Ein Modell der Open-Source-Entwicklung

vorgelegt von
Diplom-Informatiker
Steffen Evers

von der Fakultät IV – Elektrotechnik und Informatik –
der Technischen Universität Berlin

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr. Ing. –

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Thomas Magedanz
Gutachter: Prof. Dr. Bernd Mahr
Prof. Dr. Bernd Lutterbeck

Tag der wissenschaftlichen Aussprache:
4. November 2008

Berlin 2008
D83

Zusammenfassung

In den letzten zwanzig Jahren hat sich in Bezug auf Software viel verändert: die abgedeckte Funktionalität, wie ihre Umgebung aussieht, wie sie wirtschaftlich verwertet wird und letztendlich auch wie sie entwickelt wird. In Bezug auf die Vorgehensweise stellt die Open-Source-Entwicklung wohl die größte Abweichung von den klassischen Methoden dar. Die altbewährten Modelle für eine deterministische Entwicklung von Softwareprodukten sind dafür unzureichend. Trotzdem haben beide Bereiche viele gemeinsame, wesentliche Strukturen und Konzepte, die jedoch aufgrund der großen Unterschiede nur schwer zu erkennen sind. An dieser Stelle setzt die vorliegende Arbeit an. Sie versucht aus der klassischen Softwaretechnik jene grundlegenden Konzepte zu extrahieren, die sie mit der Open-Source-Methodik teilt, und sie von den Elementen zu befreien, die dafür unpassend sind (primär Determinismus, Planung und Produktorientierung). Das Ergebnis ist eine allgemeine, theoretische Fundierung von Softwareentwicklung, die auch auf die Open-Source-Methodik anwendbar ist, ohne dabei ihre Gültigkeit für bisherige Methoden zu verlieren. Es handelt sich dabei um so etwas wie eine Metasprache, die es ermöglichen soll, über Open-Source-Entwicklung zu sprechen. Sie umfasst vier Perspektiven, mit deren Hilfe jeweils andere Aspekte der Entwicklung beleuchtet werden:

Komponenten-Perspektive Es wird das Konzept der systematischen Wiederverwendung von Softwarekomponenten mit Hilfe von Komponentensystemen behandelt.

Prozess-Perspektive Die Zerlegung einer Aufgabe und des damit verbunden Entwicklungsvorgangs in mehrere Teile wird betrachtet (Phasen, Arbeitspakete, Zyklen, etc.).

Agenten-Perspektive Die Form der Zusammenarbeit der beteiligten Akteure und die dabei verwendeten Mechanismen zur Abstimmung ihrer Aktivitäten werden behandelt.

Kontextmodell-Perspektive Software dient als Werkzeug für ihre Umgebung, daher hat ihr Kontext eine zentrale Bedeutung für ihre Entwicklung. Diese Perspektive befasst sich mit Strategien und Methoden zum Umgang mit dem Kontext.

Basierend auf diesen Konzepten wurde ein Modell der Open-Source-Entwicklung erstellt, das deren wesentliche Vorgänge erfasst und sie durch die gemeinsame Basis mit anderen Entwicklungsmethoden vergleichbar macht.

Die theoretische Fundierung kann nun allgemein verwendet werden, um Entwicklungsvorgänge zu beschreiben. Das damit erfasste Modell der Open-Source-Entwicklung ist hilfreich für weitergehende Forschungen in diesem Bereich. So konnte mit Hilfe dieses Modells die zentrale Rolle der Selbstorganisation der Akteure und ihrer Aktivitäten bei dieser Entwicklungsform zusammen mit den zugehörigen Hintergründen und Konzepten erörtert werden.

Summary

In the last twenty years things have changed a lot around software: the provided functionality, its environment, the associated business domain and as a consequence the way it has been developed. Regarding the methodology Open Source development probably represents the biggest contrast to the classical approaches. The well-tried and established models for a deterministic development of software are inappropriate for the Open Source processes. However, both things have essential structures and concepts in common, but they are concealed by all their differences and therefore hard to reveal. This is the starting point of the thesis. Fundamental concepts from the field of established software engineering that are also valid for Open Source processes are extracted and inappropriate aspects are removed (primarily determinism, planning, and product orientation). The result is a general theoretical foundation of software development which is also applicable for Open Source without losing its validity for other approaches. It can be considered as a kind of meta language that makes it possible to talk efficiently about Open Source development and comprises four perspectives that each highlight different aspects of development activities:

Component-Perspective The concept of systematic reuse of software components with the help of component systems is discussed.

Process-Perspective The segmentation of a task and the related development activities in several smaller parts is examined (phases, work packages, cycles, etc.).

Agent-Perspective Different modes of collaboration of the various stakeholders and the underlying mechanisms to coordinate their activities are highlighted.

Context-Model-Perspective Software serves as a tool for its environment. Therefore the context plays a central role during its development. This perspective covers the strategies and methods to handle the context.

Based on these concepts, a model of Open Source development is created that covers its essential activities and makes it comparable to other development methods by providing a shared foundation.

The introduced theoretical framework can be used to describe development activities of software systems in general. The created model of Open Source development is useful for further studies of the Open Source phenomenon. As a first step, it has been used in the thesis itself to discuss the important role of self-organization, its background and the underlying concepts for the actors and their activities within this kind of development.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Die folgende Arbeit ist die Essenz meiner Forschung zum Thema Open-Source-Software, die vor neun Jahren mit meiner Diplomarbeit zu “Umgebungen der Open-Source-Softwareentwicklung” bei Prof. Dr. Bernd Mahr am Lehrstuhl “Formale Modelle, Logik und Programmierung” an der Technischen Universität Berlin begann (s. [Evers 00]) und dort in den folgenden Jahren in meiner Rolle als wissenschaftlicher Mitarbeiter und Doktorand vertieft wurde.

Am Anfang meiner Beschäftigung mit diesem Themenbereich war die wissenschaftliche Gemeinschaft, welche sich damit auseinandersetzte, doch sehr überschaubar. Zudem konzentrierte sich diese Forschung primär auf andere Disziplinen (z.B. Recht, Gesellschaft und Wirtschaft). Obwohl sich dies im Verlauf der Arbeit änderte, gestaltete sich am Anfang aufgrund dieses Mangels die wissenschaftliche Auseinandersetzung mit diesem Thema problematisch. Um so wichtiger war für mich meine Tätigkeit als wissenschaftlicher Mitarbeiter und die damit verbundenen Möglichkeiten.

Dadurch konnte ich im Rahmen von acht Lehrveranstaltungen zu Open-Source mit über 150 Studenten meine theoretischen Überlegungen diskutieren und sie durch dutzende studentische Open-Source-Projekte verifizieren. Im Rahmen dieser Kurse fanden auch mehrere Workshops mit zahlreichen Gästen aus der Wirtschaft und der Open-Source-Gemeinschaft statt, die mit ihren Erfahrungsberichten aus der Praxis ebenfalls einen wesentlichen Beitrag leisteten. Die Betreuung von vier daraus resultierenden Diplomarbeiten zum Thema Open-Source und die Auseinandersetzung der Diplomanden mit meinen Überlegungen lieferten mir ebenfalls wertvolle Rückmeldungen (s. [Gnodtke 03, Suhr 07, Tolzmann 07, Klink 07]).

Aber auch die anderen Lehrveranstaltungen, an denen ich beteiligt war und die in keinem direkten Bezug zu Open-Source standen, haben mir geholfen. In den Praktika zur Objektorientierten Programmierung mit insgesamt über 350 Teilnehmern vermittelte mir die Betreuung der Studenten und die Korrektur ihrer Quelltexte einen Eindruck, wie Informatiker allgemein bei der Entwicklung vorgehen und die theoretischen Vorgaben tatsächlich umsetzen. In den Seminaren und Vorlesungen zu Architekturen, Modellierung, Modelltheorie und offenen verteilten Systemen konnte ich vom umfangreichen Wissen von Bernd Mahr in diesen Bereichen profitieren und mir die notwendigen theoretischen Grundlagen für meine Dissertation aneignen, ohne die ich mich in diesen abstrakten Welten verloren hätte. Insbesondere seine Arbeiten zur Modelltheorie waren dabei von unschätzbarem Wert.

Während dieser langen Zeit halfen mir die vielen ausführlichen Gespräche mit meinem Doktorvater, so manche inhaltliche Untiefe zu umschiffen und die zahlreichen Schwierigkeiten meines Themas zu überwinden. Aber auch die umfangreichen Diskussionen im Kollegen-, Freundes- und Familienkreis trugen wesentlich dazu bei, inhaltlich die “Spreu vom Weizen” zu trennen.

Außerhalb des Lehrstuhls fanden sich mit der Zeit ebenfalls Gesprächspartner, die sich dem Thema Open-Source widmeten. So führte der alljährliche LinuxTag und der Kongress “Open Source

Meets Business" zu zahlreichen Kontakten in der Wirtschaft und der Open-Source-Gemeinde. Dabei entstand auch das "Open Source Research Network", in dem sich Doktoranden und Interessierte zu diesem Thema austauschen.

Ich möchte mich hiermit bei allen recht herzlich bedanken, die in der einen oder anderen Form einen Beitrag zu meiner Promotion geleistet haben. Mein besonderer Dank gilt jedoch meinem Doktorvater Bernd Mahr. Ohne seine Unterstützung wäre diese Arbeit so nicht möglich gewesen.

1 Einleitung

1.1 Motivation

Die Untersuchung der Open-Source-Entwicklungsarbeit wirft grundlegende Fragen auf wie Software entwickelt werden kann und wie sie entwickelt werden soll. Dafür sind einerseits die grundlegenden Vorgänge und Strukturen zu klären und andererseits die einflussgebenden Randbedingungen bzw. entscheidenden Faktoren zu erfassen.

Softwareprogramme sind Modelle von Prozessen und ihrer relevanten Umgebung, die durch einen Computer interpretiert werden können. Zu dieser relevanten Umgebung zählt der Diskursbereich in der realen Welt, die zugrundeliegende Hardware und andere Softwareelemente, mit denen das jeweilige Programm in Beziehung steht.

Softwareentwicklung umfasst jene Vorgänge, die zur Entstehung dieses Modells führen. Dies beinhaltet zwei wesentliche Aufgaben: die Erstellung dieses Modells als abstraktes Gebilde und seine Codierung in eine maschinenlesbare Repräsentation.

Diese Softwareentwicklung fand in der Vergangenheit zum Großteil in einem kommerziellen Umfeld statt und wurde von Investoren vorfinanziert, um damit ein kommerzielles Produkt zu erstellen, das anschließend gewinnbringend verkauft werden sollte. Dieses Geschäftsmodell wird heute als “Software as a Product” bezeichnet.

Zur Wahrung der Investionssicherheit wurde dabei einerseits im Vorfeld festgelegt, wie das Ergebnis aussehen sollte (z.B. in Form eines verbindlichen Vertrags oder einer Spezifikation) und anderseits der Entwicklungsvorgang detailliert geplant.

Dafür benötigt man eine Form der deterministischen Entwicklung, die berechenbar, kontrollierbar und steuerbar ist. Der Großteil der klassische Softwaretechnik dient diesem Zweck und wurde dafür entwickelt. Ein solches Vorgehen setzt jedoch folgende Gegebenheiten voraus:

Aufgabe Die Funktionalität, die durch die angestrebte Software bereitgestellt werden soll, lässt sich klar und detailliert spezifizieren und von anderen Vorhaben abgrenzen, mit denen sie in Beziehung steht (z.B. durch Schnittstellen).

Kontext Die relevante Umgebung für die Software weist eine erkennbare Ordnung auf. Diese Ordnung ist stabil und einfach genug, dass sie durch ein passendes Modell mit den bereitgestellten Mitteln erfasst werden kann und während der Verwendung der Software erhalten bleibt.

Entwicklung Es existiert ausreichendes Wissen in Bezug auf Aufgabe, Kontext und Entwicklungsarbeiten, um eine zuverlässige Prognose in Bezug auf den Verlauf der Entwicklung vornehmen zu können.

Soweit diese Bedingungen nicht auf natürliche Weise gegeben sind, müssen sie künstlich herbei geführt werden, indem man gewisse Abstriche in Kauf nimmt. Man abstrahiert von relevanten

Bereichen der Umgebung, die diesen Anforderungen nicht genügen, definiert vorab Schnittstellen zu anderen Vorhaben, ohne dabei die tatsächlichen Umstände vollständig zu berücksichtigen und akzeptiert provisorische Lösungen bzw. ignoriert absehbare Schwierigkeiten, um das Entwicklungsvorhaben innerhalb der Planvorgaben zu halten.

Durch das zunehmende Vordringen der Computer in den menschlichen Lebensraum, die Ausdehnung ihrer Aufgabengebiete und die zunehmende Vernetzung wird Software und ihre Entwicklung jedoch immer komplexer und die daraus resultierenden, notwendigen Abstriche zugunsten eines deterministisches Vorgehens immer größer.

Trotz all dieser Bemühungen und Abstriche schaffen nur 16 bis 50 Prozent der Softwareprojekte einen plangemäßen Abschluss, je nachdem welche Statistik man heranzieht und welches Jahr man betrachtet (vgl. z.B. [ChaosReport 03, Buschermöhle 06]).

In der Softwaretechnik befindet man sich permanent auf der Suche nach Alternativen, die eine Verbesserung dieser Situation bringen können. So hat sich im Jahre 2001 die "Agile Alliance" gebildet, die eine Veränderung der Prioritäten in der Softwareentwicklung befürworten und sie in ihrem folgenden Manifest ausdrücken (Manifesto for Agile Software Development `http://www.agilemanifesto.org/`, [Balzert 08, S. 653]):

> Wir finden bessere Wege, Software zu entwickeln, indem wir es tun und anderen dabei helfen. Durch diese Arbeit sind wir zu der folgenden Gewichtung von Werten gelangt:
>
> - Einzelpersonen und Interaktionen wichtiger als Prozesse und Werkzeuge.
> - Laufende Systeme wichtiger als umfassende Dokumentation.
> - Zusammenarbeit mit dem Kunden wichtiger als Vertragsverhandlungen.
> - Reaktion auf Änderungen wichtiger als das Verfolgen eines Plans.
>
> Obwohl die Dinge auf der rechten Seite wichtig sind, legen wir mehr Wert auf die Dinge auf der linken Seite.

Damals war dies etwas Neues. Heute stellen agile Prozesse den allgemeinen Trend dar oder wie Ivar Jacobsen es auf einer Entwicklerkonferenz 2006 formulierte [Balzert 08, S. 651]:

> These days, to say that you're not agile is the equivalent of saying that you're not potent.

Letztlich steckt hinter der Idee von agilen Prozessen der Versuch, den ursprünglichen Determinismus zurückzudrängen und mehr Freiräume und Flexibilität für die Durchführung der Softwareentwicklung zu schaffen, um sich an die veränderten Gegebenheiten anzupassen. Balzert charakterisiert die agilen Prozesse wie folgt [Balzert 08, S. 651]:

- "Versuch, einen Kompromiss zwischen *keinem* und *zu viel* Prozess zu finden, so dass gerade soviel Prozess vorhanden ist, damit sich der Aufwand lohnt.
- Es werden möglichst wenig Dokumente gefordert, im Extremfall ist der Code das Dokument.
- Iterative Entwicklung mit häufig ausgelieferten lauffähigen Versionen des Zielsystems, die jeweils eine Teilmenge der geforderten Eigenschaften enthalten. Diese lauffähigen Versionen besitzen oft nur wenig Funktionalität, sollten aber bereits Vertrauen in das endgültige

System geben. Die lauffähigen Versionen sollen voll integriert und so sorgfältig getestet werden wie eine endgültige Auslieferung.

- Stabile Pläne sind Pläne für kurze Zeiträume und werden für jeweils eine einzelne Iteration gemacht."

Betrachtet man diese Charakterisierung näher, so beschreibt dies auch wesentliche Merkmale, welche die Open-Source-Entwicklung von klassischen Methoden abgrenzt: "Code das Dokument", "Iterative Entwicklung mit häufig ausgelieferten lauffähigen Versionen", "Pläne für kurze Zeiträume".

Der Erfolg der agilen Prozesse und der Open-Source-Methode sind beides Indizien für die allgemeine Bewegung weg von einer deterministischen Softwareentwicklung.

Im Open-Source-Bereich geht man jedoch noch einen Schritt weiter, indem man weitgehend die wirtschaftliche Nutzung von der Softwareentwicklung durch die Verwendung von Open-Source-Lizenzen entkoppelt, die auf wesentliche Urheberrechte verzichten. Letztendlich ist dadurch die Software selber nicht mehr das wirtschaftliche Gut, sondern das damit verbundene Wissen und die darauf basierende Dienstleistungen und Produkte (vgl. Tab. 1.1).

Grundlage	Geschäftsmodelle
Betrieb (inkl. Wartung) übernehmen	Softwarebasierte (Web-)Dienste
Wartung übernehmen	Wartungsverträge
Wartung unterstützen	Hotlines, Beratung, Second-Level-Support
jmd. zur Wartung befähigen	Schulungen, Bücher
Software fortlaufend überarbeiten	Subscription Modelle

Tabelle 1.1: Alternative Geschäftsmodelle für Software

Dies entspricht dem allgemeinen Trend in der Informationstechnologie Software als Dienstleistung anzubieten ("Software as a service") statt sie als Produkt zu verkaufen.

Wenn Software als Produkt verkauft wird, ist die wesentliche Leistung des Produzenten mit der Auslieferung erfüllt. Handelt es sich um ein dienstleistungsbasiertes Geschäftsmodell gehen die Pflichten des Anbieters darüber hinaus oder beginnen sogar erst später, schließen i.d.R. in irgendeiner Form die Wartung mit ein und erstrecken sich evtl. über den gesamten Nutzungszeitraum der Software. Dabei muss dann nicht nur die selbst produzierte Software betrachtet werden, sondern das gesamte genutzte Computersystem. Der Verantwortungsbereich wird also bei solchen Geschäftsmodellen verschoben bzw. ausgedehnt.

Open-Source-Software bietet zusätzlich durch ihre Veröffentlichung der Konkurrenz die Möglichkeit eine ähnliche oder sogar die gleiche Dienstleistung anzubieten und damit den Urhebern direkt Konkurrenz zu machen, was im proprietären Bereich weiterhin durch entsprechende Lizenzen verhindert wird.

Auf diese Weise wird jedoch oftmals eine Kooperation angestoßen, die weit über Organisationsgrenzen und das bisherige Rollenverständnis von Produzent und Konsument hinaus geht. Diese Kooperation gab es wohl zunächst nur in der Open-Source-Gemeinschaft, aber man hat auch in dem proprietären Bereich die Chance einer solchen Zusammenarbeit erkannt und versucht sie darauf zu übertragen (z.B. Microsofts "Shared Source Program").

Allgemein kann man sagen, dass sich in den letzten zwanzig Jahren viel verändert hat: welche Funktion Software übernimmt, wie ihre Umgebung aussieht, wie sie wirtschaftlich verwertet wird und letztendlich auch wie sie entwickelt wird. Die größte Abweichung von den klassischen Methoden stellt dabei wohl die Open-Source-Entwicklung dar. Die altbewährten Modelle für eine deterministischen Entwicklung von Softwareprodukten ist für sie völlig unzureichend. Trotzdem haben sie viele gemeinsame, wesentliche Strukturen und Konzepte, die jedoch aufgrund der großen Unterschiede nur schwer zu erkennen sind.

An dieser Stelle setzt die vorliegende Arbeit an. Sie versucht aus der klassischen Softwaretechnik jene grundlegenden Konzepte zu extrahieren, die sie mit der Open-Source-Methodik teilt, und sie von den Elementen zu befreien, die dafür unpassend sind (primär Determinismus, Planung und Produktorientierung).

Das Ergebnis ist eine allgemeine, theoretische Fundierung von Softwareentwicklung, die auch auf die Open-Source-Methodik anwendbar ist, ohne dabei ihre Gültigkeit für andere Methoden zu verlieren. Es handelt sich dabei um so etwas wie eine Metasprache, die es ermöglichen soll, über Open-Source-Entwicklung zu sprechen.

Basierend auf diesen Konzepten wurde ein Modell der Open-Source-Entwicklung erstellt, das deren wesentliche Vorgänge erfasst und sie durch die gemeinsame Basis mit anderen Entwicklungsmethoden vergleichbar macht.

1.2 Anmerkung zu Literatur und Begrifflichkeit

Es konnte trotz intensiver Suche keine Metasprache oder etwas Ähnliches gefunden werden, um die wesentlichen Strukturen, Abläufe und Konzepte zu beschreiben, die der Open-Source-Entwicklung zugrundeliegen. Stattdessen wurde die Fachliteratur nach passenden Ansätzen und Modellen durchsucht und passende Konzepte zu einem Metamodell zusammengefügt.

Ich habe mich dafür entschieden, als Ausgangspunkt die klassische Softwaretechnik zu verwenden. Sie bietet ein breites Spektrum an Literatur aus verschiedenen Bereichen, das zur theoretischen Fundierung der Arbeit dienen kann: Softwaremanagement (z.B. [Balzert 08]), Requirements-Engineering (z.B. [Rupp 07]), Projektmanagement (z.B. [PMBOK 00]), Komponentenbasierte Entwicklung (z.B. [Heineman 01]), Vorgehensmodelle (z.B. [Abran 04]), Softwarearchitekturen (z.B. [Vogel 05]), empirische Softwaretechnik (z.B. [Endres 03]), Konfigurationsmanagement (z.B. [Haug 01]) und Produktlinienmanagement (z.B. [Clements 07]).

