freedom. In: Software Development Magazine (2004), März

[Raymond 2001] Raymond, E.S.: The Cathedral & the Bazaar: Musing on Linux and Open Source by an Accidental Revolutionary. 2. Auflage. O'Reilly & Associates, 2001. – URL http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/(27.08.2004)

[Royce 1970] Royce, W.W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: Proceedings of the IEEE WESTCON, IEEE, August 1970, S. 1–9

[Rumbaugh et al. 1999] Rumbaugh, J. ; Jacobson, I. ; Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999

[Seyff et al. 2004] Seyff, N.; Grünbacher, P.; Maiden, N.; Tosar, A.: Mobile Werkzeuge im Requirements Engineering. In: Softwaretechnik-Trends, 24 (2004) 1.

[Sommerville 2002] Sommerville, I.: Software Engineering. 6. Auflage. Addison-Wesley, 2002. – Trial Version 1.0

[Sommerville 2004] Sommerville, I.: Software Engineering. 7. Auflage. Addison-Wesley, 2004.

[Stahlknecht und Hasenkamp 2003] Stahlknecht, P. ; Hasenkamp, U.: Einführung in die Wirtschafsinformatik. 10. Auflage. Springer, 2003

[Sträter 2002] Sträter, K.: Erfolgswirkung von Entwicklungsaktivitäten, Universität Karlsruhe (TH), Dissertation, 2002

[Stein et al. 1997] Stein, Michael; Riedl, John; Harner, Soren J.; Mashayekhi, Vahid: A Case Study of Distributed, Asynchronous Software Inspection. In: Proceedings of the 19th International Conference on Software Engineering (ICSE '97), S. 107-117.

[Szyperski et al. 2002] Szyperski, C. ; Gruntz, D.W. ; Murer, S.: Component Software : Beyond Object-Oriented Programming. 2. Auflage. Addison-Wesley, 2002 (Component Software Series)

[Szyperski und Spinellis 2004] Szyperski, C. ; Spinellis, D.: How is Open Source Affecting Software Development. In: IEEE Software 21 (2004), Nr. 1, S. 28–33

[Thaller 2001] Thaller, G.E.: ISO 9001:2000 – Softwareentwicklung in der Praxis. 3. Auflage. Verlag Heinz Heise, 2001

[Torvalds und Diamond 2002] Torvalds, L. ; Diamond, D.: Just for Fun: The Story of an Accidental Revolutionary. HarperBusiness, 2002

[Turowski 1999] Turowski, K.: Ordnungsrahmen für komponentenbasierte betriebliche Anwendungssysteme. In: Turowski, K. (Hrsg.): Tagungsband 1. Workshop komponentenorientiere betriebliche Anwendungssysteme (WKBA 1), 1999, S. 3–14

[Versteegen 2000] Versteegen, G.: Projektmanagement mit dem Rational Unified Process. Springer, 2000

[Versteegen u. a. 2001] Versteegen, G. ; Salomon, K. ; Heinold, R.: ChangeManagement bei Softwareprojekten. Springer, 2001 (Xpert.press)

[VMXT 2005] V-Modell XT Dokumentation komplett, Version 1.01, Bundesministerium des Inneren, Berlin, 2005,
http://www.kbst.bund.de/static/pdf/V_Modell_XT_Komplett.pdf, 3.5.2005.

[Weischedel 2003] Weischedel, G. ; Versteegen, G. (Hrsg.): Konfigurationsmanagement. Springer, 2003 (Xpert.press)

[Zeller und Krinke 2004] Zeller, A. ; Krinke, J.: Open-Source- Programmierwerkzeuge: Versionskontrolle – Konstruktion – Testen – Fehlersuche. 2. Auflage. dpunkt, 2004

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

Anhang 1 Die NIMSAD-Methode

Elemente, Phasen und Stufen	Fragenkatalog
Methodikkontext (MK)	
Anwendungssituation	In welchen Situationen passt die Methodik?
Beginn der Methodikanwendung	Welche Ereignisse lösen die Anwendung der Methodik aus?
Abnehmer und Problembesitzer	Wer sind die Abnehmer und Problembesitzer (abgesehen von den Anwendern)?
Kontextbeschreibung	Wie ist der Kontext für das zu erstellende System beschrieben?
Kultur und Politik der Methodikanw.	Was ist die Kultur und Politik der Methodikanwendung, explizit und implizit?
Risiken der Kontextbeschreibung	Welche Risiken identifiziert die Methodik, wenn der Kontext beschrieben wird?
Risiken der Methodik	Worin bestehen die Risiken, die Methodik zu verwenden?
Methodikanwender (MA)	
Anwendermotive und -werte	Welche sind die Motive und Werte des MA (implizit)?
Verlangte abstrakte Argumentation	Welche Ebene abstrakter Argumentation wird vom MA verlangt?
Verlangte Fähigkeiten	Welche Fähigkeiten werden von einem MA verlangt, um Aufgaben zu erledigen, die beim Anwenden der Methodik benötigt werden?
Methodik (M)	
Problemsituation und -grenzen	Inwiefern hilft die Methodik beim Verstehen der einzelnen Situation und der Grenzsetzung?
Diagnose der Situation	Wie diagnostiziert man, welche Art von System benötigt wird?
Prognose für das System	Wie erstellt man eine Prognose für das zu erstellende System?
Problemdefinition	Wie definiert man Probleme, die gelöst werden müssen?
Ableiten von fiktiven Systemen	Wie kommt man zu Systemen, die beschrieben werden müssen?
Logischer Entwurf	Ist diese Phase vorhanden? Wie setzt man diese Phase um?
Physischer Entwurf	Ist diese Phase vorhanden? Wie setzt man diese Phase um?
Entwurfsumsetzung	Ist die Implementationsphase beschrieben? Was beinhaltet sie?
Evaluation (E)	
Vor dem Eingriff	Wie werden MK/MA/M vor dem Eingriff bewertet?
Während dem Eingriff	Wie werden MK/MA/M während dem Eingriff bewertet?
Nach dem Eingriff	Wie werden MK/MA/M nach dem Eingriff bewertet?

Tabelle 8: Fragenkatalog der NIMSAD-Analyse

Anhang 2 Vorgehensmodell-Vergleichsrahmen

Hauptkriterium	Unterkriterien
Philosophie	zugrunde liegendes Paradigma primäres Ziel antreibendes Moment intendierte Domänen
Terminologie	Ergebnisse und Artefakte Rollen und Ressourcen Aktivitäten Phasen Techniken Notationen Richtlinien und Standards
Modellumfang	Prozessdokumentation Phasenabdeckung (Softwarelebenszyklus) Prozessabdeckung (Entwicklung und Querschnittsfunktionen) Rollenabdeckung (Rollenmodell)
Prozess	Prozessarchitektur Prozesssteuerung
Komponenten	Komponentenbegriff Komponentenbildung (*Top-Down* oder *Bottom-Up*) Komponentenlebenszyklus
Kollaboration	Kollaborationspunkte Verteilbarkeit der Aufgaben
Adaption	Anwendungstyp organisatorische Rahmenbedingungen domänenspezifische Anforderungen Projektstruktur (Größe und Komplexität)
Werkzeuge	Unterstützung des Entwicklungsprozesses Unterstützung der Querschnittsfunktionen

Tabelle 9: Vergleichsrahmen