Zudem wurden angrenzende Forschungsgebiete studiert, um ein Verständnis des wesentlichen Hintergrunds zu erhalten und entsprechende Grundlagen entwickeln zu können: Systemtheorie (z.B. [Flood 93]), Modelltheorie (z.B. [Mahr 08]), Erkenntnistheorie (z.B. [Vollmer 90]), Wissenschaftstheorie (z.B. [Wilkins 95]), Managementlehre (z.B. [Staehle 99]), Selbstorganisation (z.B. [Göbel 98]), Komplexe Adaptive Systeme (z.B. [Axelrod 99]) und Multiagenten-Systeme (z.B. [Woolridge 01]).

Dabei konnten viele nützliche Strukturen und Konzepte gefunden werden. Die in dieser Arbeit dargestellten Grundlagen sind vor diesem Hintergrund zu sehen. Allerdings war es oftmals nicht möglich diese durch direkte Referenzen zu belegen. Einerseits sind sie in ihrem fachspezifischen Kontext eingebettet und damit nicht direkt verwendbar. Andererseits spielen sie in den

jeweiligen Arbeiten oftmals eine untergeordnete Rolle, werden nur implizit oder indirekt verwendet, nicht weiter begründet und in dem jeweiligen Forschungsbereich allgemein anerkannt. Im Referenz-Modell für offene verteilte Systeme (vgl. [ISO10746-1 98]) wurde dieses Problem durch eine entsprechende Abstraktion der Begriffe und Konzepte auf das Essentielle gelöst, was dort die Verteilung ist. In Anlehnung an diese Vorgehensweise stellt ein Großteil der Grundlagen in dieser Arbeit daher Verallgemeinerungen von vielen unterschiedlichen Konzepten aus den obig aufgeführten Forschungsbereichen dar. Es geht oftmals darum das Implizite explizit und damit selbst zum Gegenstand der Betrachtung zu machen. So befasst sich z.B. Abschnitt 4.3 mit der allgemein üblichen Zerlegung von einer größeren Aufgabe in mehrere Teile. Dieses allgemeine Konzept wird dort analysiert und die Folge der unterschiedlichen Größe von Arbeitspaketen (Schrittgröße) für den Gesamtverlauf untersucht. Eine entsprechende explizite Verbindung derartig allgemeiner Konzepte mit der Literatur wäre mühsam, schwer zu lesen und aus meiner Sicht nicht hilfreich.

Da Standards bereits eine entsprechende Verallgemeinerung darstellen, wurden sie an passenden Stellen referenziert (z.B. [IEEE12207.0 96]). Weil sie sich jedoch i.d.R. ebenfalls auf die kommerzielle Entwicklungsarbeit beziehen, waren sie nur bedingt verwendbar und konnten nicht als alleinige, theoretische Grundlage dienen.

Diese Arbeit verwendet Elemente aus ganz unterschiedlichen Bereichen. Deshalb tauchen in den jeweiligen Abschnitten Begriffe mit ähnlichen Bedeutungen auf (z.B. Agent, Rolle, Akteur), die hier nicht weiter gegeneinander abgegrenzt werden. Eine konsistente Begrifflichkeit über alle diese Bereiche hinweg gibt es bisher nicht. Teilweise definiert sich jeder Autor seine zentralen Begriffe neu bzw. evaluiert verschiedene Definitionen, wie es hier mit Komponenten gemacht wurde (vgl. Abs. 3.1.2). Dies für alle verwendeten Begriffe durchzuführen, würde den Rahmen dieser Arbeit sprengen. Stattdessen wurde versucht die meisten Begriffe entsprechend ihrem allgemeinen Verständnis zu benutzen bzw. ihre Bedeutung durch die Verwendung im zugehörigen Kontext zu verdeutlichen.

Aus diesem Grund fehlt teilweise auch die Verbindung zwischen einzelnen Teilen durch gemeinsame Begriffe, da die Ausdrücke aus den jeweiligen Quellen verwendet wurden. Die inhaltliche Verbindung wurde dann jedoch durch den Text hergestellt bzw. ergibt sich aus dem Kontext. So wurde in Abschnitt 5.1 das Vokabular aus dem Projektmanagement zur Beschreibung belassen und inhaltlich in Abschnitt 5.3 der kooperativen Entwicklung mit ihrer eigenen Begrifflichkeit gegenüber gestellt.

Eine ähnliche Problematik zeigte sich bei der darauf folgenden Beschreibung der Entwicklungsvorgänge im Open-Source-Bereich. Es gibt umfangreiche Literatur zur Open-Source-Entwicklung, wie folgende Beispiele zeigen:

- Katalog von Online-Veröffentlichungen zum Thema Open-Source: Free/Open Source Research Community `http://opensource.mit.edu`
- Open Source Jahrbuch: Zwischen Softwareentwicklung und Gesellschaftsmodell [Lutterbeck 04, Lutterbeck 05, Lutterbeck 06, Lutterbeck 07, Lutterbeck 08]
- Perspectives on Free and Open-Source-Software [Feller 05]
- Modell und Optimierungsansatz für Open-Source-Softwareentwicklung [Dietze 04]
- Free/Open Source Software Development [Koch 04]
- Open-Source-Software: Eine ökonomische und technische Analyse [Brügge 04]

- The Success of Open Source [Weber 04]
- Understanding Open Source Software Development [Feller 02]
- Embracing Insanity: Open Source Software Development [Pavlicek 00]
- The Cathedral & the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary [Raymond 99]
- Open Sources - Voices from the Open Source Revolution [DiBona 99]

Darin werden wesentliche Strukturen, Vorgänge, Zusammenhänge und Hintergründe dargestellt. In der vorliegenden Arbeit wurde jedoch ein neues Metamodell zur Beschreibung der Entwicklungsvorgänge erstellt, das auf den extrahierten Konzepten der klassischen Softwaretechnik basiert. Zur Erfassung der Vorgänge im Open-Source-Bereich mit diesem Hilfsmittel war es daher notwendig auch ein eigenes Modell der Open-Source-Entwicklung zu erstellen, das an vielen Stellen durch entsprechende Referenzen untermauert werden konnte.

Es stellte sich jedoch heraus, dass die hier betrachteten Bereiche der Open-Source-Entwicklung in der bisherigen Open-Source-Forschung nur eine relativ untergeordnete Rolle spielen:

Kapitel 2 Entwicklung von Computersystemen als vollständige funktionale Einheit

Kapitel 3 Komponentenbasierte Software/Systementwicklung

Kapitel 4 Entwicklung durch Wartung bzw. Lebenszyklus einer Veränderung

Kapitel 5 Abgrenzung der kooperative Entwicklungsarbeit gegen klassische Projekte

Kapitel 6 Umgang mit dem Kontext einer Software

Kapitel 7 Rolle der Selbstorganisation in der Entwicklungsmethodik

Daher war es ähnlich wie bei der Erstellung des Metamodells oftmals nicht möglich entsprechende Vorgänge mit Referenzen direkt zu belegen, obwohl es sich dabei um alltägliche Vorgänge im Open-Source-Bereich handelt. So konnte z.B. der Open-Source-Lebenszyklus in Abschnitt 4.5.2 in dieser oder einer ähnlichen Form nicht in der Literatur gefunden werden, obwohl es den typischen Weg einer Veränderung im Open-Source-Bereich von der Idee bis zur Nutzung beschreibt, der in der Praxis leicht nachzuvollziehen ist. Teilabschnitte oder der Zyklus in einer anderen Abstraktion sind dafür in vielen Dokumenten enthalten (z.B. [Dietze 04]).

Das vorgestellte Open-Source-Modell sollte daher hier ebenfalls verstanden werden als das Hervorheben von Strukturen und Vorgängen, die größtenteils in der Literatur implizit in verschiedener Form bereits erfasst wurden und in der Praxis oftmals eine Selbstverständlichkeit darstellen.

Die eigentliche Schwierigkeit war dabei nicht die Entdeckung, sondern das Wesentliche vom Unwesentlichen zu trennen, die richtige Abstraktionsebene zu finden, alle Teile möglichst aufeinander abzustimmen und eine gemeinsame Basis mit der klassischen Softwareentwicklung zu finden.

1.3 Abgrenzung der Open-Source-Entwicklung

In diesem Abschnitt wird zunächst erörtert, was im Weiteren unter den Begriffen “Open Source” und “Entwicklung” verstanden wird und warum ihre Kombination so problematisch ist. Zunächst gibt es mehrere, unterschiedliche Versuche, das Open-Source-Phänomen begrifflich oder

definitorisch zu erfassen, von denen im Folgenden einige aufgegriffen werden. Zudem werden in der Softwaretechnik unter den Begriffen “Entwicklung”, “Entwicklungsprozess” oder “Entwicklungsarbeit” typischerweise Vorgänge im Sinne der klassischen Methodik verstanden. Die Übertragbarkeit dieser Konzepte auf den Open-Source-Bereich ist jedoch nicht sichergestellt.

Die vorliegende Arbeit liefert am Ende aufgrund dieser Schwierigkeiten auch keine Definition der Open-Source-Entwicklung, sondern ein Modell, das versucht die wesentlichen Vorgänge und Charakteristika der Entwicklung zu erfassen. Damit geht sie über die reine Begriffsklärung hinaus und zeigt die zentralen Elemente der beobachteten Vorgehensweise auf.

1.3.1 Open-Source-Phänomen

Oftmals wird der allgemeine Begriff “Open Source” als Referenz auf das subjektiv beobachtbare Phänomen verwendet. Wegen seiner großen Unschärfe ist er jedoch für eine sachliche Auseinandersetzung mit dem Thema nicht ausreichend fundiert. Daher ist zunächst zu klären, worauf man sich dabei konkret bezieht, und damit die Frage zu beantworten, was man in diesem Zusammenhang genau darunter versteht. Entsprechend universelle Definitionen sind nur von eingeschränktem Nutzen, da sie entweder viel zu allgemein bzw. abstrakt sind und damit zu wenig aussagen oder dem Phänomen nicht gerecht werden, wie es z.B. bei der Definition der Open Source Initiative der Fall ist [OSI 07, Open Source Definition].

Durch die Betrachtung aus einer passenden Perspektive kann man einzelne Aspekte des Phänomens isolieren und sie als Grundlage für weitere Diskussionen und Forschungsarbeiten verwenden. Dadurch erhebt man nicht mehr den Anspruch der Vollständigkeit (nur Aspekte) und entzieht sich der problematischen Frage nach dem Ding an sich, da das Urteil, ob es sich um ein gültiges Modell des Betrachtungsgegenstands handelt, subjektiv und damit jedem einzelnen überlassen ist.

In Bezug auf die beobachtbare Entwicklungsarbeit in der Open-Source-Gemeinschaft können unterschiedliche Aspekte des Phänomens von Interesse sein. Die folgende Liste soll eine Vorstellung davon geben, wie vielschichtig die Fragestellung nach den Hintergründen der Open-Source-Entwicklung tatsächlich ist:

Vorgehensweise Betrachtet man die Vorgänge in der Gemeinde der Open-Source-Entwickler, so ist es schwer, sie mit den Modellen der proprietären Softwareentwicklung zu erfassen (z.B. [IEEE12207.0 96]). Es steht daher die These im Raum, dass es sich dabei um eine neue Methodik handelt, die in bestimmten Szenarien effizienter ist und bezüglich einiger Aspekte bessere Ergebnisse liefert.

Lizenzen Die Lizenzen legen die Möglichkeiten fest, wer was mit der Software machen kann. Jede Vorgehensweise muss sich an diese Vorgabe halten und sie stellen damit wesentliche Randbedingungen dafür dar.

Geschäftsmodell Das im proprietären Bereich etablierte Geschäftsmodell “software as a product” hat aufgrund der großzügigen Lizenzen im Open-Source-Bereich seine Probleme. Daher suchen sich viele Firmen andere Geschäftsmodelle, mit denen sie aus ihrer Betätigung wirtschaftlichen Nutzen ziehen können. Geschäftsmodelle und Entwicklungsmodelle stehen miteinander in einer engen Beziehung und müssen miteinander vereinbar sein.

Projekte Entwicklung von Open-Source-Software finden zum großen Teil im Rahmen von sogenannten Open-Source-Projekten statt. Ihre Organisation, ihr Ablauf und ihre Steuerung sind wesentlicher Teil der Methodik.

Zusammenarbeit Die Aufteilung und Koordination der Arbeitsprozesse kann aufgrund der hohen Unabhängigkeit der Beteiligten über die Grenzen einzelner Projekte und Organisationen hinaus gehen. Zudem ist auch die Interaktion zwischen diesen Organisationen relevant.

Software Wie die Freigabe von ehemals proprietärer Software unter einer Open-Source-Lizenz gezeigt hat, muss Software für eine erfolgreiche Etablierung im Open-Source-Bereich bestimmte Bedingungen erfüllen (Architektur, verwendete Komponenten, etc.). Sind diese nicht erfüllt, wird die Software eventuell genutzt, aber die meistens erhofften positiven Effekte für ihre Entwicklung bleiben aus. Man sollte daher von einer Wechselbeziehung zwischen Entwicklung und Software ausgehen.

Infrastruktur Der größte Teil der Open-Source-Projekte verwenden ähnliche Hilfsmittel (Mailinglisten, Versionsverwaltung, Bugtracker, etc.). Die Vereinheitlichung wurde durch das Entstehen von Hosting-Plattformen wie Sourceforge noch verstärkt. Zudem gibt es Werkzeuge wie Editoren, Compiler, Debugger oder ganze Entwicklungsumgebungen, die von Entwicklern vorzugsweise genutzt werden. Aber auch für andere Phasen des Lebenszyklus gibt es favorisierte Werkzeuge wie Paketverwaltungssysteme oder Konfigurations- und Installationshilfen. Alle diese Hilfsmittel bilden zusammen eine eng verflochtene und weitestgehend konsistente Infrastruktur, die zur Unterstützung von Entwicklungs- und Wartungsarbeiten genutzt wird. Durch ihre Funktionalität und Verbreitung sorgen sie für eine gewisse Vereinheitlichung der Abläufe und erleichtern damit Entwicklern den Einstieg in neue Projekte. Da es sich bei den Werkzeugen meistens selbst um Open-Source-Software handelt, ist ihre Anpassung an veränderte Abläufe möglich. Es besteht daher auch hier eine enge Wechselbeziehung.

Nutzereinbindung Dem Nutzer die Möglichkeit zu geben, an der Entwicklung in verschiedener Form teilzunehmen, ist typisch für den Open-Source-Bereich. Dabei kann es sich um das Mitverfolgen der Abläufe, Melden von Fehlern, Vorschlagen von Änderungen, Beisteuern von einzelnen Code-Fragmenten oder die Aufnahme als vollwertiges Mitglied ins Entwicklungsteam handeln.

Menschen Es war eines der wesentlichen Motive der Open Source Initiative bei der Einführung des Open-Source-Begriffs, die Verbindung mit dem weltanschaulichen Hintergrund zu lösen, die mit dem Begriff “freie Software” in Beziehung steht, der von der “Free Software Foundation” geprägt wurde. Trotzdem ist die Entstehung der Open Source Gemeinschaft und ihres Software-Pools ohne die Gemeinsamkeiten vieler Entwickler bezüglich ihrer Wertvorstellungen und Antworten auf gesellschaftliche Fragen nicht zu erklären. Man sollte daher den “menschlichen Faktor” in diesem Zusammenhang nicht außer Acht lassen.

1.3.2 Open-Source-Software

Der Begriff “Open Source Software” (OSS) wurde 1997 von Eric Raymond eingeführt und im Rahmen der Gründung der Open Source Initiative 1998 von Bruce Perens durch die Open Source

Definition präzisiert. Sie beruht auf den Debian Free Software Guidelines, die wiederum von der Free Software Definition der Free Software Foundation abgeleitet wurden [OSI 07, History of OSI][Debian 07, Debian Social Contract].

Die Open Source Definition bezieht sich auf Bedingungen an die Lizenzen, mit der eine Software vertrieben wird. Software entspricht dieser Definition, wenn sie im Wesentlichen die Freiheit einräumt, die Software zu nutzen, zu studieren, zu verändern und beliebig weiter zu geben.

Auch wenn eine derartige Lizenz für OSS eine notwendige Bedingung ist, so ist das gegenwärtige Open-Source-Phänomen mit all seinen Facetten damit allein nicht zu erklären. Diese Erscheinung allein auf Fragen der Lizenzierung zu reduzieren, abstrahiert von wesentlichen Aspekten, die es für viele Menschen überhaupt erst interessant macht. Große Teile der Wirtschaft scheinen sogar eine entsprechende Lizenzierung nur als notwendiges Übel anzusehen, um damit andere Vorteile nutzen zu können (vgl. [Brown 02, S. 126]). Einerseits möchte man die vorhandene OSS für eigene Zwecke nutzen und anderseits hofft man, dass im Vorgehen der Open-Source-Gemeinschaft eine Methodik zur Entwicklung von Software enthalten ist, die zumindest in bestimmten Situationen bessere Ergebnisse liefert als bekannte Verfahren. Diese Hoffnung basiert auf der Tatsache, dass durch die Open-Source-Gemeinde in den vergangen Jahren eine beachtliche Menge an Software mit scheinbar geringen Mitteln produziert wurde, die inzwischen in vielen Bereichen eine gleichwertige oder überlegene Alternative zu proprietärer Software darstellt.

Brown/Booch fragen sich deshalb, was “Open Source” eigentlich ist, und kommen zu dem Schluss, dass es viele unterschiedliche Meinungen darüber gibt. Ihrer Ansicht nach basieren die Verwirrungen und Missverständnisse auf der mangelnden Verankerung entsprechender Diskussionen in einem allgemeinen Kontext. Sie verzichten daher bewusst in ihrem Aufsatz darauf, der Liste der bereits vorhandenen Definitionsversuche eine weitere hinzuzufügen. Stattdessen identifizieren sie einige Kernpunkte der Open-Source-Herangehensweise, mit deren Hilfe man Entwicklungsprozesse und Geschäftsmodelle charakterisieren kann. Sie schlagen vor, in entsprechenden Diskussionen den allgemeinen Begriff “Open Source” durch folgende Merkmale zu ersetzen [Brown 02, S. 126]:

open-software Veröffentlichung der Software unter einer Open-Source-Lizenz (inkl. Sourcecode)

open-collaboration internetbasierte Kommunikation und Koordination (Diskussionsgruppen, etc.)

open-process extern sichtbarer Entwicklungs- und Veröffentlichungsprozess

open-releases häufige Veröffentlichung von Arbeitsergebnissen, d.h. es gibt keine großen Sprünge zwischen den freigegebenen Versionen

open-deployment Ausrichtung neuer Produkt-Releases auf Plattformen, die (hauptsächlich) aus Open-Source-Produkten bestehen

open-environment Entwicklung von Systemen mit Werkzeugen die (hauptsächlich) aus Open-Source-Produkten bestehen

In dieser Arbeit wird Software genau dann Open-Source-Software genannt, wenn es alle genannten Merkmale aufweist.

1.3.3 Eingrenzung des Entwicklungsgegenstands

In den letzten zwanzig Jahren wurde eine beträchtliche Menge an Software unter einer Open-Source-Lizenz veröffentlicht. Ein großer Teil davon kann auch wirklich im Sinne von Abschnitt 1.3.2 als Open-Source-Software bezeichnet werden.

Im Zusammenhang mit ihrer Entstehung wird oft von der "Open-Source-Entwicklung" oder dem "Open-Source-Entwicklungsprozess" gesprochen und auf Literatur zu diesem Thema verwiesen, die entsprechende Vorgänge in der Open-Source-Gemeinschaft unter verschiedenen Gesichtspunkten beschreibt (z.B. [Raymond 99, Bezroukov 99, Feller 02, Koch 04]). Dabei bleibt jedoch i.d.R. offen, was genau damit gemeint ist. Einerseits geht aus dem Zusammenhang oftmals nicht hervor, was unter dem Begriff "Entwicklung" bzw. "Entwicklungsprozess" verstanden werden soll, andererseits wird diese Unklarheit auch nicht durch die entsprechende Referenz auf die Literatur beseitigt, da nicht explizit benannt wird, worauf man sich im Detail bezieht. Es bleibt daher meistens dem Leser bzw. dem Zuhörer überlassen, was genau darunter zu verstehen ist.

Zunächst kann man festhalten, dass es sich bei einem Entwicklungsprozess ganz allgemein um ein Modell der Abläufe handelt, die zum Entstehen oder der Veränderung eines bestimmten Gegenstands führen. Dabei ist jedoch oftmals nicht klar, ob die konkreten Abläufe für den konkreten Gegenstand gemeint sind oder die wiederkehrenden, abstrakten Vorgängen, welche die Entwicklung von einer ganzen Klasse von Gegenständen erfassen. Es stellt sich also jeweils die Frage nach dem Gültigkeitsbereich, d.h. ob es sich um ein spezifisches oder ein allgemeines Modell handelt. Ein solches allgemeines Modell kann verstanden werden, als ein Muster von Vorgängen, das zeitlich gesehen parallel und/oder sequentielle mehrfach auftritt. Im letzten Fall wiederholen sich die Abläufe. Im ersten gibt es gleichzeitig mehrere Vorgänge, die zu diesem Muster passen (z.B. mit unterschiedlichen Entwicklungsgegenständen).

In jedem Fall muss man für eine exakte Begriffsdefinition den Entwicklungsgegenstand (spezifisch) bzw. die Klasse von Gegenständen (allgemein) benennen, der/die das Ergebnis eines solchen Open-Source-Entwicklungsprozesses ist/sind (z.B. spezifisch der Linux Kernel oder allgemein alle Open-Source-Softwareelemente). Eine weitere Präzisierung ist dann noch in Hinblick auf die Identität des jeweiligen Entwicklungsgegenstands notwendig, die ebenfalls ein Problem darstellt. So kann es sich einerseits um eine bestimmte Version handeln (z.B. Linux Kernel Version 2.6.24) zu einem konkreten Zeitpunkt (veröffentlicht am 24.01.2008) oder um einen Gegenstand, der über die Zeit verändert wird (Linux Kernel allgemein). Die jahrzehntelange Entwicklung des Linux Kernels seit Beginn ist etwas anderes als die viel kürzeren Arbeiten zwischen zwei Version (z.B. 2.6.23 und 2.6.24).

In der vorliegenden Arbeit sollen nun jene Entwicklungsabläufe betrachtet werden, die maßgeblich zum Entstehen der heute verfügbaren Gesamtheit an Open-Source-Software im Sinne von Abschnitt 1.3.2 geführt haben. Es hat sich als problematisch herausgestellt, eine exakte Definition zu finden, die alle diese Vorgänge umfasst (vgl. [Arief 04]). Daher wurde in der vorliegenden Arbeit stattdessen ein Modell erstellt, das versucht, den wesentlichen Kern zu erfassen und von den weitläufigen Grenzbereichen zu abstrahieren.

Dabei sollen nicht nur spezifische Vorgänge, sondern ein allgemeines Muster von charakteristischen Abläufen dargestellt werden, das eine bestimmte Klasse von Entwicklungsgegenständen hervorbringt. Im Open-Source-Bereich gibt es unterschiedliche Entwicklungsgegenstände,

die eine wesentliche Bedeutung haben: Änderungen (patches), Veröffentlichungen von Open-Source-Projekten (releases), Pakete, Distributionen und natürlich die daraus erstellten Softwaresysteme. Alle diese Gegenstände haben eine gemeinsame Verbindung. Sie stehen in einem engen Zusammenhang mit Komponenten. Patches, Software-Releases und Pakete können als unterschiedliche Formen von Komponenten aufgefasst werden. Da es sich dabei um Open-Source-Software handelt, müssen diese Komponenten auch die in Abschnitt 1.3.2 aufgestellten Anforderungen erfüllen. Distributionen und Softwaresysteme wiederum setzen sich aus solchen Komponenten zusammen, stellen sozusagen Sammlungen von derartigen, aufeinander abgestimmten Komponenten dar.

Da sich alle wesentlichen Elemente auf Komponenten zurückführen lassen, bilden sie auch die Klasse der zentralen Entwicklungsgegenstände. Zudem werden in der Arbeit auch die Entwicklung der Systeme mit einbezogen, die sich aus diesen Komponenten zusammensetzen. Dies umfasst vor allem das resultierende Softwaresystem auf einem Computer und Distributionen, die als Vorlage für solche Softwaresysteme dienen und den Aufwand für ihre Erstellung wesentlich reduzieren.

Es geht also bei dem vorgestellten Modell zunächst um die Abläufe, die zur Entstehung von Open-Source-Softwarekomponenten und ihren konstituierenden Systemen führen und die dahinter liegenden Strukturen. Unter der Open-Source-Entwicklung werden in dieser Arbeit jene Vorgänge verstanden, die durch das hier vorgestellte Modell erfasst werden.

1.4 Vorgehensweise und Aufbau der Arbeit

Wie bereits erörtert, war es nicht möglich, das erstellte Modell der Open-Source-Entwicklung direkt auf vorhandene, theoretische Grundlagen oder eine fundierte Metasprache zu stützen. Stattdessen mussten bereits existierende Ergebnisse im Rahmen dieser Arbeit angepasst werden. Als Ausgangspunkt dafür wurde die kommerzielle Softwareentwicklung gewählt. Sie bietet umfangreiche Modelle, Konzepte, und Hilfsmittel zur Beschreibung von entsprechenden Abläufen, die sich sowohl in der Wissenschaft wie auch in der Wirtschaft bewährt haben und im wissenschaftlichen Prozess über die Jahre gereift sind. Es handelt sich in beiden Fällen um Softwareentwicklung und es war anzunehmen, dass sie eine große Menge an Gemeinsamkeiten aufweisen. Solche Gemeinsamkeiten werden in dieser Arbeit identifiziert, mit Hilfe der klassischen Literatur fundiert und daraus eine allgemeine Theorie gebildet, die sich für beide Bereiche nutzen lässt. Aufgrund der Differenzen können entsprechende Grundlagen jedoch nicht unverändert übernommen werden, sondern müssen zunächst von den spezifischen Details der klassischen Theorie befreit werden, die nicht zur Open-Source-Entwicklung passen.

Im Open-Source-Bereich liegt bei näherer Betrachtung eine faktische Trennung zwischen Entwicklungsarbeit und kommerzieller Verwertung vor. Im Gegensatz dazu ist dies in der klassischen Softwaretechnik eng miteinander verflochten. Daher muss für eine gemeinsame Betrachtung von proprietärer und Open-Source-Entwicklung zunächst von den kommerziellen Aspekte der klassischen Vorgehensweise abstrahiert werden. Es wird sozusagen eine theoretische Fundierung benötigt, die sich ausschließlich auf die Entwicklungsarbeit bezieht und von kommerziellen Anforderungen und Vorgängen abstrahiert. Es handelt sich dabei um ein ähnliches Vorgehen wie bei dem Referenz-Modell für offene verteilte Systeme (RM-ODP) (vgl. [ISO10746-1 98]). Bei der Erstellung dieses Standards wurden ebenfalls ausschließlich Strukturen und Aspekte mit in

das Modell aufgenommen, die für die Verteilung eine Rolle spielen und von allen anderen abstrahiert. In ähnlicher Form soll hier eine theoretische Grundlage geschaffen werden, die sich ausschließlich auf die Entwicklungsarbeit konzentriert und andere Aspekte zunächst ausblendet. Dabei handelt es sich insbesondere um Produktorientierung, Determinismus, verbindliche Zielvorgaben durch Spezifikationen und detaillierte Planung der Abläufe. Zudem sollte es nicht auf eine bestimmten Technologie beschränkt sein.

Durch die Loslösung von den klassischen Strukturen ist es zunächst schwierig den Entwicklungsbegriff einzugrenzen. Dieses Problem wurde ebenfalls in Anlehnung an RM-ODP durch die Verwendung von Sichtweisen gelöst. Jede Sichtweise beleuchtet und strukturiert unterschiedliche Bereiche der Entwicklungsarbeit, die hier allgemein als wesentlich angesehen werden. Alle Perspektiven zusammen ergeben das, was hier unter Entwicklung verstanden wird. Es konnten vier derartige Sichtweisen identifiziert werden:

Komponenten-Perspektive Es wird das Konzept der systematischen Wiederverwendung von Softwarekomponenten mit Hilfe von Komponentensystemen behandelt.

Prozess-Perspektive Die Zerlegung einer Aufgabe und des damit verbunden Entwicklungsvorgangs in mehrere Teile wird betrachtet (Phasen, Arbeitspakete, Zyklen, etc.).

Agenten-Perspektive Die Form der Zusammenarbeit der beteiligten Akteure und die dabei verwendeten Mechanismen zur Abstimmung ihrer Aktivitäten werden behandelt.

Kontextmodell-Perspektive Software dient als Werkzeug für ihre Umgebung, daher hat ihr Kontext eine zentrale Bedeutung für ihre Entwicklung. Diese Perspektive befasst sich mit Strategien und Methoden zum Umgang mit dem Kontext.

Dieser Perspektiven werden jeweils in eigenen Kapitel (3 bis 6) behandelt. Am Anfang jedes Kapitels werden die theoretischen Grundlagen erarbeitet und anschließend damit die entsprechenden Aspekte der Open-Source-Entwicklung beschrieben. Diese fundierten Beschreibungen aus vier verschiedenen Perspektiven bilden zusammen dann das angestrebte Modell der Open-Source-Entwicklung.

In Kapitel 2 wird der Entwicklungsarbeit zunächst ein allgemeiner Rahmen gegeben, indem die Vorgänge in Bezug auf das Computersystem betrachtet werden, auf dem die Software letztendlich zum Einsatz kommt.

In Kapitel 7 wird dann als ein erstes Anwendungsbeispiel das vorgestellte Modell genutzt, um die Rolle der Selbstorganisation in der Open-Source-Entwicklung zu erörtern.

Die Arbeit schließt die Betrachtungen mit einer zusammenfassenden Darstellung der gefunden Ergebnisse ab und erörtert die daraus gezogenen Schlussfolgerungen.

2 Entwicklung von Computersystemen

Einzelne Bestandteile eines Systems sind Gegenstand von Softwareentwicklung. Dabei wird aber oft außer Acht gelassen, dass der Benutzer am Ende nicht vor einer Softwarekomponente sitzt, sondern vor einem Computersystem, das sich aus vielen Software- und Hardwareelementen zusammen setzt, die gemeinsam als eine funktionale Einheit betrieben werden.

Die Neuentwicklung aller dieser Elemente für die Erstellung eines Computersystems ist heute eher die Ausnahme. Stattdessen werden vorhandene Elemente verwendet und dann noch das erarbeitet, was damit nicht abgedeckt werden konnte. Abb. 2.1 veranschaulicht dieses Szenario. Die Akteure (Mitte) entwickeln Komponenten (links), passen sie an und verwenden sie für den Auf- und Umbau von Computersystemen (rechts).

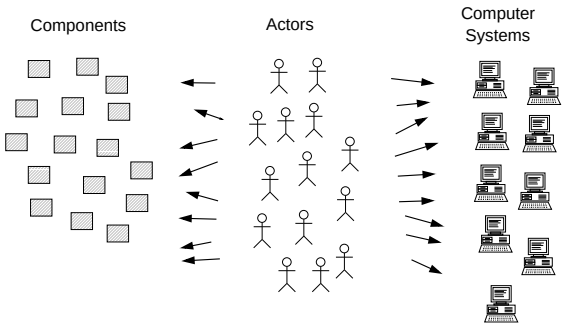

Abbildung 2.1: Systementwicklung: Akteure bauen mit Hilfe von Komponenten Computersysteme

Die genutzten Elemente sind Hardware und Software. Hardware kann eine Festplatte, ein Hauptspeicher-Baustein, ein Drucker, etc. sein. Bei der Software kann es sich um Betriebssystem, Bibliotheken der verwendeten Programmiersprachen, frühere eigene Entwicklungen, Webbrowser, Datenbanken und vieles mehr handeln. Jedoch sind alle diese Dinge normalerweise in einer ständigen Veränderung begriffen. Die Akteure kommen und gehen, Komponenten werden verändert; es werden neue entwickelt und alte geraten ins Abseits, verkümmern oder sind nicht mehr zu gebrauchen, die Hardware wird ersetzt, aufgerüstet, angepasst oder ausgemustert. Die Umgebung der jeweiligen Systeme verändert sich mit der Zeit, es kommen neue Aufgaben dazu, alte fallen weg und die beibehaltenen werden im Ablauf geändert. Dass es sich hierbei nicht um vernachlässigbare Größen handelt, belegen z.B. Studien, die eine durchschnittliche Änderung der Anforderungen von 2% pro Monat zeigen (vgl. [Jones 98]). Diese Vorgänge als Ganzes kommen nicht wirklich zu einem Stillstand oder Ende, sondern gehen kontinuierlich weiter und nur einzelne Elemente, Systeme und Akteure scheiden aus diesen Abläufen vorübergehend oder dauerhaft aus.

In der proprietären Entwicklungsarbeit geht es primär um das Erstellen eines angestrebten Softwareprodukts. Im Open-Source-Bereich steht die Entwicklung der Computersysteme selbst im

Vordergrund. Open-Source-Software wird typischerweise (weiter-)entwickelt, um im Verbund mit anderen Komponenten auf den fokussierten Computersystemen eine bestimmte Funktionalität bereit zu stellen.

Die Motivation hinter diesen Vorgängen bleibt jedoch in beiden Fällen das Bereitstellen von Computersystemen mit der gewünschten Funktionalität. Betrachtet man nun ein einzelnes Computersystem, so wird es zunächst aus einer Menge von Komponenten erstellt und im Laufe seiner Existenz immer wieder durch die Integration, den Austausch oder die Ausmusterung von Komponenten aktualisiert, um den veränderten Anforderung gerecht zu werden bzw. Mängel zu beseitigen. Es handelt sich dabei um alltägliche Vorgänge, die jeder Computerbenutzer kennt, der schon einmal neue Software installiert, vorhandene aktualisiert, eine neue Maus angeschlossen, den Hauptspeicher erweitert oder den Computer einfach zur Reparatur gebracht hat. Das alles sind Beispiele für die Aktualisierung eines Systems durch eine Anpassung der enthaltenen Komponenten. Dies gilt ganz allgemein für Computersysteme, egal, ob es sich dabei um ein Mobiltelefon, den Großrechner in einer Bank, einen normalen Desktop-Rechner oder einen Laptop handelt. Es spielt dabei auch keine wesentliche Rolle, was für ein Betriebssystem sich darauf befindet: Windows, Mac OS X, GNU/Linux oder Solaris. Die grundsätzlichen Vorgänge bleiben die gleichen. Bei der konkreten Ausgestaltung dieser Abläufe gibt es jedoch enorme Unterschiede, die um so größer werden je genauer man die Mechanismen betrachtet.

In der Fachliteratur werden diese Vorgänge in Bezug auf Software in mehreren Bereichen behandelt: Konfigurationsmanagement (z.B. [IEEE828 05, SCM 05, Westfechtel 03, Estublier 99, Magnusson 98]), Produktlinienmanagement (z.B. [Clements 07]), komponentenbasierte Entwicklung (z.B. [Heineman 01]) und in der allgemeinen Softwaretechnik (Auslieferung, Integration, Wartung; z.B. [Balzert 08]). Trotz intensiver Literaturrecherche konnte jedoch kein passendes Modell in ausreichend allgemeinen, abstrakten Form für diesen grundlegenden Sachverhalt gefunden werden. Die in der Literatur dargestellten Sachverhalte beziehen sich entweder auf einen spezifischen technischen Kontext oder befassen sich primär mit den wirtschaftlichen Anforderungen der Vorgänge in Bezug auf Analyse, Planung, Kontrolle und Steuerung. Beides ist für diese Arbeit nicht direkt zu verwenden.

Es wurde daher ein eigenes Modell erstellt, was die Entwicklung von Computersystemen darstellt. Dieses Modell ist der Versuch die in der Literatur geschilderten Sachverhalte und in der Praxis beobachtbaren Phänomene so weit zu abstrahieren und zu vereinheitlichen, dass sie als allgemein gültig anerkannt werden können, sich auf die Open-Source-Entwicklung übertragen lassen und trotzdem ihre Gültigkeit für den proprietären Bereich nicht verlieren.

2.1 Systemknoten als Modell eines Computersystems

2.1.1 Aufbau eines Systemknotens

Ein Systemknoten (system node) SN ist ein Computersystem, das eine vollständige funktionale Einheit darstellt, der in der realen Welt Aufgaben zugeordnet werden. Der für die Bewältigung dieser Aufgaben relevante Ausschnitt der Realität wird hier als Nutzungssystem (usage system) US bezeichnet. Ein SN besteht aus Hardware (physikalische Elemente) und Software (Informationselemente). Aus diesen zwei Elementarten lassen sich zwei Subsystemen bilden: das technische System (technical system) TS und das Informationssystem (information system) IS (vgl.

Abb. 2.2). Dabei übernimmt das IS eine Steuerungsfunktion und das TS eine Ausführungsfunktion. Das IS ist sozusagen das „Gehirn", welches das Verhalten des physikalischen „Körpers" TS steuert und dabei über die Ein-/Ausgabe-Schnittstellen des TS mit anderen Objekten des US interagiert. Man kann das IS verstehen als großes, komplexes Prozessmodell, was dem TS vorgibt, wie es sich in jeder Situation verhalten soll. Diese Handlungsanweisungen werden ergänzt durch im IS gespeicherte Daten, dem „Gedächtnis", die entsprechend dem Prozessmodell gesammelt, gespeichert und ausgegeben werden.

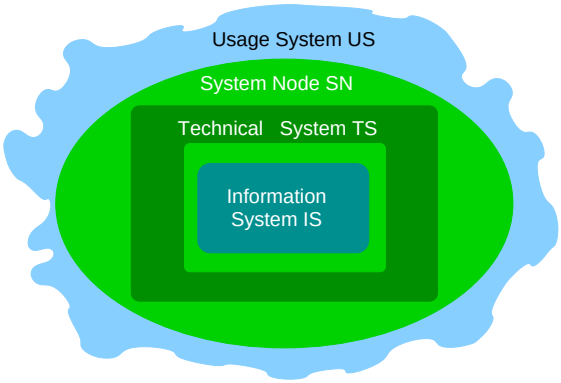

Abbildung 2.2: Modell eines Systemknotens

Für eine ungefähre Vorstellung, was der Inhalt dieser drei Systeme ist, hier eine Beschreibung:

Usage System Ein Computersystem wird immer zu einem bestimmten Zweck eingesetzt. Dadurch ergibt sich ein Nutzungszusammenhang, der ein soziotechnisches System bildet, in dessen Rahmen ein SN eingesetzt wird. Es umfasst den Ausschnitt der realen Welt, der für dieses System relevant ist, d.h. es wird von den Dingen, Personen, Zuständen, Vorgängen, etc. abstrahiert, die für den SN nicht von Bedeutung sind. Es ist schwierig, wenn nicht sogar unmöglich eine scharfe Grenze für dieses System anzugeben. Dies gilt bereits für einen bestimmten Zeitpunkt, jedoch ganz besonders für einen längeren Zeitabschnitt. Primär handelt es sich dabei um die unterstützten, realen Prozesse. Zudem kann es jedoch auch relevante Einflussfaktoren enthalten, wie folgende Beispiele illustrieren:

- Kultur, Sprache, Schrift, regionale Besonderheiten
- Einsatzgebiet, Fehlertoleranz, Umgebungsbedingungen
- zugehörige Abläufe in der realen Welt
- Nutzereigenschaften: Wissen, Erfahrung, Fähigkeiten, Vorlieben

Information System Dieses System umfasst alle Datenobjekte, die auf einem Systemknoten abgelegt sind. Insbesondere schließt dies alle Softwarekomponenten ein, die Anweisungen beinhalten, was das technische System machen soll. Zudem sind hier die Nutzerdaten und Subsysteme für das Management des Systemknotens enthalten: Nutzer, Komponenten, Konfiguration, Zugriffsrechte, etc.

Technical System Es umfasst alle Hardwareelemente eines Computersystems inklusive der enthaltenen Firmware. Eine besondere Bedeutung haben ihre Schnittstellen zum Nutzungssystem US (Ein-/Ausgabe) und zum Informationssystem IS. Die Hardware umfasst zwar bereits das gesamte Informationssystem, wenn man es genau betrachtet, da es die

physikalische Grundlage darstellt und die Informationen nur eine Zustandsbeschreibung sind, d.h. jede Änderung des Informationssystems hat seine Entsprechung auf der physikalischen Ebene. Von diesen Zuständen und ihren Änderungen wird jedoch hier abstrahiert.

Ein weiteres wesentliches System eines Systemknotens ist das Entwicklungssystem (development system) DS. Es beinhaltet die Mechanismen und Abläufe, die der Entwicklung des Systemknotens dienen. Dies umfasst Erstellung, Verwaltung, Überarbeitung und Stilllegung des Knotens und seiner Bestandteile. Die Vorgänge im DS werden in Abschnitt 2.3 genauer beschrieben.

2.1.2 Zusammenhang der Systeme

Die Veränderung und Optimierung der Abläufe im Nutzungssystem US ist die primäre Motivation für die Einrichtung und (Weiter-)Entwicklung des Systemknotens SN mit den darin enthalten Subsystemen TS (technisches System) und IS (Informationssystem). Die Einrichtung und Entwicklung findet im und durch das Entwicklungssystem DS statt (s. Abs. 2.3).

Das TS ist das eigentlich agierende System eines Knotens. Es beinhaltet alle seine physikalischen Elemente und stellt daher auch die einzige Möglichkeit der Interaktion mit anderen Elementen des US dar. Obwohl das IS natürlich eine physikalische Basis hat und jede Veränderung im IS eine äquivalente Änderung des TS nach sich zieht, werden hier derartige Zustandsänderungen ausschließlich im IS betrachtet und im TS werden die entsprechenden Objekte als zustandslos angesehen. Das IS stellt sozusagen eine Ausgliederung dieses Bereichs aus dem TS aufgrund seiner zentralen Rolle dar.

Das wesentliche eines SN ist damit das Verhalten seines TS, was den Handlungsanweisungen im IS folgt bzw. dort Daten ablegt, verarbeitet oder abruft. Das Verhalten kann somit nach Außen (Interaktion TS <-> US) oder nach Innen (Interaktion TS <-> IS) gerichtet sein.

Die Handlungsanweisungen, die vom TS zur Bestimmung seines Verhaltens im US genutzt werden, müssen in einer Form im IS abgelegt werden, die das TS verstehen kann. Dabei kann dieses “Verstehen” auch über mehrere Ebenen ablaufen, d.h. es können Anweisungen abgelegt sein, wie andere Anweisungen verarbeitet werden müssen, die wiederum dazu dienen andere Daten zu verarbeiten, usw. Diese Sammlungen von Anweisungen können als Prozessmodelle verstanden werden, da sie die Rolle des TS in Abläufen innerhalb des US exakt festlegen, die wiederum als Prozesse aufgefasst werden können. In diesen Prozessmodellen wurde von allen Details abstrahiert, die für das Verhalten des TS irrelevant sind. Sie umfassen jedoch direkt oder indirekt alle einflussgebenden Objekte, die aus allen Bereichen stammen können: dem zugehörigen IS, TS, US oder auch anderen Systemknoten. Der Einfluss eines Objekts kann z.B. in seinem bloßen Vorhandensein, seinen Eigenschaften oder seinen im Prozessmodell festgelegten Interaktionen mit dem TS bestehen. Wie diese Einflüsse im einzelnen aussehen und in welcher Form sie sich auswirken ist wiederum im Prozessmodell festgelegt. Die Summe aller Objekte, die in allen Prozessmodellen eines Systemknotens eine Rolle spielen, kann als das Nutzungssystem angesehen werden.

Sowohl das IS wie auch TS sind normalerweise hauptsächlich aus generischen Komponenten zusammengesetzt, die nicht speziell für das betrachtetet Nutzungssystem sondern einen allgemeineren Anwendungsfall konzipiert wurden. Daher sind während ihrer Entwicklung nicht alle wesentlichen Umstände bekannt und die resultierenden Ergebnisse sind nicht ausreichend auf

den spezifischen Systemknoten und seine Umgebung angepasst. Diese Anpassung muss dann anschließend in Form einer Konfiguration oder einem ähnlichem Adaptionsprozess nachgeholt werden (vgl. Kapitel 3).

So muss z.B. die Software (Elemente des IS) stets an die gegenwärtige Hardware (Elemente des TS) angepasst sein, da sie als die einzige Schnittstelle zur Umgebung eine zentrale Einflussgröße für das erwartete Verhalten des Systemknotens darstellt. Je nachdem wie anpassungsfähig die jeweilige Software gestaltet ist, kann dies im Vorfeld oder auch erst zur Laufzeit stattfinden. Evtl. können auch Mechanismen vorhanden sein, welche (teilweise) die Hardware des Knotens automatisch erkennen und dann eine entsprechende Anpassung am IS eigenständig durchführen. Jedoch ist gegenwärtig oft noch eine zusätzliche, manuelle Konfiguration nötig.

Die Subsysteme eines Knotens stehen in einer Wechselbeziehung zueinander. Daher ist es notwendig, dass ein gewisse Kompatibilität herrscht, damit die Interaktionen zwischen ihnen zu den gewünschten Ergebnissen führt. Gibt es Veränderungen in dem einen System, hat dies eine direkte oder indirekte Wirkung auf die anderen Systeme. Der Effekt einer Veränderung kann vernachlässigbar sein, das Zusammenspiel der Systeme fördern, stören oder im Extremfall unmöglich machen. Ist es gestört, müssen entsprechende Anpassungsvorgänge stattfinden. Welches System im Zweifelsfall angepasst und welches als unveränderlich angesehen wird ist von verschiedenen Faktoren abhängig und die Frage kann entsprechend den Umständen für verschiedene Knoten ganz unterschiedlich beantwortet werden. Zudem ist das gezielte Durchführen von entsprechenden adaptiven Eingriffen aufgrund der komplexen Wechselwirkungen schwierig und zieht oftmals ungewollte Nebenwirkungen nach sich, die dann wiederum korrigiert werden müssen.

2.1.3 Zusammenhang der Prozesse

Computersysteme werden zur Unterstützung von Prozessen in ihrem Nutzungssystem US verwendet. Dies kann z.B. das alphabetische Sortieren der Teilnehmerliste einer Lehrveranstaltung sein. Die Namen und ihre gegenwärtige Reihenfolge sind Teil vom US. Um die Namen mit Hilfe des Systemknotens SN zu sortieren, müssen sie zunächst aus dem US in den SN transferiert, dort bearbeitet und anschließend wieder in das US übertragen werden. Dieser Vorgang stellt den Prozess P dar, der dabei auf den technischen Prozess Pt zurückgreift, der dann ebenfalls Eingabe, Bearbeitung und Ausgabe umfasst. Der relevante Kontext für diesen Prozess wird hier als K bezeichnet und schließt auch den entsprechenden Knoten ein (bestehend aus TS und IS). Pt findet in dessen technischem System TS statt und umfasst alle Abläufe im TS, die zur Unterstützung von P dienen. Der relevante Kontext von Pt wurde hier mit Kt bezeichnet und umfasst auch alle relevanten Bereiche des Informationssystems IS und des US, insbesondere auch P und Software S. Das TS wird dabei einerseits durch die Interaktion mit dem US (im Rahmen von P) gesteuert und anderseits durch die zugehörige Software S, die im IS abgelegt ist. Der relevante Kontext von S wird hier als Ks bezeichnet und deckt alle relevanten Bereiche des IS, TS und US für S ab. (s. Abb. 2.3).

Alle diese Elemente stehen in einer Wechselbeziehung zueinander. Die Prozesse und die Software beeinflussen sich untereinander ebenso, wie sie ihre jeweiligen *relevanten* Kontexte beeinflussen. Gibt es Änderungen an den Prozessen bzw. der Software muss man damit rechnen, dass andere Teile der Umgebung relevant sind als vorher. Genauso können Änderungen im Kontext Anpassungen der Prozesse bzw. der Software erforderlich machen.

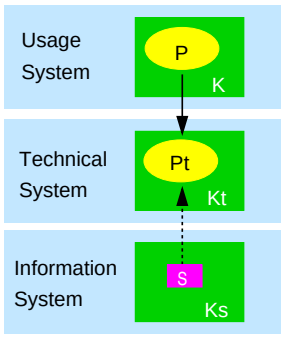

Abbildung 2.3: Zusammenhang der Subsysteme eines Systemknotens

2.2 Komponenten als Bausteine der Entwicklung

2.2.1 Komponenten als zentrale Struktureinheit

Die Bestandteile eines Systemknotens können in vielen unterschiedlichen Formen aufgeteilt und zerlegt werden. Atomare Bestandteile eines Systems, deren Zerlegung nicht weiter sinnvoll erscheint, werden oft als Objekte bezeichnet (z.B. Dateien). I.d.R. besteht jedoch zwischen mehreren von diesen einzelnen Objekten ein enger Zusammenhang, so dass man diese Gruppe im überwiegenden Teil der Zusammenhänge als eine Einheit betrachten kann. Eine solche Gruppierung von Objekten wird hier als Komponente bezeichnet. Zur Vereinfachung der administrativen Aufgaben wird dabei oftmals von den Einzelobjekten völlig abstrahiert und ein Knoten ausschließlich auf der Komponentenebene betrachtet.

Zur Vermeidung von Missverständnissen wird im Folgenden präzisiert, was im Zusammenhang mit einem Systemknoten als Komponenten verstanden wird:

1. Alle ihre Objekte gehören entweder dem TS oder dem IS an. Es gibt keine gemischten Gruppen. Damit gibt es Hardware- (TS) und Softwarekomponenten (IS).
2. Sie haben einen Namen und eine Version, die sie eindeutig identifiziert.
3. Sie werden nicht verändert, sondern durch eine modifizierte Version ersetzt.
4. Jede Komponente erfüllt eine bestimmte Funktion innerhalb des Systemknotens, d.h. i.d.R. handelt es sich bei der Gruppierung um eine funktionale Abgrenzung.

Softwarekomponenten haben weitere Charakteristiken, die hier von Bedeutung sind:

1. Sie bezeichnen ausschließlich Objekte aus dem Bereich der Systemdaten (vgl. Abs. 2.2.3).
2. Sie sind Teil des Gestaltungsbereichs des SoftAdmins und nicht der Benutzer (vgl. Abs. 2.3.1).
3. Ihre Anpassung auf die konkrete Umgebung wird als Konfiguration bezeichnet, die evtl. bei einer Veränderung der relevanten Bereiche wiederholt werden muss.

Komponenten können in diesem Sinne als Mittel zur Strukturbildung auf einem Systemknoten verstanden werden. Auf diese Weise kann einerseits eine gewisse Übersicht geschaffen und

andererseits die administrativen Arbeiten vereinfacht und standardisiert werden. Die daraus entstehenden Möglichkeiten und Probleme für Softwaresysteme werden in Kapitel 3 erläutert.

Da sich diese Arbeit mit der Entwicklung von Softwaresystemen befasst, werden im Weiteren primär die Komponenten des Informationssystems betrachtet und die des technischen Systems vernachlässigt.

2.2.2 Zerlegung des Informationssystems in Komponenten

In Abschnitt 2.1.3 wurde die verwendete Software S als eine monolithische Einheit dargestellt, die ausschließlich zur Steuerung dieses technischen Prozesses Pt dient und keine weiteren Prozesse unterstützt. Oftmals setzt sich jedoch S aus mehreren Komponenten zusammen (vgl. Abb. 2.4).

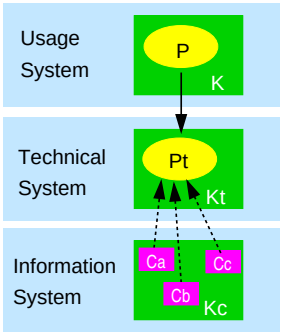

Abbildung 2.4: Verwendung von Softwarekomponenten in einem Systemknoten

Dies hat jedoch zunächst keinen direkten Auswirkungen auf das TS oder US, da man alle Komponenten letztlich auch als ein großes Softwareelement ansehen kann, in dem man vom internen Aufbau der Software abstrahiert. Der primäre Unterschied tritt bei den Komponenten selbst auf. Eine strukturelle Zerlegung in mehrere Elemente erfordert zunächst zusätzliche Arbeiten (Schnittstellen, Sicherstellung der Kompatibilität, etc.), die durch die angestrebten Vorteile eines Komponentensystems ausgeglichen werden sollen, die in Kapitel 3 näher beschrieben werden.

Ein wesentlicher Vorteil bei der Verwendung von Komponenten zeigt sich bereits, wenn wir den üblichen Einsatz eines Systemknotens für *mehrere* Prozesse berücksichtigen, statt wie bisher nur seine Unterstützung eines einzigen Prozesses zu betrachten (vgl. Abb. 2.5). Denn es ist bereits in diesem Fall wahrscheinlich, dass einige Komponenten für mehrere Prozesse verwendet werden können, in dem z.B. die teilweise Übereinstimmung der jeweiligen relevanten Umgebung K(x), Kt(x) und Kc ausgenutzt werden.

2.2.3 Charakterisierung der Komponenten des Informationssystems

Da Komponenten des Informationssystems nur eine Zusammenfassung von Objekten darstellt, bestehen sie ausschließlich aus Datenobjekten und lassen sich damit auch durch die nähere Betrachtung dieser Objekte charakterisieren.

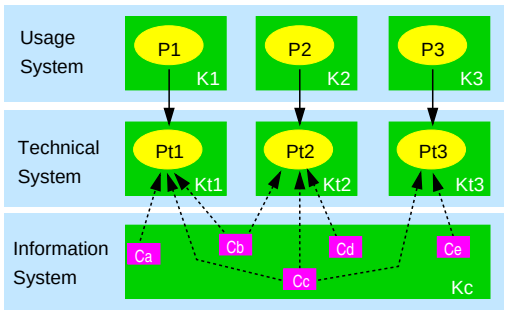

Abbildung 2.5: Nutzung eines Systemknotens für mehrere Prozesse

Im Informationssystems können sich grundsätzlich sowohl transiente (flüchtige) wie auch persistente (beständige) Datenobjekte befinden. Berücksichtigt man jedoch langfristige Nutzungsprozesse einerseits und die fortlaufenden Überarbeitung von “dauerhaften” Daten anderseits, relativieren sich diese Begriffe, wenn man sich dabei nur auf ihre Gültigkeitsdauer bezieht. Man kann jedoch Datenobjekte unterscheiden, deren Veränderung bei ihrer regulären Nutzung vorgesehen ist und solche bei denen dies nicht der Fall ist:

Systemdaten werden nicht durch eine reguläre Nutzung, sondern nur durch gesonderten administrativen Prozessen verändert. Sie stehen i.d.R. unter einer besonderen Kontrolle und werden oftmals durch entsprechende Mechanismen vor einer unkontrollierten Modifikation geschützt. Meistens werden sie in einer persistenten Form in TS-Objekten gespeichert, die sich dauerhaft im Knoten befindet (z.B. Dateisystem einer (internen) Festplatte).

Nutzungsdaten können während ihrer Nutzung erzeugt, gelöscht und verändert werden. Sie können entweder transient sein, d.h. sie gehen beim Beenden eines Nutzungsprozesses verloren, oder auch persistent, d.h. sie werden dauerhaft gespeichert und können damit von einem Nutzungsprozess in den nächsten übernommen werden (z.B. gespeicherte Dokumente, Spielstand, etc.). Oftmals gehören diese Daten zu bestimmten Applikationen oder Applikationsgruppen, die in vielen Fällen diese Daten dann auch selbst verwalten.

Systemdaten kann man wiederum in folgende drei Gruppen aufteilen:

Prozessdaten legen das Verhalten des Systemknotens SN in seinem Kontext fest. Die Daten müssen dabei in einem durch das technische System TS verarbeitbaren Format im Informationssystem IS abgelegten werden. Zum Erreichen eines bestimmten Verhaltens des Knotens müssen die entsprechenden Daten dort gespeichert sein. Um den Aufwand ihrer Erstellung auf ein bewältigbares Maß zu reduzieren, basieren sie auf der Gültigkeit eines bestimmten Kontextes, an den sie angepasst sind, d.h. der Systemknoten liefert nur dann ein gewünschtes Verhalten, wenn die in den Daten festgelegten Randbedingungen von der Umgebung eingehalten werden. Dies bezieht sich gleichermaßen auf das IS selbst, die technische Umgebung (primär das TS) wie auch die nicht-technische Umgebung (primär das Nutzungssystem US). Diese Randbedingungen sind derart umfangreich, dass man von einem in den Daten implizit enthalten Kontextmodell sprechen kann (s. Kap. 6). Steht die tatsächliche Umgebung in irgendeiner Weise im Widerspruch zu diesem Modell, führt dies zu einem unerwünschten Verhalten des Systemknotens.

Inhaltsdaten Auf einem System sind normalerweise zusätzliche Informationen gespeichert, die

durch das System selbst nicht genutzt werden, sondern lediglich Daten darstellen, die dem Nutzer zur Verfügung gestellt werden. Diese werden hier als Inhaltsdaten bezeichnet. Beispielsweise könnte dies ein Lexikon sein. Im weitesten Sinn wird dadurch zwar ebenfalls das Verhalten des Systemknotens beeinflusst, aber dies ist von einer anderen Natur als bei den Systemdaten. Es wird nur dem Nutzer der jeweilige Inhalt übermittelt. Im gegebenen Beispiel mit dem Lexikon führt die Änderung des Textes zu einer anderen Anzeige bei der Nutzung (nämlich der Anzeige eines anderen Textes), das Systemverhalten ist damit zwar beeinflusst, aber auf einer abstrakteren Ebene bleibt das Verhalten des Systems gleich: der Lexikoneintrag wird angezeigt.

Metadaten Es werden für installierte Softwareelemente im IS häufig noch Zusatzdaten hinterlegt, die zur Unterstützung der administrativen Vorgänge wie Entfernung, Aktualisierung, etc. dienen sollen. Dabei kann es sich z.B. um eine Liste der erzeugten Objekte, Kontaktdaten, Prüfsummen, etc. handeln.

Die beschriebenen Kategorien stellen eine grobe Strukturierung des IS dar und ermöglichen die Zuordnung von Datenobjekten zu unterschiedlichen Typen. Auf diese Weise kann man über Objekttypen auf einer abstrakten Ebene sprechen und ihnen Eigenschaften und Abläufe zuordnen, die für die Daten im Allgemeinen nicht zutreffend wären.

Wird übliche Software (Anwendungen, Programme, Applikationen, etc.) auf einem Systemknoten installiert, sollen sie während der Nutzung normalerweise nicht verändert werden. Es handelt sich dabei also (zum größten Teil) um Systemdaten.

Nutzungsdaten sind aufgrund ihrer permanenten Veränderung bei der Verwendung für eine übergeordnete Verwaltung nicht geeignet, daher wird im Folgenden davon ausgegangen, dass es sich bei den betrachteten Softwarekomponenten ausschließlich um Systemdaten handelt, die eine Mischung aus Prozess-, Inhalts- und Metadaten darstellen. Sie greifen dann evtl. auf Nutzungsdaten zurück, die hier aber nicht als Teil von ihnen aufgefasst werden.

2.2.4 Kontext von Komponenten

Die Komponenten in den drei Systemen IS, US und TS stehen in einem komplexen Beziehungsgeflecht zueinander. Schon allein die Komponenten innerhalb des IS haben vielfältige Verbindungen: Metadaten machen Aussagen über andere Daten, Nutzungs- und Inhaltsdaten benötigen passende Prozessdaten für ihre Verwendung und Prozessdaten greifen auf andere Prozessdaten und Inhalte zurück. Die Prozessdaten wiederum müssen an die entsprechenden Elemente des TS und US angepasst sein und auch die Beziehungen zwischen den Komponenten im TS und US sind nicht zu vernachlässigen.

Die Einhaltung und Berücksichtigung dieser Beziehungen ist für das ordnungsgemäße Funktionieren des Systemknotens unerlässlich. Eine Verletzung der damit verbundenen Regeln kann vielfältige Störungen verursachen, die bis zur Unbrauchbarkeit des Knotens reichen.

Aufgrund des damit verbunden Fehlerpotentials, ist es sinnvoll die Abhängigkeiten zwischen den Elementen möglichst zu reduzieren, sie explizit zu dokumentieren und sie in Bezug auf Systemveränderungen zu überwachen.

Nimmt man alle Elemente, mit denen eine Komponente in direkter Beziehung steht, so kann man diese Menge von Komponenten als seine Umgebung oder auch seinen Kontext bezeichnen, wie es in Abbildung 2.6 (links) dargestellt wurde.

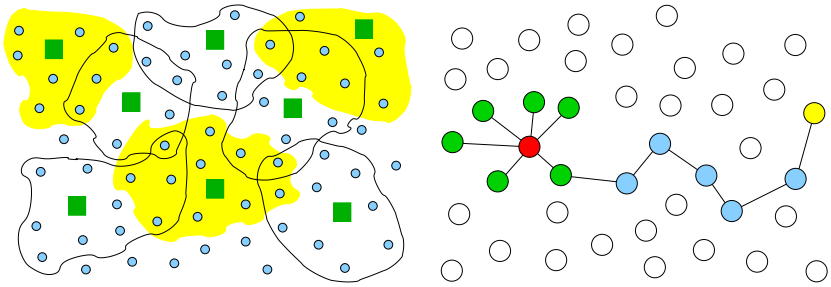

Abbildung 2.6: Systemelemente und ihre Umgebung aus direkten (links) und indirekten Nachbarelementen (rechts)

Die fokussierten Elemente (grün), stehen in Beziehung zu bestimmten Nachbarelementen (blau), die ihre Umgebung bilden (gelb). Da die Elemente dieses Kontexts wiederum selbst ihre eigene Umgebung haben, mit denen sie verbunden sind, entstehen dadurch eine Vielzahl von indirekten Beziehungen mit zunehmender Entfernung. Abb. 2.6 (rechts) zeigt exemplarisch eine solche indirekte Relation. Die grünen Elemente stellen den direkten Kontext der roten Komponente dar. Das gelbe Element steht über sechs andere Komponenten (4xblau und 1xgrün) mit dem roten indirekt in Beziehung. Indirekte Relationen könnten nun jedoch theoretisch auch als eine direkte Beziehung angesehen werden, da das jeweilige Element zu dem betrachteten Element in einer Relation steht. Es ist daher eine Frage der Auffassung, wie man die Zusammenhänge im Diskursbereich interpretiert und sie bei einer expliziten Erfassung jeweils als direkte, indirekte oder vernachlässigbare Beziehung deklariert.

Diese Problematik ist jedoch ein generelles Problem, dass sich aus der notwendigen Interpretation des Diskursbereichs ergibt. Versucht man den Zusammenhang zwischen den unterschiedlichen Komponenten in irgendeiner Form zu dokumentieren, muss man zunächst klar definieren, was man unter den jeweiligen Beziehungen versteht und welche Bedeutung sie haben. Diese Definitionen wiederum müssen für die späteren Verwendung der zu erstellenden Dokumentation geeignet sein. So wie Objekte Modelle von Entitäten sind, sind Relationen Modelle von Zusammenhängen, die der Betrachter bei dem entsprechenden Blickwinkel als relevant erachtet.

In Abhängigkeit von der jeweiligen Betrachtungsweise können daher ganz unterschiedliche Darstellungen des Kontexts einer Komponente entstehen, die andere Komponenten, Aspekte, Relationstypen und Relationen enthalten.

Im Folgenden werden diese Darstellungen eines Komponentenkontexts als Kontextmodelle bezeichnet und in Kapitel 6 näher erläutert. Dabei ist stets zu berücksichtigen, dass es sich um eine zweckorientierte Interpretation des Diskursbereichs handelt und nicht um eine vollständige Erfassung der tatsächlichen Umstände.

2.3 Entwicklung eines Systemknotens

Unter der Entwicklung eines Systemknotens wird hier die Erstellung, Verwaltung, Überarbeitung und Stilllegung des Knotens und seiner Bestandteile verstanden. Das System, in dem diese Vorgänge stattfinden, wird hier als das Entwicklungssystem (DS) eines Knotens bezeichnet. Die damit verbunden Aktivitäten werden auch als Administration bezeichnet.

Der Sinn und Zweck eines Systemknotens liegt in seiner Funktion für das Nutzungssystem US. Die Änderung oder Wiederherstellung dieser Funktionalität ist daher die zentrale Motivation für die Entwicklungsarbeiten an einem Systemknoten, die im Rahmen des DS geleistet werden. Ein weiterer wesentlicher Beweggrund ist die Erleichterung von administrativen Arbeiten, die innerhalb des DS selbst stattfinden. In beiden Fällen geht es um Veränderungen, die dem Erreichen eines bestimmten Zieles dienen, z.B. Steigerung der Effizienz eines Prozesses im US oder eine strukturelle Anpassung der internen Komponenten des IS. Aufgrund der Wechselwirkungen, muss man bei einer solchen Änderung in einem System stets auch die Auswirkungen auf die anderen drei berücksichtigen.

2.3.1 Rollen, Aktivitäten und Bereiche eines Systemknotens

Einerseits wird bei der Entwicklung eines Systemknotens dieser teilweise selbst als Hilfsmittel verwendet und stellt damit auch eine Form der Nutzung dar. Andererseits verändert auch die reguläre Nutzung einen Systemknoten, in dem z.B. Daten abgelegt werden, die das Systemverhalten anpassen. Daher kann auch die Nutzung gewissermaßen als Entwicklung angesehen werden. Entwicklung und Nutzung lassen sich deswegen nicht so leicht voneinander trennen.

Man kann jedoch verschiedene Formen der Entwicklungsarbeit differenzieren, die jeweils unterschiedliche Bereiche betreffen. Daraus ergeben sich drei Kategorien von Aktivitäten, die im Zusammenhang mit einem Systemknoten stehen, aus denen dann Rollen mit den zugehörigen Gestaltungs- und Ausführungsbereiche abgeleitet werden können.

Man kann die Aktivitäten der Akteure entsprechend in drei Kategorien aufteilen:

Administrative Vorbereitung umfasst alle Vorgänge, die dazu dienen den Systemknoten in einen funktionsfähigen Zustand zu bringen und zu halten, der seine eigentliche Nutzung ermöglicht. Damit ist die Einrichtung und Wartung des technischen Systems TS und eines initialen Informationssystems IS gemeint. Dies bildet zusammen ein Basissystem, was den rudimentären Einsatz des Systemknotens als funktionale Einheit ermöglicht. Die Hauptaufgabe dieser Aktivitätskategorie ist die Instandhaltung des TS.

Administrative Nutzung Ist das Basissystem durch die Vorbereitung erstmal eingerichtet, so kann ein Großteil der administrativen Vorgänge mit der Unterstützung des Systemknotens selbst geschehen. Für die reguläre Nutzung müssen noch im IS die entsprechenden Prozess- und Inhaltsdaten hinterlegt werden, die das Verhalten des spezifischen TS in dem zugehörigen US vorgeben. Diese Prozesse finden im Rahmen des Entwicklungssystems DS statt, das sich jedoch mit dem Nutzungssystem US überschneidet, soweit sie mit einer (administrativen) Verwendung des Systemknotens verbunden sind. Die Aufgabe dieser Aktivitätskategorie ist die Instandhaltung des IS, soweit es benutzerunabhängig ist.

Reguläre Nutzung ist der Einsatz des Systemknotens zur Unterstützung der fokussierten Prozesse im Nutzungssystem US und der benutzerspezifischen Anpassung des IS.

Basierend auf diesen drei Aktivitätskategorien lassen sich drei passende Rollen definieren:

NodeAdmin (administrative Vorbereitung) Er benötigt physikalischem Zugriff auf den Systemknoten, um das TS einzurichten und zu warten. Evtl. reguliert er auch den physikalische

Zugriff auf den Systemknoten durch andere Akteure. Er hat damit die Möglichkeit ein passendes TS zusammenzustellen bzw. zu beschaffen, darauf das Basissystem zu installieren und durch Wartungsarbeiten die entsprechende Grundfunktionalität sicherzustellen.

SoftAdmin (administrative Nutzung) Er benötigt für das IS administrative Rechte. Er hat keine Möglichkeit das TS zu verändern, kann aber entsprechende Informationen darüber erhalten, um die Daten im IS anzupassen. Diese Informationen können ihm einerseits vom TS selbst über eine entsprechende Schnittstelle zur Verfügung gestellt werden oder er kann sie vom NodeAdmin erfahren. Der SoftAdmin hat uneingeschränkte Kontrolle über den größten Teil des IS. Evtl. kann er jedoch nicht das Basissystem (inkl. der enthalten Systemdaten im IS) und damit bestimmte Teile seiner eigenen Ausführungsumgebung verändern. Der SoftAdmin reguliert durch die Hinterlegung von entsprechenden Konfigurationsdaten für das TS den virtuellen Zugriff der Benutzer auf das technische System (von Innen nach Außen) und die darin enthalten Schnittstellen zur Außenwelt (Drucker, Laufwerke, Steuerungsgeräte). Auf die selbe Weise legt er die Befugnisse des Benutzers im IS fest und verteilt die vorhanden Ressourcen. Zudem ist er dafür verantwortlich passende Systemdaten im IS zu hinterlegen, so dass die Prozesse der Nutzer möglichst effizient unterstützt werden.

Benutzer (reguläre Nutzung) Ein Nutzer verwendet den bereitgestellten Systemknoten, um Prozesse durchzuführen bzw. zu unterstützen, für die der Knoten eine passenden Funktionalität bereitstellt.

NodeAdmin und SoftAdmin werden im Folgenden Administratoren genannt. Obwohl es auch denkbar ist, dass die Administrator-Rollen von mehreren Akteuren in einem Systemknoten ausgefüllt werden, gibt es keine systemimmanenten Gründe dafür. Es geht bei diesen Rollen primär um die Erfüllung der entsprechenden Aufgaben bzw. Funktionen. Daher wird hier davon ausgegangen, dass es nur jeweils eine Instanz von ihnen gibt.

Anders sieht es bei den Nutzern aus: Ein Systemknoten wird oftmals von mehreren Nutzern unabhängig voneinander verwendet, was mehrere unabhängige Instanzen von dieser Rolle erfordert.

Jede Instanz einer Rolle hat ihren Gestaltungsbereich, in dem sie die Möglichkeit hat, die darin befindlichen Objekte zu manipulieren (vgl. Abb. 2.7 links) :

NodeAdmin Space Dieser Bereich ist identisch mit dem Systemknoten, der ja aus dem TS und IS besteht. Dies umfasst auch die Bereiche der anderen Rollen, jedoch ist es denkbar, sinnvoll und üblich, dass ein NodeAdmin seine Zugriffsmöglichkeiten auf diese Bereiche im Normalfall nicht nutzt, sondern nur in Ausnahmesituationen darauf zugreift und ansonsten seine Tätigkeit ausschließlich auf Bereiche beschränkt, die von anderen Rollen nicht abgedeckt werden (TS und evtl. Basissystem im IS).

SoftAdmin Space Dieser Bereich ist identisch mit dem IS (evtl. abgesehen vom Basissystem). Er umfasst auch die Bereiche der Benutzer, jedoch ist es ebenfalls denkbar, sinnvoll und üblich, dass ein SoftAdmin auf die Benutzer-Bereiche im Normalfall nicht zugreift, sondern nur in Ausnahmesituationen von dieser Möglichkeit Gebrauch macht.

User Spaces Es handelt sich dabei um abgegrenzte Bereiche des IS, die dem jeweiligen Nutzer zugänglich gemacht werden, wenn ihn das System als solchen identifizieren kann. Hier legt ein Nutzer seine spezifischen Daten ab, die im Zusammenhang mit den jeweiligen Prozessen anfallen. I.d.R. handelt es sich daher um Nutzungsdaten.

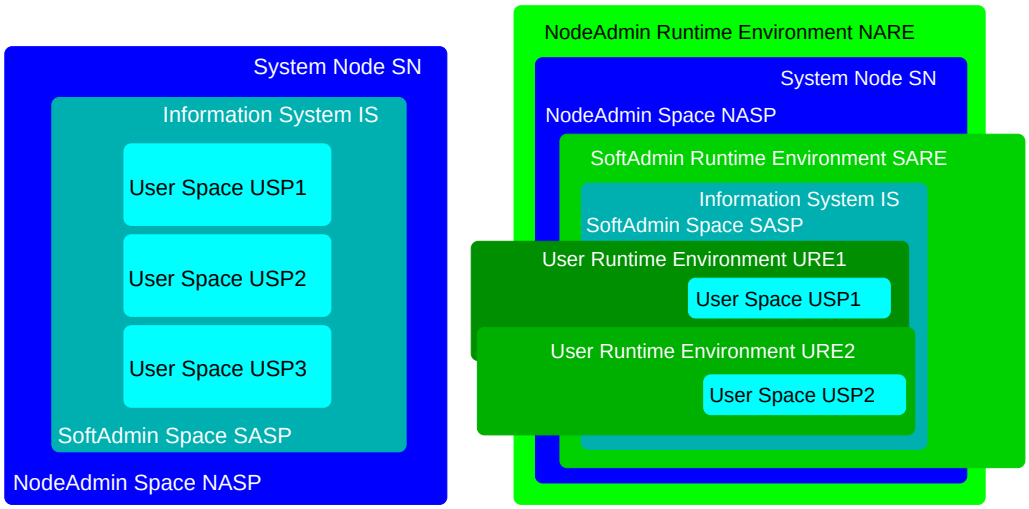

Abbildung 2.7: Einflussbereiche (links) und Ausführungsumgebungen (rechts) der drei wesentlichen Rollen innerhalb eines Systemknotens

Die Ausführungsumgebungen der Akteure ist eine zweite wesentliche Kategorie von Bereichen. Sie umschreibt hier den Teil eines Systemknotens, der an der Interaktion mit dem jeweiligen Akteur beteiligt ist bzw. dabei das Verhalten des Systems beeinflussen könnte.

Dabei ist zu beachten, dass SoftAdmin und Benutzer nur ein Teil dieses Systemverhaltens selbst bestimmen können, d.h. es wird ihnen teilweise von einer anderen Rolle (Node- bzw. SoftAdmin) vorgegeben. Konkret handelt es sich bei diesen Vorgaben um relevante Systemelemente, die sie nicht beeinflussen können. Dies können sowohl TS- wie auch IS-Objekte sein.

Das Systemverhalten wird jedoch auch durch externe Objekte im US beeinflusst, was insbesondere Objekte auf vernetzten Systemknoten einschließt. Aus diesem Grund reicht die Ausführungsumgebung bei allen Rollen über den Systemknoten selbst hinaus. Da die Gestaltungsbereiche der Admin-Rollen jedoch auf diesen begrenzt sind, stehen die Ausführungsumgebungen der Benutzer (Node- und SoftAdmin) bzw. des SoftAdmin (NodeAdmin) nicht unter ihrer vollständigen Kontrolle (vgl. Abb. 2.7 rechts).

Die Entwicklung eines Systemknotens, findet also primär durch die Administratoren statt, jedoch nicht ausschließlich. Auch die Benutzer können den Ablauf ihrer Prozesse durch das hinterlegen entsprechender Informationen gestalten.

2.3.2 Charakterisierung der Entwicklungsarbeit

Für eine gezielte Veränderung eines Systemknotens stehen folgende Gestaltungsmöglichkeiten zur Verfügung:

Manipulation von Komponenten Im TS oder IS können neue Komponenten hinzugefügt, unpassende entfernt und überarbeitete ausgetauscht werden.

(Weiter-)Entwicklung von Komponenten Zur Veränderung des TS und IS werden Komponenten aus einem Hardware- bzw. Softwarepool verwendet, die im Rahmen der Entwicklungstätigkeit auch überarbeitet oder neu entwickelt werden können. Aufgrund der Kon-

zentration auf die Softwareentwicklung in dieser Arbeit werden entsprechende Aktivitäten für den Hardwarepool hier nicht weiter betrachtet.

Konfiguration von IS-Komponenten Wann immer notwendig, können die lokalen Einstellungen für eine Komponente des Informationssystem geändert werden.

Etablierung von neuem Prozessverlauf Im Zusammenhang mit Änderungen des Systemknotens oder auch zur Verbesserung der Interaktion mit ihm ist eine Anpassung der Verhaltensweisen der Akteure und anderer Vorgänge des US im Rahmen der fokussierten Prozesse eine weitere Möglichkeit.

Dementsprechend wird Entwicklungsarbeit hier verstanden als eine Reihe von Entwicklungsschritten bzw. Prozessen Pd mit Kontext Kd im Rahmen des Entwicklungssystems DS, welche die betrachteten drei Systeme (US, TS und IS), den zugehörigen Softwarepool SP und die jeweiligen Elemente von einem gegenwärtigen Zustand zum Zeitpunkt t in einen neuen Zustand zum Zeitpunkt t' überführen (vgl. Abb. 2.8).

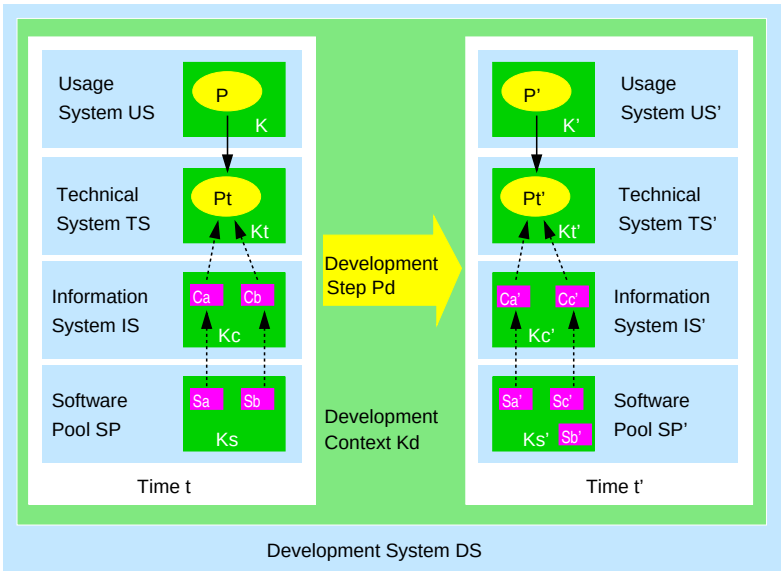

Abbildung 2.8: Entwicklungsschritt mit Kontext im Entwicklungssystem

Hier wird zudem davon ausgegangen, dass eine Komponente C des IS nicht dort entwickelt wird, sondern aus einer Software S des Softwarepools SP in einem Installationsprozess hervorgeht. Dieser Prozess ist Teil der administrativen Nutzung des SN und findet im Rahmen des DS als Bestandteil eines Entwicklungsschritts Pd statt.

2.3.3 Strukturierung der Entwicklung

Die gezielten Entwicklungsarbeiten an einem Systemknoten im Rahmen des Entwicklungssystems DS, (Entwicklungsschritt Pd) lassen sich entsprechend den Ausführungen in Abschnitt 4.2 in vier Phasen einteilen: Analyse, Konzeption, Umsetzung und Ausgabe. Es gibt verschiedene Varianten von einer solchen Entwicklung in Abhängigkeit des primären Ziels:

2.3.3.1 Entwicklung des Nutzungsprozesses P

Dient das Entwicklungsvorhaben einer Veränderung der Abläufe im US, so ist der Ausgangspunkt der Nutzungsprozess P:

Analyse Erfassung des gegenwärtigen Prozesses P und seinem relevanten Kontext K im US.

Konzeption Entwurf des zukünftigen Prozesses P' und seinem Kontext K' im zukünftigen US' entsprechend den auslösenden Anforderungen.

Umsetzung Entwicklung des zugehörigen, zukünftigen, technischen Prozesses Pt' und der dafür nötigen Veränderungen im gegenwärtigen Systemknoten SN.

Ausgabe Etablierung von neuem Prozessverlauf P' im US' und Aktivierung der Änderungen im SN'.

Ein wesentlicher Teil der Arbeiten im Rahmen des DS dient der Verbesserung zukünftiger Entwicklungsprozesse selbst. Dies umfasst teilweise auch zugehörige Anpassungen des SN. Es kann sich daher bei dem fokussierten Prozess P sowohl um eine reguläre wie auch administrative Nutzung handeln (vgl. Abs. 2.3.1).

2.3.3.2 Entwicklung des technischen Prozesses Pt

Soll der technische Prozess Pt und sein Kontext Kt angepasst werden, wie dies z.B. im Rahmen der Durchführung der vorhergehenden Entwicklung von P oder bei einer Fehlerkorrektur im SN der Fall ist, können die dabei anfallenden Arbeiten am SN ebenfalls in die besagten vier Phasen zerlegt werden:

Analyse Erfassung des gegenwärtigen, technischen Prozesses Pt und des zugehörigen Kontexts Kt im TS und IS.

Konzeption Entwurf des zukünftigen Pt' und des zugehörigen Kt' im TS' und IS' basierend auf den Vorgaben durch den angestrebten P' und seinem K'.

Umsetzung Sammlung von verfügbaren, passenden Komponenten für die vorgesehene Anpassung im TS und IS; Entwicklung von fehlenden Komponenten; Identifikation und Vorbereitung von notwendigen Konfigurationsarbeiten an betroffenen IS-Komponenten (inkl. der neuen Komponenten)

Ausgabe Bereitstellung von passenden TS- und IS-Komponenten

2.3.3.3 Entwicklung von Softwarekomponenten

Einerseits müssen neue Softwarekomponenten erstellt und anderseits bereits verwendet Elemente überarbeitet werden. Die Motivation dafür kann eine angestrebten Änderung des Systemverhaltens in Bezug auf einen technischen Prozess Pt sein oder auch eine strukturelle bzw. inhaltliche Überarbeitung, die auf das Systemverhalten keine Auswirkungen hat und stattdessen andere Vorgänge im DS erleichtert oder verbessert. Zu der letzten Kategorie gehören z.B. Aufräumarbeiten oder Umstrukturierungen innerhalb des Quelltextes, welche eine bessere Übersicht schaffen und damit die Entwicklungsarbeit unterstützen.

Analyse Erfassung der gegenwärtigen, integrierten Komponente C und ihrem relevanten Kontext Kc im IS. Handelt es sich um eine neue Komponente, so ist die Erfassung auf Kc beschränkt.

Konzeption Entwurf der zukünftigen Komponente C' und ihres Kontexts Kc' im IS' entsprechend der angestrebten Zielsetzung.

Umsetzung Soll eine vorhandene Komponente C angepasst werden, umfasst dies zunächst die Überarbeitung der Software S, aus der C generiert wurde, und ihres Kontexts Ks. Das Ergebnis ist ein modifiziertes S' und Ks'. Bei einer Neuerstellung wird ein neues S' und Ks' entwickelt. In beiden Fällen muss ein passender Installationsprozess vorbereitet werden, der dann zur Generierung der neuen Komponente C' im IS' dient.

Ausgabe Bereitstellung der passenden Software S'.

2.3.4 Mehrfachverwendung von Softwarekomponenten

Die im letzten Abschnitt erörterten Vorgänge beschränken sich ausschließlich auf einen isolieren Systemknoten und seine Umgebung. Die Entwicklung findet damit für jeden Knoten einzeln und unabhängig von anderen Knoten statt, d.h. bei x Systemen, die ähnliche Nutzungsprozesse unterstützen, würde es keinerlei Synergieeffekte geben und es wäre der x-fache Aufwand für die Entwicklung notwendig (vgl. Abb. 2.3).

Aus einer übergeordneten Perspektive liegt daher bei einer größeren Anzahl von ähnlichen Systemknoten der Ansatz nahe, die Gemeinsamkeiten so weit wie möglich zu nutzen und damit den Gesamtaufwand zu reduzieren. Vom Standpunkt der jeweiligen Verantwortlichen für die einzelnen Knoten aus, ergibt sich durch ein geeignetes kooperatives Vorgehen die Möglichkeit zur Effizienzsteigerung und Kostensenkung.

Dies erreicht man u.a. durch das grundlegende Prinzip der Wiederverwendung. Man kann versuchen, erstellte Softwarekomponenten für mehrere Systemknoten zu verwenden. In diesem Fall gibt es nicht mehr für jeden Systemknoten einen eigenen Softwarepool, sondern nur noch einen gemeinsamen (vgl. Abb. 2.9 rechts).

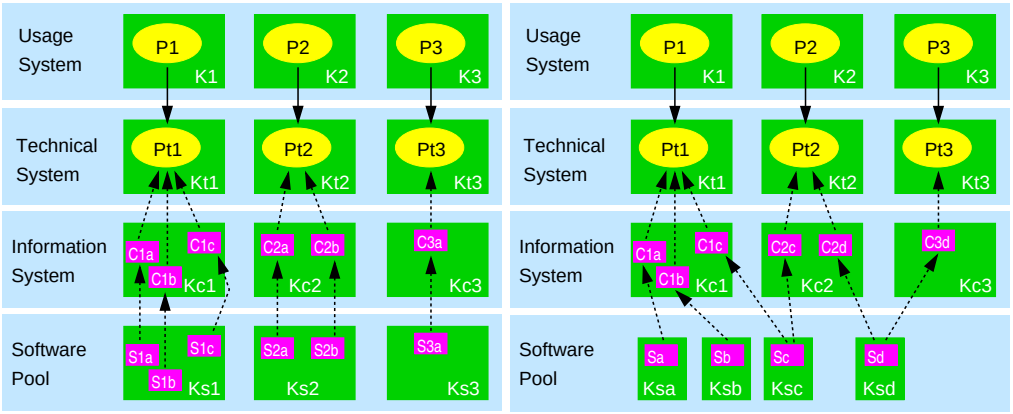

Abbildung 2.9: Systementwicklung mit knotenspezifischen (links) und generischen Komponenten (rechts)

Allerdings ergeben sich daraus auch Nachteile. So ist es dann z.B. aufgrund des breiteren Einsatzgebietes schwierig den relevanten Kontext derartiger Softwareelemente zu erfassen, da es nicht mehr ausreicht, nur die konkreten Umstände eines bestimmten Systemknotens zu untersuchen, sondern viele verschiedene berücksichtigen werden müssen.

Es gibt eine Wechselwirkung zwischen den Prozessen P und Pt einerseits und den Komponenten C und den Softwareelementen S andererseits. Diese führt bei gemeinsam genutzter Software auch zu einer Angleichung der jeweiligen Prozesse. Eine Abweichung von den typischen Abläufen zieht einen Mehraufwand nach sich, in dem entsprechende individuelle Anpassungen notwendig werden, die dann zudem bei weiteren allgemeinen Überarbeitungen an dem Element selbst oder dem zugehörigen Kontext nicht unbedingt berücksichtigt werden. Dabei ist jedoch oftmals zunächst gar nicht klar, was dieser Standard eigentlich ist. Vielmehr bildet er sich als Ergebnis von einem komplexen Abstimmungsprozess erst heraus, in dem alle beteiligten Knoten, ihre Subsysteme und die zugehörigen Prozesse eine gewisse Rolle spielen. Zudem kann sich das Resultat im Laufe der Zeit wieder ändern. Dieser Abstimmungsprozess kann daher verstanden werden als die fortlaufende Modellierung eines verbindlichen Kontextmodells Ks für das jeweilige Element, dass jeweils die relevanten Bereiche aller beteiligten Systemen abdeckt und als gültiges Modell davon angesehen wird. Dies erschwert jedoch die vorausschauende Festlegung von klaren Vorgaben und damit auch die frühzeitige Formulierung der angestrebten Ziele erheblich. Vielmehr ergibt sich daraus eine schrittweises Annäherung an ein bewegliches Ziel. Das Kontextmodell stellt damit eine Augenblickaufnahme des aktuellen Ergebnisses dieses Abstimmungsprozesses dar.

Das genau Vorgehen sowie die daraus entstehenden Vor- und Nachteile der Wiederverwendung werden in Kapitel 3 näher behandelt. Das dabei zentrale Problem der Erfassung des Kontexts wird in Kapitel 6 erörtert.

3 Komponenten-Perspektive

Die Komponenten-Perspektive umfasst Strukturen und Vorgänge im Zusammenhang mit einer systematischen Wiederverwendung.

Im Mittelpunkt dieser Betrachtung stehen Softwarekomponenten als die wiederverwendeten Elemente, die bei der Erstellung von vielen unterschiedliche Softwaresysteme als Bausteine dienen. Auf diese Weise entstehen zwei getrennte Bereiche: die Komponenten- und die Systementwicklung. Sie werden durch das Komponentensystem miteinander verbunden, das ihre Interaktionen regelt.

3.1 Komponentenbasierte Systementwicklung

3.1.1 Systematischer Wiederverwendung

Die Wiederverwendung von Software ist ein grundlegendes Konzept der Informatik, das schon früh zum Einsatz kam. Die einfachste Form ist wohl das Kopieren von Quelltext aus einem Programm und das Einfügen in ein anderes, in dem an der entsprechenden Stelle ein ähnlicher Ablauf programmiert werden muss.

Krueger definiert Wiederverwendung als den Erstellungsprozess von Softwaresystemen aus existierender Software im Gegensatz zur vollständigen Neuentwicklung [Krueger 92, 11.3 General Conclusions]. Man kann dabei grundsätzlich die ad-hoc und die systematische Wiederverwendung unterscheiden. Im ersten Fall wird bestehende Software oder Teile davon einfach in ein neues Softwaresystem übernommen. Der ad-hoc Ansatz ist jedoch vom Einzelfall abhängig und kann in einer allgemeinen Form nur schwer erfasst werden.

Im Gegensatz dazu wird bei einem systematischen Ansatz die zukünftige Mehrfachverwendung bereits bei der Erstellung der Software berücksichtigt und/oder sie ist integriert in ein unterstützendes System zur Wiederverwendung (s. a. [IEEE1517 99]). Dieser Ansatz, in dem die wiederverwendeten Elemente auch als Komponenten bezeichnet werden, entstammt schon den Anfängen der Softwaretechnik. So beschrieb bereits 1968 McIlroy in seinem Aufsatz „Mass Produced Software Components" seine Vision von einer eigenen Industrie, die Softwarekomponenten entwickelt, die dann als Baukasten zur schnellen Erstellung von Softwaresystemen genutzt werden sollten [McIlroy 68]. Dies war der Versuch, die sogenannte "Softwarekrise" in den Griff zu bekommen, die sich auf die zunehmende Bürde und Frustration bezog, die mit der Entwicklung und Wartung von Software verbunden war bzw. ist. Man wollte die Methode einer industriellen Fertigung, wie sie bereits aus der Automobilindustrie bekannt war, auf den Softwarebereich übertragen. Doch auch dieser Ansatz brachte keinen Durchbruch. Obwohl seitdem viele Fortschritte gemacht und eine Menge hilfreicher Konzepte entwickelt wurden, hat man bis heute keine umfassende Lösung gefunden, die Softwareentwicklung allgemein in diesem Maße beherrschbar macht. Brooks stellte bereits 1987 angesichts der anhaltenden Schwierigkeiten

die These auf, dass dies in der Natur der Sache liegen und es keine solche magische “silberne Kugel” geben könnte, die diese Probleme schnell und umfassend löst [Brooks 87]. Trotzdem plädierte auch er in diesem Aufsatz für die Wiederverwendung, da gerade angesichts des hohen Entwicklungsaufwands der Rückgriff auf vorhandene Software vielversprechend ist.

Es ist unbestritten, dass Wiederverwendung ein hilfreiches Konzept ist und die heutige Softwareentwicklung wäre ohne sie nicht denkbar, aber eine Fließbandproduktion von neuer Software konnte damit bisher nicht erreicht werden. Trotzdem hat eine systematische Wiederverwendung wesentlichen Vorteile gegenüber unabhängigen Neuentwicklungen. Sie lassen sich auf die folgenden drei Punkte reduzieren (vgl. [Sametinger 97, S. 11]):

Qualität Da eine Komponente vielfach verwendet wird, sieht eine Kosten/Nutzen Rechnung anders aus und Optimierungen werden eher durchgeführt. Zudem werden Defekte schneller erkannt und die Zuverlässigkeit steigt. Durch eine entsprechende Verbreitung einer Komponente wird insgesamt die Kompatibilität der Systeme gefördert, da sie quasi einen De-facto-Standard darstellt.

Aufwand Durch die entsprechende Reduzierung des Code-Umfangs bei gleicher Funktionalität, ist weniger Aufwand für seine Erstellung und damit verbundene Tätigkeiten wie Testen, Dokumentation, Auslieferung, Wartung etc. notwendig. Dieser Effekt wird nochmals durch den Wegfall von dem zugehörigen Koordinationsaufwand verstärkt, der bei großen Projekten eine wesentliche Rolle spielt (vgl. [Brooks 95]). Zudem kann erworbenes Wissen ebenfalls wiederverwendet werden und erspart eine erneute Einarbeitung.

Prototyp Der Rückgriff auf vorhandene Komponenten ermöglicht es in kurzer Zeit relativ nahe an das Ziel zu kommen, in dem ein rudimentärer, lauffähiger Prototyp gebaut wird. Dies bringt wesentliche Vorteile, da die Erfahrung gezeigt hat, dass viele Probleme und Missverständnisse erst offenkundig werden, wenn man etwas zum Vorzeigen und Testen zur Verfügung hat.

Diesen Vorteilen stehen nicht unerhebliche Zusatzarbeiten gegenüber, die durch eine Wiederverwendung entstehen. Sowohl bei der Erstellung wie auch bei der Verwendung von Komponenten müssen Arbeiten erledigt werden, die bei einer Einzelentwicklung nicht notwendig wären. Zwar sollte dies im Normalfall durch die Reduzierung des Aufwandes mehr als ausgeglichen werden, aber dies ist nicht garantiert und abhängig von mehreren Faktoren (Art und Zahl der Nutzer, Aufbau ihres Softwaresystems, etc.). Zudem muss ein großer Teil der Zusatzarbeiten für die Wiederverwendung am Anfang geleistet werden und ist damit eine Investition in eine ungewisse Zukunft, die sich nicht unbedingt auszahlt. Daher ist die Minimierung dieser notwendigen Zusatzarbeiten der zentrale Erfolgsfaktor einer systematischen Wiederverwendung.

Geht man nun von der massenhaften Nutzung einer Komponente aus, ist die Minimierung des Aufwandes bei ihrer Verwendung ein vielversprechender Ansatz, da er dort jedes Mal anfällt und nicht nur einmal wie bei der Entwicklung.

Krueger benennt vier wesentliche Aktivitäten, die bei der Wiederverwendung solcher Komponenten eine Rolle spielen [Krueger 92, Introduction]: ihre Beschreibung (abstraction), Auswahl (selection), Anpassung (specialization) und Integration (integration). Sie sind aus einem Aufsatz von Biggerstaff und Richter abgeleitet, die folgende vier operationalen Probleme benennen, die ein Wiederverwendungssystem berücksichtigen muss [Biggerstaff 89]: das Auffinden (finding), Verstehen (understanding), Anpassen (modifying) und Zusammenfügen (composing) von Kom-

ponenten. Zusammenfassend ergeben sich daraus vier wesentliche Aufgaben auf der Nutzerseite für eine erfolgreiche Wiederverwendung, die hier interessant sind:

Auffinden Die bisher entwickelten Komponenten, die bereits in das Wiederverwendungssystem integriert wurden, müssen für einen potenziellen Nutzer zunächst sichtbar sein, damit er sich dann mit Hilfe von entsprechenden Informationen die passende auswählen kann.

Verstehen Die Komponente, ihre Funktionalität und ihr Verhältnis zu ihrer Umgebung sollte der Nutzer so weit verstehen, dass er sie auswählen, sachgerecht verändern und in das vorgesehene Softwaresystem ordnungsgemäß integrieren kann.

Anpassen Für eine sinnvolle Nutzung ist es oft erforderlich eine Komponente zumindest minimal an die lokalen Gegebenheiten anzupassen. Dies kann z.B. durch eine entsprechende Parametrisierung oder Veränderung des Codes selbst geschehen.

Integrieren Die betreffende Komponente muss in das Softwaresystem eingebaut werden, dabei muss insbesondere sichergestellt werden, dass die unterschiedlichen Teile des Softwaresystems zusammenarbeiten bzw. miteinander kompatibel sind.

Folgende drei Maßnahmen können einen Beitrag dazu leisten, den notwendigen Aufwand zur Bewältigung dieser Aufgaben zu minimieren:

Unterstützung Passende Werkzeuge können einfache Routineabläufe automatisieren und die übrigen notwendigen Arbeiten vereinfachen. Die meisten Wiederverwendungssysteme haben daher eine entsprechende Infrastruktur, die aus derartigen Hilfsmitteln besteht. Im Folgenden wird dies Komponenteninfrastruktur genannt.

Vorbereitung Damit zugehörige Werkzeuge möglichst viel Last von den Akteuren nehmen können, müssen Komponenten genormte Schnittstellen (Anpassung/Integration) und entsprechende Metadaten (Auffinden/Verstehen) in einem passenden Format bereitstellen. Eine systematische Wiederverwendung besitzt daher i.d.R. einen Standard, der eine verbindliche Beschaffenheit von Komponenten vorgibt. Dieser Standard wird im Folgenden Komponentenmodell genannt.

Wartung Der Aufwand für die besagten Aufgaben steht in Abhängigkeit mit den betreffenden Akteuren, ihren Fähigkeiten, Anforderungen, Prioritäten, ihrem Vorwissen und insbesondere auch ihren Softwaresystemen. Um so besser eine Komponente zu diesen Faktoren passt, um so geringer ist der notwendige Aufwand. Da jedoch eine Komponente in tausenden Softwaresystemen eingesetzt werden kann, ist die vollständige Berücksichtigung all dieser Faktoren und eine entsprechend gezielte Überarbeitung ein zu aufwendiger und komplexer Vorgang. Jedes Wiederverwendungssystem muss jedoch eine Methode bereitstellen, um diese Faktoren bei der (Weiter)-Entwicklung einer Komponente zu berücksichtigen. Im Folgenden wird ein derartiger Mechanismus Komponentenwartung genannt.

Eine systematische Wiederverwendung wird im Folgenden als Komponentensystem bezeichnet und mit Hilfe dieser drei Aspekte charakterisiert: Komponenteninfrastruktur, Komponentenmodell, Komponentenwartung.

3.1.2 Komponenten

Wenn man über Wiederverwendung spricht, werden die betreffenden Softwareelemente auch als Komponenten bezeichnet. Durchsucht man die Literatur, findet man ein ganze Reihe von

Definitionen, die diesen Begriff in unterschiedlicher Weise belegen (vgl. Liste von Definitionen in [Sametinger 97, S.70]). Alle betrachteten Formulierungen decken sich in der Charakterisierung einer Komponente als ein (potenzieller) Bestandteil eines größeren Ganzen. Diese recht vage Definition ist in vielen Zusammenhängen ausreichend. Jedoch ist diese Beschreibung auch ebenso passend für Softwareelemente im Allgemeinen. Council/Heineman sehen ebenfalls Schwierigkeiten in der Abgrenzung dieser beiden Begriffe und folgern, dass Komponenten sich nur abgrenzen lassen, durch die Art und Weise wie sie genutzt werden [Heineman 01, S. 7]. Dies fasst den Begriff bereits ein wenig enger und deckt sich mit der Auffassung, die in einer Komponente ein Element sieht, dass für eine Wiederverwendung genutzt wird bzw. dafür vorgesehen ist.

Sametinger grenzt den Begriff in seiner Definition weiter ein, in dem er zusätzlich Forderungen aufstellt, die ein Element erfüllen muss, bevor es als Komponente angesehen wird [Sametinger 97, S.68]:

> "Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status."

Die darin enthaltenen Bedingungen, die er im Anschluss erläutert, lassen sich wie folgt zusammenfassen:

Abgeschlossenheit (self-containedness) Die Komponente sollte eine gewisse Autarkie aufweisen und nicht auf weitere Elemente angewiesen sein. So sollte man eine vollständige Bibliothek als eine Komponente ansehen und nicht einzelne enthaltene Funktionen. Da Software jedoch immer irgendwie von seiner Umgebung abhängig ist, kann diese Forderung nur relativ gesehen werden. Abhängigkeiten sollten jedoch implizit klar (z.B. Betriebssystem) oder explizit dokumentiert sein (z.B. verwendete Bibliotheken).

Identifikation (identification) Sie sollte klar von ihrer Umgebung abgegrenzt (z.B. eine Datei) und nicht mit anderen Elementen vermischt sein.

Funktionalität (functionality) Sie sollte eine genau bezeichnete Funktionalität zur Verfügung stellen.

Schnittstellen (interfaces) Sie sollte klare Schnittstellen zu entsprechenden Systemen und Bereichen haben, die jeweils Unwesentliches für die Verwendung dahinter verbergen.

Dokumentation (documentation) Sie sollte mit einer angemessenen Dokumentation versehen sein, die ein Auffinden, Verstehen, Anpassen und Integrieren von ihr ermöglicht.

Status (reuse status) Der Status einer Komponente umfasst alle wesentlichen Meta-Informationen zu ihrer Entwicklung (Ersteller, Weiterentwicklung, Support-Kontakt) und ihren gegenwärtigen Entwicklungsstand.

Diese Definition ist für den Großteil der heutigen Komponentensysteme passend und gibt eine ungefähre Vorstellung, was darunter normalerweise verstanden wird. Jedoch wird die Definition aufgrund dieser spezifischen Forderungen manchmal nicht vollständig zutreffen und an anderer Stelle nicht ausreichend präzise sein. Council/Heineman haben mit ihrer Definition eine Lösung für dieses Problem geschaffen, in dem sie entsprechende Forderungen nicht konkret benennen, sondern auf ein zusätzliches Komponentenmodell verweisen [Heineman 01, S. 7]:

> "A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

Das referenzierte Modell kann dann von Fall zu Fall unterschiedlich sein und ist als Standard anzusehen, dem ein Softwareelement genügen muss, um dieser Definition zu entsprechen. Da es mehrere solche Modelle geben kann, gibt es auch unterschiedliche Arten von Komponenten, die sich durch den zugehörigen Standard abgrenzen lassen. Daraus folgt aber auch, dass nach dieser Auffassung zu jeder Komponente ein entsprechendes Modell existieren muss, auch wenn es evtl. nicht explizit formuliert wurde. Welche Aspekte in einem solchen Standard festgelegt werden sollten, erläutern Sametinger/Weinreich im selben Buch durch Benennung der grundlegenden Elemente eines Komponentenmodells [Heineman 01, S.38]: Standards zu Schnittstellenbeschreibung (interfaces), Namensgebung (naming), Metadaten (meta data), Interoperabilität (interoperability), Anpassungsformen (customization), Verbindungsformen (composition), Austauschmechanismen (evolution support), Kapselung und Auslieferung (packaging and deployment). Die genannten Bereiche sind alles Schnittstellen, die der Unterstützung von Aktivitäten und zugehörigen Systemen dienen, die eine Komponente während ihres Lebenszyklus durchläuft: Auffinden, Verstehen, Anpassung und Integration. Ein derartiges Modell basiert daher auf einem Wiederverwendungssystem und beschreibt, wie eine Komponente beschaffen sein muss, um darin erwartungsgemäß zu funktionieren. Ohne eine konkrete Referenz zu einem solchen System, wird durch ein Komponentenmodell implizit auch ein entsprechendes Wiederverwendungssystem standardisiert.

Komponenten sind in diesem Sinn als Elemente eines Komponentensystems zu verstehen, wie es in Abschnitt (vgl. 3.1.1) beschrieben wurde. Den Komponentencharakter bekommen sie durch die Integration in dieses System, die damit verbundenen Arbeiten, die daraus resultierende Konformität mit dem Komponentenmodell und die daraus entstehende Möglichkeit ein Element von einem Softwaresystem zu sein, was aus diesem Komponentensystem gebaut wird. Eine Komponente zeichnet sich daher durch die Zugehörigkeit zu einem Komponentensystem als potenziellen Baustein für damit gebaute Softwaresysteme aus.

3.1.3 Komponenten- und Systementwicklung

McIlroys Vision von Komponentenfabriken, hat die Entkoppelung der Entwicklung von Komponenten und Systemen zur Folge (vgl. [McIlroy 68]). Komponenten werden dann nicht mehr für ein bestimmtes System entwickelt oder angepasst, sondern stellen „Fertigbauteile" dar, die auf Vorrat produziert und bei Bedarf „von der Stange" gekauft werden. Man spricht dabei von einem Twin-Life-Cycle Modell. Diese Aufteilung wird in der Literatur vielfach beschrieben, wobei die beiden Bereiche teilweise unterschiedlich benannt werden: "development *for* reuse" vs. "development *with* reuse", "component development" vs. "application development", "component process" vs. "solution process" (vgl. [Sametinger 97, S. 158], [Overhage 06, S. 36], [Fritschi 02, S. 59], [Allen 98], [Sindre 95, Karlsson 95]). Im Folgenden werden sie als Komponenten- bzw. (Software-)Systementwicklung bezeichnet (vgl. Abb. 3.1).

Es entsteht durch diese Aufteilung zunächst der Eindruck, dass Komponenten ausschließlich für die Erstellung von Applikationen genutzt werden und sich damit nicht gegenseitig verwenden. Dies ist zwar möglich, aber Komponenten können auch auf andere Komponenten zurückgreifen. Ein Softwareelement kann einerseits auf anderen aufbauen und gleichzeitig andere selbst

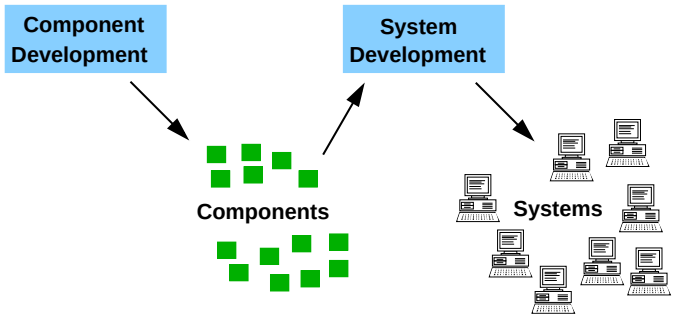

Abbildung 3.1: Twin-Life-Cycle der komponentenbasierten Systementwicklung

unterstützen. Im Grunde wird hier nur ähnlich wie bei dem Client-Server-Prinzip eine Benutzt-Beziehung dargestellt. Man kann auf diese Weise die Zusammenstellung eines vollständigen Informationssystems (s. Abs. 2.1.1) betrachten, das aus vielen Applikationen besteht, oder auch die Erstellung eines einzelnen Softwareelements, das eine Bibliothek für seine Aufgaben verwendet. Die primär betrachtete Entwicklungsarbeit ist die Systementwicklung, die Komponenten verwendet, die wiederum in Komponentenprozessen entwickelt wurden und evtl. auch zukünftig überarbeitet werden. Man kann nicht mit Sicherheit ausschließen, dass das Ergebnis einer Systementwicklung wieder als Komponente genutzt wird. Jede Software kann theoretisch wiederverwendet werden und damit letztlich auch als Komponente dienen, selbst wenn das wie bei der ad-hoc Wiederverwendung nicht die ursprüngliche Absicht war. Genauso kann man nicht ausschließen, dass Komponenten bereits auf anderen Softwareelementen basieren, nur weil diese Information nicht in dem zugehörigen Komponentensystem hinterlegt ist. Es handelt sich bei dieser Trennung nicht um notwendige, reale Unterschiede in den tatsächlichen Vorgängen, sondern um eine Betrachtungsweise bzw. eine daraus resultierende Strukturierung, die Vorgänge und Ergebnisse in Beziehung zu dem jeweiligen Komponentensystem stellt. Da diese jedoch stets vorhanden sind, kann man so etwas wie ein typisches Szenario erstellen, in dem die für die Betrachtung wesentlichen Vorgänge im Zusammenhang mit dem Komponentensystem hervorgehoben und alle anderen vernachlässigt werden.

Ein notwendiger Teil der Komponentenentwicklung ist die Vorbereitungs- und Integrationsarbeit für das Komponentensystem und im Rahmen der Systementwicklung müssen entsprechend die Zusatzarbeiten für die Nutzung der Komponenten stattfinden. Diese Vorgänge sind primär gemeint, wenn im Folgenden von Komponenten- bzw. Systementwicklung gesprochen wird, obwohl sie teilweise untrennbar mit anderen Abläufen vermischt sind, die auch am Aufwand gemessen deutlich umfangreicher sein können. Es wird damit ein Bezug hergestellt zu den typischen Abläufen: Der Erstellung und Nutzung von Komponenten und den damit verbundenen Problemen und Besonderheiten. So weiß man bei der Komponentenentwicklung nicht sicher wie der Kontext aussieht, in dem die Software letztendlich verwendet wird und bei der Systementwicklung kennt man wiederum nicht den genauen Inhalt und die innere Struktur der Elemente, die man in seine Software einbaut. In beiden Bereichen geht durch diese Trennung etwas verloren, auf der einen Seite Orientierung (Komponenten), da die konkrete Umgebung fehlt, auf der anderen Seite Flexibilität (System), weil man die Kontrolle über den entsprechenden Teil abgibt. Der erhoffte Gewinn ist dabei die Reduzierung von redundanten Arbeiten und die damit verbundenen Vorteile. Dabei wird ähnlich wie bei der Client/Server Architektur eine künstliche Grenze geschaffen, die in diesem Fall das Allgemeine (Komponente) vom Speziellen (System)

trennt. Diese Umstände müssen auch in der Wahl der Entwicklungsmethodik berücksichtigt werden. Einerseits muss man einen Weg finden mit einem allgemeinen, abstrakten, diffusen Kontext umzugehen (Kontextproblem: vgl. 6.3.2). Anderseits muss man lernen, mit unhandlichen, groben Bauteilen zu entwickeln, die ihr Eigenleben besitzen und sich über die Zeit evtl. auch noch verändern. Mögliche Strategien sind, diese gut mit eigenem Material zu umhüllen oder entsprechende Wartungsmechanismen zu entwickeln. Das Letztere kann jedoch nur in Abstimmung bzw. Kooperation mit den zugehörigen Entwicklern geschehen. Das gemeinsame Problem dieser beiden Bereiche ist das "Zueinander-Finden": Komponenten brauchen passende Systeme und umgekehrt. Diese Verbindungen aufzubauen und zu entwickeln ist sehr wissensintensiv und komplex. Zusammenfassend lassen sich folgende drei Problemfelder benennen: Umgang mit dem Kontext, Wartung und Vermittlung.

3.2 Komponentenmodell

3.2.1 Eingrenzung des Gegenstands

Ein Komponentenmodell ist ein Standard, der das Verhältnis von Komponenten zu dem zugehörigen Komponentensystem festlegt. Es enthält Vorgaben zu Inhalt, Format und Schnittstellen einer Komponente. Dabei kann man den Kern einer Komponente von seiner Hülle differenzieren. Der Kern ist der Teil, der von der Entwicklungs- bis in die Ausführungsumgebung übertragen wird. Auf diesem Weg wird er evtl. transformiert, beschnitten oder ergänzt, aber das Wesentliche bleibt dabei erhalten. Seine Hülle jedoch entsteht erst bei der Vorbereitung für den Transfer und wird während der Integration wieder entfernt. Es handelt sich dabei primär um Meta-Informationen und Mechanismen, für seine Anpassung, die bei der eigentlichen Verwendung nicht mehr benötigt werden. Man kann diese eingepackten Komponenten als Transportformen betrachten. Es ist auch denkbar, dass diese Ergänzung von dem Kernobjekt getrennt verarbeitet wird und sie nur durch eine entsprechende Referenz miteinander verbunden sind.

Das Komponentenmodell regelt die Wechselwirkung der Komponenten mit ihrer Umgebung in den jeweiligen Bereichen, die für das Komponentensystem von Bedeutung sind. In Abschnitt 3.1.2 wurden bereits kurz die möglichen Bestandteile eines Komponentenmodells erörtert. Eine Komponente entsteht in ihrer Entwicklungsumgebung, durchläuft die Komponenteninfrastruktur und gelangt dadurch in ihre Ausführungsumgebung (vgl. Abs. 3.3). Das Komponentenmodell sorgt dafür, dass die dabei beteiligten Systeme und die Komponenten selbst zusammen passen, um einen reibungslosen Verlauf zu ermöglichen. Dabei sind drei Aspekte einer Komponente von besonderer Bedeutung: Metadaten, Komposition und Anpassungsmechanismen.

3.2.2 Metadaten

Während der Entwicklung wird viel Hintergrundwissen und zugehörige Informationen gesammelt. Darauf basierend werden Entscheidungen getroffen, Ressourcen verwendet, Tests durchgeführt, Strategien entwickelt, Absprachen getroffen und Erfahrungen gesammelt. Die Software selbst ist zwar das Endergebnis, aber sie enthält nur einen Bruchteil des Wissens, das zu ihr geführt hat. Teile dieses Wissens sind jedoch z.B. für ihre Vermittlung, Anpassung und Integration wichtig, weswegen entsprechende Informationen eine Komponente begleiten sollten, da sie

ansonsten später mit viel Aufwand wieder rekonstruiert werden müssen bzw. man auf sie, mit entsprechenden Konsequenzen, völlig verzichten muss. Es ist daher sinnvoll diese Informationen für die jeweilige Komponente zu hinterlegen.

Das Komponentenmodell sollte nun genaue Vorgaben machen wie, wo und in welchem Format entsprechende Metadaten hinterlegt und später wieder abgerufen oder verändert werden können. Diese Interaktion kann dann einerseits mit der Komponenteninfrastruktur selbst stattfinden, wie es in Abschnitt 3.3 geschildert wird, oder auch durch externe Werkzeuge in der Entwicklungs- bzw. Ausführungsumgebung.

Die hinterlegten Daten können dann von zugehörigen Hilfsmitteln entweder zur Information von beteiligten Akteuren oder auch für den Ablauf von automatisierten Vorgängen genutzt werden. Im letzten Fall müssen sie dann jedoch auch in einem entsprechend maschinenlesbaren Format gespeichert sein.

Inhaltlich kann es sich dabei um jegliche Informationen handeln, die für den Umgang mit der jeweiligen Komponente nützlich sind: Beschreibungen des Inhalts, Version, Entwicklungsstand, Beziehungen zu anderen Komponenten, Herkunft, Anlaufstelle für Fragen und Probleme, Verweise auf weitere Informationen, Rechte und Pflichten der Beteiligten, Eingliederung in Ordnungsstrukturen, etc.

3.2.3 Komposition

Der primäre Zweck eines Komponentensystems ist die Versorgung eines Entwicklers mit passenden Bauteilen für sein Softwaresystem. Sie können dabei unterschiedliche Rollen innerhalb eines Softwaresystems einnehmen, wie in Abschnitt 2.2.3 gezeigt wurde. Ein großer Teil der Komponenten beinhaltet dabei Prozessmodelle, die im Rahmen eines Systemknotens den Verlauf der zugehörigen Abläufe festlegen. Dafür muss der Systemknoten das enthaltene Modell verarbeiten. Diese Aufgabe wird durch andere Software oder passende Hardware erledigt, die diese Daten interpretiert und für die Ausführung entsprechender Anweisungen sorgt. Diese Prozessmodelle sind jedoch i.d.R. nicht in sich abgeschlossen, sondern verknüpft mit Modellen in anderen Komponenten, mit denen sie interagieren. Für diese Komposition der Prozessmodelle gibt es unterschiedliche Möglichkeiten, die sich durch folgende drei Aspekte charakterisieren lassen:

Kopplung Das enthaltene Prozessmodell (Client) kann auf andere verweisen (Server), die nicht in der Komponente selbst enthalten sind (z.B. der Aufruf einer Funktionsbibliothek). Oder es wird von anderen Modellen (Client) auf dieses (Server) verwiesen. Die entsprechende Kopplung zwischen den beiden Elementen kann statisch oder dynamisch sein. Im ersten Fall verweist ein Client auf einen ganz bestimmten Server (z.B. bestimmte Version einer Bibliothek), der dann für einen korrekten Ablauf auch verfügbar sein muss. Bei einer dynamischen Kopplung wird nur auf eine bestimmte Funktionalität verwiesen, in dem eine klar definierte Schnittstelle angesprochen wird (z.B. SMTP für Versenden einer E-Mail), die evtl. von vielen Servern zur Verfügung gestellt werden kann und es spielt keine Rolle, welcher ansprechbar ist, so lange die Schnittstelle besetzt ist. Die statische Variante hat den Vorteil, dass die explizite Definition einer abstrakten Schnittstelle nicht notwendig ist. Der Nachteil ist die notwendige Überarbeitung bzw. Anpassung bei dem Wechsel des entsprechenden Servers. Im Gegensatz dazu bietet die dynamische Kopplung die Flexibilität,

konkrete Elemente ohne Anpassung auszutauschen, so lange die abstrakte Schnittstellenspezifikation eingehalten wird. Dies sind zwei Extreme und es gibt verschiedene Lösungen dazwischen. So kann eine erweiterte statischen Kopplung Upgrades von Servern zulassen, die bis auf Fehlerkorrekturen keine Änderung enthalten, ohne eine Überarbeitung der Clients zu erfordern. Bei der dynamischen Variante kann man den gegenwärtigen Status als eine implizite De-Facto-Spezifikation ansehen, ohne den Aufwand einer zusätzlichen expliziten Spezifikation zu betreiben.

Interaktion Es gibt ganz unterschiedliche Arten, wie Komponenten miteinander interagieren können. Die häufigste Variante ist wohl der Funktionsaufruf, aber auch die gemeinsame Nutzung von Daten (Datenbank, Dateisystem, Arbeitsspeicher, etc.), die Inter-Prozess-Kommunikation eines Betriebssystems, die Verwendung von Datenports oder auch Ein- und Ausgabekanäle sind alles übliche Interaktionsmöglichkeiten. Zudem stellen spezifische Plattformen noch weitere Varianten zur Verfügung. Bei der Komposition von zwei oder mehr Komponenten ist die Nutzung einer dieser Methoden notwendig. Die Standardisierung dieser Interaktionsformen schränkt zwar die Auswahl der Entwickler ein, erhöht aber die Chancen für eine entsprechende Kombinationsmöglichkeit.

Einbettung Ein Prozessmodell benötigt andere Geräte und Ressourcen für seine Ausführung (Software, Ein-, Ausgabe-, Speicher-, Netzwerk-Geräte, etc.). Diese werden vom Systemknoten bzw. von dem zugehörigen Betriebssystem verwaltet. Die Integration in diesen größeren Rahmen kann nun durch die Bereitstellung einer eigenen Ausführungsumgebung durch das Komponentensystem erfolgen, z.B. die Virtual Machine von Java. Dann ist die Ausführungsumgebung der entsprechenden Komponente ein Subsystem eines Knotens und jede Komponente muss nur noch in dieses Subsystem integriert werden. Durch diese Aufteilung kann man Komponentensysteme von Betriebssystemen und Hardware-Architekturen unabhängig machen. Sie sind aber deswegen nicht plattformunabhängig, sondern die Komponentensysteme werden damit selbst zur Plattform, von der dann entsprechende Komponenten abhängig sind. So kann ein Java-Programm nicht ohne eine passende Java-Umgebung genutzt werden.

Die Alternative dazu ist das Verhältnis zur Ausführungsumgebung nicht durch das Komponentensystem vorzugeben, sondern dies im Verantwortungsbereich der einzelnen Komponenten zu belassen. Daraus kann sich jedoch das Problem ergeben, dass sie von Elementen des Systemknotens abhängen, die nicht unter der Kontrolle oder Beobachtung des Komponentensystems stehen. In diesem Fall muss der entsprechende Akteur (typischerweise der System-Administrator) für das Vorhandensein der passende Umgebung sorgen.

Ein Komponentenmodell sollte mögliche Kompositionsmechanismen vorgeben, standardisieren und damit der Infrastruktur die Möglichkeit geben, sie zu unterstützen und zu überwachen.

3.2.4 Anpassungsmechanismen

Die Verknüpfung mit ihrer Umgebung ist ein wesentliches Charakteristikum von Komponenten. Ohne irgendeine Interaktion mit ihrer Umgebung ist sie in einem Softwaresystem nutzlos. Es muss daher eine Verknüpfung stattfinden. Es gibt mehrere Möglichkeiten, in welchem Prozess diese Verbindung konkretisiert wird. Auch eine schrittweise Annäherung ist möglich. Dies kann im Rahmen jedes Prozesses, von der Entwicklung bis zur Nutzung, stattfinden, der an der Verteilung beteiligt ist (vgl. Abschnitt 3.3.2). Um so später diese Verknüpfung stattfindet, um so

abstrakter und damit schwieriger ist die vorhergehende Entwicklung, um so länger bleibt die Komponente flexibel und um so spezifischer kann dann später die Verknüpfung auf den endgültige Nutzungskontext optimiert werden. Dies bezieht sich sowohl auf die konkrete Gesamtumgebung des jeweiligen Systemknotens wie auch die im selben Softwaresystem befindlichen Komponenten.

Die Anpassung einer Komponente bedeutet, sie oder Daten, die sie beeinflussen, zu verändern. Um so mehr eine Komponente jedoch während der Verteilung verändert wird, um so höher wird die Wahrscheinlichkeit der Inkompatibilität mit dem Ausgangselement und damit erhöht sich das Risiko von Unverträglichkeiten mit einer Umgebung, die auf die unveränderte Komponente abgestimmt ist. Diesem Problem kann man entgegen wirken, in dem man die zugehörige Umgebung ebenfalls verändert, und die Konsistenz all dieser Änderungen sicherstellt. Dies kann jedoch unter Umständen mit umfangreichen Arbeiten verbunden sein und es können sich dabei zudem unbemerkte Fehler einschleichen. Es ist daher immer eine Abwägungssache, ob man einerseits eine Komponente verändert, evtl. betroffene Elemente anpasst und damit ihre Passgenauigkeit erhöht und den Aufwand reduziert, der mit ihrer Nutzung verbunden ist, oder anderseits eine Komponente und gekoppelte Elemente weitgehend unverändert lässt und dadurch eine mangelhafte Passgenauigkeit und den damit verbundenen höheren Nutzungsaufwand akzeptiert.

Zudem muss man davon ausgehen, dass es stets neue Versionen einer Komponente geben wird und die erstellten, lokalen Veränderungen damit verloren gehen. Ein weiteres Problem bei einem Upgrade ist eine zu enge Kopplung an eine Komponente. Benötigt man z.B. einen ganz speziellen internen Ablauf und nicht nur ein bestimmtes Verhalten an den bereitgestellten Schnittstellen, so könnte dies bei der nächsten Version verschwunden sein und im Extremfall die entsprechende Komponente für diesen Anwendungsfall unbrauchbar werden. Es kann also einerseits für das Verständnis hilfreich sein, Informationen über die Internas zu erhalten und anderseits problematisch, sie für die Integration tatsächlich zu verwenden.

Mögliche Änderungen während der Verteilung benötigen daher eine entsprechende Abstimmung mit den zugehörigen verbundenen Komponenten und der Infrastruktur. Man kann dabei grundsätzlich zwei Arten der Anpassung unterscheiden:

Konfiguration Hier werden bereits in einem frühen Stadium des Transfers von der Entwicklungs- zur Nutzungsumgebung entsprechende Änderungsmechanismen eingebaut, die eine kontrollierte Anpassungsmethode darstellen. Die Durchführung dieser vorbereiteten Modifikationen muss spätestens bei der Integration in das Komponentensystem stattfinden. Sie können dann aufgrund ihrer Kennzeichnung sowohl durch andere Elemente wie auch die Infrastruktur berücksichtigt werden und erhalten daher i.d.R. die Kompatibilität der Komponente zu der generischen Umgebung.

Modifikation Darunter wird hier die Überarbeitung einer Komponente verstanden, die bei der Integration in das Komponentensystem nicht vorgesehen wurde. Wenn derartige Änderungen vorgenommen werden, finden sie normalerweise im Rahmen der Integration in das Nutzungssystem statt. Aufgrund von der Unvorhersehbarkeit für die Infrastruktur und andere Komponenten besteht die Gefahr für Konflikte mit der generischen Umgebung. Deren Vermeidung liegt in diesem Fall in der Verantwortung desjenigen, der die Veränderungen durchführt.

Erinnert man sich daran, dass Komponentensysteme nichts anderes sind als konkrete Formen von systematischer Wiederverwendung, so kann man diesen Umstand nutzen, um unterschiedli-

che Anpassungsmethoden anhand der Nutzungsarten zu differenzieren. Es gibt eine ganze Reihe von Möglichkeiten der Kategorisierung, aber in diesem Zusammenhang ist die allgemeine Aufteilung in die vier Kategorien in Tabelle 3.1 nützlich.

Art	Nutzung	Studium	Änderung	Beschreibung
black-box	ja	nein	nein	einfache Nutzung ohne Änderung oder Berücksichtigung der Interna
glass-box	ja	ja	nein	zusätzliche Analyse und Nutzung der Interna; entsprechende Optimierung der Software
gray-box	ja	teilweise	teilweise	interne Änderungen; keine Veränderung der Schnittstellen
white-box	ja	ja	ja	beliebige Veränderungen der Interna und Schnittstellen

Tabelle 3.1: Einteilung der Anpassungsmethoden anhand der Nutzungsart

Sie differenziert vier Strategien der Anpassung von Komponenten nach Abschluss der Entwicklung, die sich in Art und Umfang des Zugriffs in Bezug auf Studien- und Änderungsmöglichkeiten unterscheiden. Mit Studium ist hier die Analyse der Internas gemeint, die es z.B. bei der Glass-Box-Methode ermöglicht, die Umgebung der jeweiligen Komponente zusätzlich auf ihre internen Abläufe abzustimmen und evtl. interne Schnittstellen zu nutzen, die nicht für den externen Zugriff gedacht sind. Dabei stellen Gray-Box- und White-Box-Methode eine Modifikation im obigen Sinne dar. Bei der Gray-Box ist noch anzumerken, dass es keine allgemeine Festlegung gibt, auf welchen Teil zugegriffen werden darf und auf welchen nicht. Die Trennung in Internas und äußeres Verhalten (Schnittstellen) ist hier jedoch eine sinnvolle Interpretation und davon wird im Folgenden ausgegangen.

Aus der Art der Anpassung ergeben sich bestimmte Eigenschaften, der entsprechend genutzten Komponente, da eine Analyse, Korrektur oder ein Umbau einen entsprechenden Aufwand erfordern. Das Ergebnis ist dann aber eine bessere Passgenauigkeit, die allerdings auch zu einer entsprechend engen Kopplung des restlichen Systems an die jeweilige Komponente führt, die dann den Anpassungsaufwand bei einem Austausch der Komponente normalerweise erhöht. Aus diesen Überlegungen ergibt sich die Aufstellung in Tabelle 3.2.

Art	Passgenauigkeit	Kopplung	Aufwand
black-box	niedrig	lose	wenig
glass-box	abgestimmt	anhänglich	etwas
gray-box	gut	dicht	viel
white-box	optimal	eng	maximal

Tabelle 3.2: Eigenschaften der Anpassungsarten

Desto höher die Passgenauigkeit, um so höher wird auch der Aufwand und um so enger die Kopplung. Eine enge Kopplung macht sich dann negativ bemerkbar, wenn man entsprechende Komponenten austauschen möchte, und dies dann mit einem entsprechend hohen Aufwand verbunden ist, weil man sich auf das zu ersetzende Element spezialisiert hat. Die “anhängliche”

Kopplung der Glass-Box-Methode begründet sich durch die Optimierung der entsprechenden Umgebung auf die Internas.

Ein Komponentenmodell sollte nun festlegen, welche Art der Anpassung in dem jeweiligen Komponentensystem möglich ist und wie genau sie abläuft.

3.3 Komponenten-Infrastruktur

3.3.1 Eingrenzung der Aufgaben

Wie in Abschnitt 3.1.1 erläutert, besteht ein Komponentensystem aus drei Bestandteilen: Infrastruktur, Komponenten-Standard und Wartungsmechanismus. Dabei dienen diese drei Elemente primär einem Ziel. Sie sollen die Synergieeffekte durch die Wiederverwendung maximieren und den daraus entstehenden Zusatzaufwand minimieren.

Die Infrastruktur liefert dabei einen wesentlichen Beitrag, da sie mit einer entsprechenden Automatisierung und technischen Hilfestellung viele Vorgänge erst ermöglicht und andere in ihrem Aufwand auf einen Bruchteil reduziert. Der Komponenten-Standard legt dafür fest, was die Infrastruktur von den Elementen erwartet. Der Wartungsmechanismus wird durch die Infrastruktur realisiert und hilft dabei die Passgenauigkeit, Qualität und Flexibilität der Komponenten zu optimieren. Im Folgenden soll erörtert werden, was für Aufgaben sich daraus für die Infrastruktur ergeben.

Bei der Verwendung von Komponenten entstehen neben den regulären Entwicklungsarbeiten vier zusätzliche Aufgaben: Auffinden, Verstehen, Anpassen und Integrieren der Komponenten. Für ihre Aufwandsminimierung wurden drei Maßnahmen identifiziert: Unterstützung, Vorbereitung, Wartung (s. Abs. 3.1.1).

Im Abschnitt 3.1.3 wurden drei Probleme aufgezeigt, die durch die Spaltung der Entwicklungsbereiche entsteht: Erfassung des Kontexts, Wartungsmechanismus, Vermittlung der Komponenten.

Für die Möglichkeit einer Wartung ist die Überarbeitung von Komponenten, die bereits im Einsatz sind, eine notwendige Voraussetzung. Dieser Vorgang wird in Abschnitt 3.4 betrachtet und zwei Möglichkeiten aufgezeigt: die lokale Überarbeitung im Bereich der Systementwicklung und die globale im Bereich der Komponenten-Entwicklung.

Da die Werkzeuge des Komponentensystems sowohl von den Produzenten wie auch ihren Nutzern eingesetzt werden, ist es naheliegend, alles dazwischen durch die Infrastruktur abzudecken und damit auch die Übertragung der Komponenten vom Produzenten zum Nutzer einzuschließen.

Zudem wurde bisher nicht betrachtet, dass es sich nicht nur um einzelne Komponenten handelt, sondern teilweise um tausende. So entstehen bei der Entwicklung im Laufe der Zeit dutzende Varianten und Versionen. Ein Komponentensystem kann tausende, unterschiedliche Elemente umfassen und in einem Softwaresystem können ebenfalls hunderte kombiniert werden. Die Organisation und Verwaltung dieser Mengen von Komponenten kann ohne eine entsprechende Infrastruktur nicht bewältigt werden.

Aus diesen Überlegungen ergeben sich zusammenfassend folgende potentiellen Aufgaben für die Infrastruktur:

Informieren Zum Verstehen und anderen Tätigkeiten benötigt man Informationen, die durch die Infrastruktur verwaltet und bereitgestellt werden können.

Kommunikation Der direkte Austausch von Wissen durch Kommunikation hilft beim Verstehen und kann die anderen Aufgaben unterstützend begleiten.

Vermittlung Das Auffinden von passenden Komponenten für ein Softwaresystem oder umgekehrt ist ein komplexer Vorgang, der umfangreiche Kenntnisse über alle potenziellen Kandidaten und die Anforderungen an sie benötigt.

Management An unterschiedlichen Stellen eines Komponentensystems werden Elemente gesammelt, kombiniert und wieder entfernt. Bei all diesen Vorgängen darf man die Übersicht nicht verlieren und muss für den Erhalt einer gewissen Ordnung sorgen.

Konfiguration Dies ist die Anpassung einer Komponente, die von ihren Erstellern vorgesehen ist. Dies kann mit Hilfe von entsprechenden Schnittstellen, der Einstellung von ausgewiesenen Parametern oder der Transformation des Inhalts geschehen.

Wartung Wenn eine Komponente über das ursprünglich vorgesehene Maß hinaus angepasst werden soll, muss sie inhaltlich verändert werden. Dies geschieht im Rahmen der Wartung.

Kontexterfassung Bei der Komponenten-Entwicklung fehlt der direkte Kontakt mit einer konkreten Umgebung, die erst durch die Integration in ein Softwaresystem entsteht. Dieser Mangel kann durch einen entsprechenden Rückfluss von Information bei einer zukünftigen Überarbeitung (Wartung) ausgeglichen werden.

Integration Die Eingliederung in die lokale Umgebung ist stark abhängig von dem entsprechenden Komponentensystem, Element und Softwaresystem. Der Aufwand kann von vernachlässigbar bis sehr aufwendig variieren und unterschiedliche Teilaufgaben beinhalten, die durch die Infrastruktur unterstützt werden können.

Vorbereitung Die Integration eines Softwareelements in das Komponentensystem wird hier als Vorbereitung bezeichnet. Es umfasst die Arbeiten, die notwendig sind, um den Komponentenstandard zu erfüllen. Dies kann z.B. die Bereitstellung von Schnittstellen, Metadaten, etc. umfassen.

Übertragung Hat sich ein Nutzer für eine Komponente entschieden, so muss sie auf sein System übertragen werden. Dies kann zusätzlich auch Vertragsabschlüsse, Lizenzvereinbarungen oder Zahlvorgänge umfassen. Zudem kann eine Integritätsprüfung oder ähnliches stattfinden.

Die vorstehenden Aufgaben fasse ich in vier Gruppen zusammen, die jeweils in den folgenden Abschnitten behandelt werden:

Informative Aufgaben Informieren und Kommunikation

Adaptive Aufgaben Konfiguration, Wartung und Kontexterfassung

Transfer-Aufgaben Vorbereitung, Übertragung, Integration

Organisatorische Aufgaben Management und Vermittlung

3.3.2 Transfer-Aufgaben

Die Transfer-Aufgaben decken den Prozess der Verteilung (deployment) ab, der die Übertragung einer Komponente aus der Entwicklungs- (development environment) in die Ausführungsumgebungen (execution environments) darstellt (vgl. [Dolstra 06]). In einem Komponentensystem und damit auch für diesen Prozess spielen drei wesentliche Systeme eine Rolle (siehe 3.2):

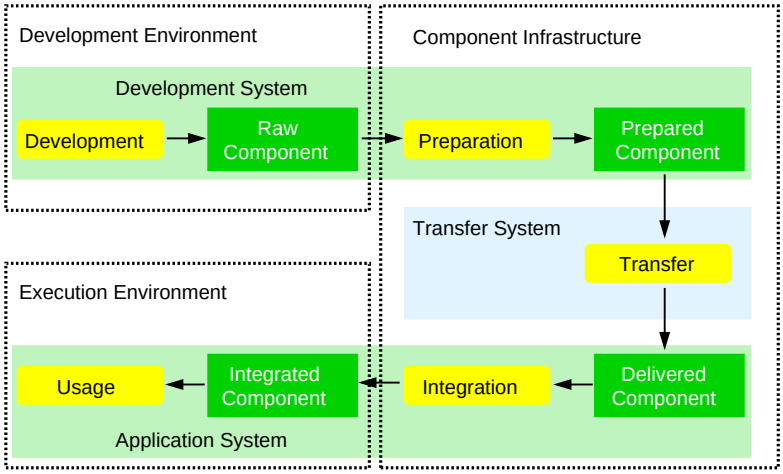

Abbildung 3.2: Wesentliche Bereiche einer Komponenten-Infrastruktur

Development System Der Ursprung der Komponente ist das System, in dem sie entwickelt wird. Hier wird auch die Aufnahme in das Komponentensystem vorbereitet.

Transfer System Das Übertragungssystem unterstützt die Übertragung einer Komponente vom Entwicklungs- zum Anwendungssystem. Es kann entweder nur als Vermittler fungieren, zusätzlich als Lager dienen oder auch eine gewisse Transformation durchführen, um die spätere Integration zu vereinfachen. Welche Funktion es genau erfüllt, hängt vom spezifischen Komponentensystem ab.

Application System Das Anwendungssystem, auf dem die Komponente genutzt werden soll. Vorher wird sie jedoch in die lokale Umgebung integriert.

Die Komponenten-Infrastruktur erstreckt sich über diese drei Systeme, schließt das Transfersystem vollständig ein und ragt in die beiden anderen Systeme hinein, da u.a. der Vorbereitungs- bzw. Integrationsprozess zumindest teilweise dort stattfindet und durch die Infrastruktur mit Diensten und Werkzeugen unterstützt wird. Der eigentliche Entwicklungs- bzw. Nutzungsvorgang findet außerhalb der Infrastruktur in der Entwicklungs- bzw. Ausführungsumgebung statt.

Eine Komponente bewegt sich durch diese Systeme und wird dabei ständig transformiert. Aus der Entwicklung (development) entsteht die Ausgangskomponente (raw), die in der Vorbereitung (preparation) durch Anpassung an den Komponenten-Standard zu einer vorbereitete Komponente (prepared) wird, die evtl. innerhalb der Übertragung (transfer) verändert wird und als ausgelieferte Komponente (delivered) im Anwendungssystem ankommt, wo sie durch die Integration (integration) in eine integrierte Komponente (integrated) umgewandelt wird. In dieser Form kann sie schließlich in der Ausführungsumgebung verwendet werden (usage).

Unter den Transfer-Aufgaben wird hier die Übertragung einer Komponente von der Entwicklung zur Nutzung gesehen, was in der Softwaretechnik als Verteilung (deployment) bezeichnet wird. In Komponentensystemen lässt sich dies in die besagten drei Prozesse aufteilen: Vorbereitung, Übertragung, Integration.

3.3.3 Informative Aufgaben

Auch wenn die Übertragung der Komponenten die zentrale Aufgabe der Infrastruktur ist, wird sie nicht ausschließlich dafür verwendet, sondern dient auch zur Verteilung von anderen Informationen, die für ein Komponentensystem von Bedeutung sind. Dabei kann man immer das selbe Muster erkennen, dass auf der Produzent-Vermittler-Konsument-Architektur beruht, in dem das Transfersystem immer die Rolle des Vermittlers spielt, der Informationsfluss aber teilweise umgekehrt wird, also vom Komponenten-Nutzer, der in diesem Fall der Produzent ist, zum Komponenten-Entwickler, der als Konsument fungiert. Es ist auch möglich, dass zwei unterschiedliche Nutzer oder Entwickler jeweils die Rolle des Produzent und Nutzer einnehmen. Zudem benötigt das Transfersystem selbst gewisse Informationen, um die Rolle des Vermittlers wahrnehmen zu können.

In jedem der drei involvierten Systeme werden unterschiedliche Informationen benötigt:

Application System Zum Auswählen, Anpassen und Integrieren einer Komponente ist es wichtig, sie möglichst schnell so weit wie nötig zu verstehen bzw. einschätzen zu können, wofür z.B. folgende Informationen hilfreich sein können: Funktionalitätsbeschreibung, Architektur, Anforderungen an die Umgebung, Schnittstellen, erfüllte Standards, gegenwärtiger Status, Ursprung, Wartungsdienst, bekannte Probleme, angestrebte Entwicklungsziele, Änderungshistorie.

Development System Auf der andern Seite sind für die Entwickler von Komponenten Informationen interessant, die sie für die Überarbeitung und Verbesserung verwenden können. Je nach Verwendung und Umgebung sind die Anforderungen an eine Software verschieden. Daher hat jeder Nutzer eine andere Sicht auf ein Element, die teilweise auch den Entwicklern helfen kann, die Komponente und ihre Aufgabe besser zu verstehen. Dies gilt insbesondere für generische Elemente, die aufgrund ihrer vielseitigen Einsatzmöglichkeiten manchmal eine unerwartete Verwendung finden, welche von den Produzenten gar nicht bedacht wurde. Daher sind hochwertige Rückmeldungen der Benutzter wie Anregungen, Fehlerberichte und Problembeschreibungen eine wertvolle Hilfe für die Weiterentwicklung.

Transfer System Auch bei der Suche von Komponenten spielen Informationen eine entscheidende Rolle, da mit einem möglichst geringen Aufwand das Element gefunden werden soll, was den jeweiligen Anforderungen am Nächsten kommt. Einige Beispiele für sinnvolle Suchkritierien sind Zugehörigkeit zu einer Kategorie, Schlagworte, Beziehungen zu anderen Komponenten oder Technologien, Attribute und formale Beschreibungen der Funktionalität.

Die informativen Aufgaben dienen nun der Versorgung der Akteure in den entsprechenden Systemen bzw. Abläufen mit den benötigten Informationen. Dazu gibt es zwei unterschiedliche Wege:

explizit Man bringt das Wissen in eine explizite Form, hinterlegt es und stellt einen geeigneten Mechanismus zum Abruf bereit. Dabei müssen vier Probleme bewältigt werden: Erstens, die Lösung der Frage, welches Wissen eigentlich benötigt wird. Zweitens, die richtige Aufbereitung des Wissen. Da viele verschiedene Akteure mit unterschiedlichem Vorwissen auf die abgelegten Informationen zugreifen, ist es schwierig die richtige Abstraktionsebene bzw. den passenden Detaillierungsgrad zu finden. Und drittens, die Konstruktion von effektiven Mechanismen zum Durchsuchen von größeren, abgelegten Informationsmengen ist ein allgemeines, komplexes Problem. Viertens, explizites Wissen veraltet allmählich und es erfordert eine ständige Überarbeitung.

kommunikativ Die Infrastruktur stellt die Möglichkeit zu einer direkten Kommunikation zur Verfügung. Dabei ist jedoch das Problem oftmals die große Zahl der Beteiligten. Dies macht es schwieriger, auf der Suche nach einer bestimmten Information den richtigen Ansprechpartner zu finden, da das Stellen von allen Fragen an alle Beteiligten zu einem unüberschaubaren Kommunikationschaos führen würde. Zudem ist diese Vorgehensweise ineffizient, wenn eine Information ständig benötigt wird.

Beide Methoden dienen dem selben Zweck: den Informationsbedarf der beteiligten Akteure und Systeme zu decken. Sie haben jeweils Vor- und Nachteile, die im Einzelfall gegeneinander abgewogen werden müssen. Soweit eine entsprechende Bereitschaft der Beteiligten vorhanden ist, ist es sinnvoll, beide Formen in einem Komponentensystem zur Verfügung zu stellen, um dem entsprechenden Akteur die Wahl zu lassen. Eine Kombination der beiden Ansätze ist ebenfalls denkbar, in dem man ablaufende Kommunikation archiviert und dann als abgelegte Information bereitstellt, wie dies bei Mailinglisten, Newsgroups oder Foren der Fall ist.

Ein weiterer wichtiger Aspekt ist die Strukturierung, die das Auffinden der Informationen und der Ansprechperson erleichtern kann. Folgende Dimensionen können dafür z.B. genutzt werden: Komponenten und ihre Versionen, Prozesse des Lebenszyklus, Anliegen oder Probleme (issues) und Zeitpunkte.

3.3.4 Adaptive Aufgaben

Die adaptiven Aufgaben der Infrastruktur ergeben sich aus dem Bemühen, eine Komponente in ihre vielen Ausführungsumgebungen zu integrieren bzw. ihre Passgenauigkeit dafür zu verbessern. Zur Erreichung dieses Ziels kann man die zwei Vorgehensweisen aus Abschnitt 3.2.4 unterscheiden: die Konfiguration und die Modifikation. Beide Vorgänge werden hier als Teilvorgang der Verteilung einer neuen Komponente gesehen. Die Anpassung von bereits integrierten Komponente wird als Wartung bezeichnet und in Abschnitt 3.4 behandelt.

Zudem kann man die Erfassung des Kontextes, d.h. der Umgebung einer Komponente in einer konkreten Ausführungsumgebung als Teil der Anpassung betrachten, da das Wissen über den Kontext eine entsprechende Voraussetzung darstellt.

Im Folgenden werden diese drei Vorgänge näher betrachtet.

3.3.4.1 Konfiguration

Die Konfiguration findet im Rahmen der Verteilung statt, wie sie in Abschnitt 3.3.2 beschrieben wurde und eine Komponente von der Entwicklung zu der Nutzung überträgt. Typischerweise

findet dies während der Integration in das Anwendungssystem statt. Es ist aber auch schon eine entsprechende Anpassung im Transfersystem während der Übertragung vom Entwicklungssystem zum Anwendungssystem denkbar. Für diesen Vorgang ist charakteristisch, dass er schon von den Entwicklern vorgesehen ist und sie passende Schnittstellen und Informationen bereitstellen, um eine entsprechende Überarbeitung zu ermöglichen.

Sowohl in der Übertragung wie auch in der Integration führen entsprechende Akteure diesen Prozess nicht nur mit *einer* Komponente durch, sondern in Abhängigkeit von dem Komponentensystemen teilweise mit hunderten. Daher erleichtert ein einheitliches Konfigurationssystem die zugehörigen Arbeiten beträchtlich. Auf diese Weise können einerseits entsprechende Werkzeuge und Vorlagen in dem Vorbereitungsprozess genutzt werden und anderseits wird durch einheitliche Schnittstellen eine Automatisierung und effiziente Bearbeitung in der Übertragung und Integration erleichtert.

3.3.4.2 Modifikation

Sind die bereitgestellten Konfigurationsmöglichkeiten nicht genug, um die Komponente ausreichend in den lokalen Kontext einzubinden, muss man entweder sie oder die zugehörige Umgebung entsprechend verändern. Da es sich hier um lokale Veränderungen handelt, werden sie zunächst nicht bei der globalen Entwicklung berücksichtigt und machen ein reguläres Upgrade schwierig oder unmöglich und es kann zu Inkompatibilitäten mit der sonstigen Umgebung führen, die durch das Komponentensystem bereitgestellt wird.

Die Infrastruktur kann nun entsprechende Hilfestellung leisten bei der Vermeidung von derartigen Konflikten, der Verwaltung dieser lokalen Änderungen und ihrer Integration in neue Komponentenversionen.

3.3.4.3 Kontexterfassung

Der Umgang mit dem Kontext ist ein grundlegendes Problem bei einer komponentenbasierten Entwicklung, wie dies bereits in Abschnitt 3.1.3 erörtert wurde, da bei der Erstellung der Komponenten zunächst keine konkreten Informationen über die tatsächlichen Anwendungssysteme vorliegen. Erst nachdem die initialen Arbeiten abgeschlossen wurden und die Komponenten sich im Einsatz befinden, können Entwickler im Rahmen einer Auswertung etwas über die konkreten Umgebungen erfahren.

Die Anforderungsanalyse ist jedoch bereits bei *einem* konkreten Anwendungssystem eine schwierige und aufwendige Arbeit (vgl. [Rupp 07]), die mit Berücksichtigung aller involvierten Systeme in der üblichen Detaillierung in keinem Verhältnis zu der Einsparung von redundanter Arbeit steht. Insbesondere, wenn man sich klar macht, dass die Zusammenführung von entsprechenden Anforderungskatalogen einzelner Systeme nochmal einen erheblichen, zusätzlichen Aufwand verursacht. Der Kontext kann daher nicht in der gewohnten Exaktheit erfasst werden. Stattdessen muss man einen Kompromiss finden zwischen vollständiger Erfassung und einer perfekten Passgenauigkeit auf der einen Seite und dem zugehörigen Aufwand auf der anderen. Man liefert damit den Nutzern eine Komponente, die einigermaßen zu ihren Ansprüchen passt und überlässt es dann ihnen, wie sie diese suboptimale Komponente in ihr System einpassen.

Es gibt nun unterschiedliche Möglichkeiten, wie man mit dieser Situation umgeht und welche Verfahren und Strategien man nutzt, um den Aufwand zu minimieren und dabei die Passgenauigkeit der Komponenten zu maximieren. Dieses sehr umfangreiche Thema wird in Kapitel 6 ausführlich behandelt.

Eine solche Strategie ist die in Abschnitt 3.4 beschriebene Wartung, da in einem solchen Zyklus nur konkrete Abweichungen zum Gegenstand der Entwicklung gemacht werden und damit eine höhere Passgenauigkeit ohne eine vollständige Spezifikation erreicht wird.

3.3.5 Organisatorische Aufgaben

Es geht hier um Management und Vermittlung von Komponenten und Informationen, die im Rahmen des Komponentensystems eine Rolle spielen. Zu den Informationen wurde in Abschnitt 3.3.3 bereits einiges geschrieben.

In allen drei involvierten Systemen von Komponentensystemen spielt die Verwaltung von Komponenten und die zugehörigen Informationen eine wichtige Rolle:

Development System Komponenten müssen sich wie jede Software ständig weiterentwickeln, um mit der Veränderungen ihrer Umgebung Schritt zu halten. Dadurch gibt es immer wieder neue Versionen. Zudem ist es nicht immer möglich mit einer Komponente alle erwünschten Umgebungen abzudecken, aber es macht auch keinen Sinn zwei völlig getrennte Komponenten zu bauen. Der Ausweg aus dieser Situation sind mehrere Varianten, die sich nur in den notwendigen Bereichen unterscheiden und ansonsten gleich sind und daher gemeinsam entwickelt werden. Des Weiteren können Komponenten auf andere aufbauen und eine Verwaltung der daraus entstehenden Abhängigkeiten kann die Entwicklung und Vorbereitung unterstützen. Auch werden viele Informationen bei der Entwicklung gebraucht oder für die Nutzer erzeugt, die eine entsprechende Verwaltung benötigen.

Transfer System Das Transfersystem fungiert als Bindeglied zwischen den Produzenten und Nutzern von Komponenten, daher müssen Komponenten dort abgelegt oder registriert sein, um von potenziellen Nutzern durch einen entsprechenden Such- oder Vermittlungsmechanismus gefunden zu werden. Zudem können die dort bekannten Elemente als Sammelpunkt für zugehörige Informationen dienen, die dann weitergeleitet werden oder ebenfalls für entsprechende Anfragen bereit gehalten werden.

Application System Im Anwendungssystem können viele unterschiedliche Komponenten verwendet werden, die evtl. auf andere Elemente angewiesen sind. Zudem kann es gelegentlich neuere Versionen dieser Komponenten geben, die dann die alten ersetzten sollen. Die Bedürfnisse des Nutzers können sich mit der Zeit auch ändern, so dass eine Komponente entfernt oder hinzugefügt werden soll. Zudem kann der Nutzer zugehörige Informationen zu diesen Elementen benötigen oder will seine eigenen veröffentlichen.

In jedem dieser Systeme ist die Unterstützung des Managements entsprechender Komponenten und auch der zugehörigen Informationen eine mögliche Aufgabe der Komponenten-Infrastruktur. Passende Managementsysteme auf den jeweiligen Systemen sind zudem miteinander verbunden und es ist dem jeweiligen Akteur damit möglich, auf die anderen Systeme und

die dort vorhandenen Komponenten und Informationen zuzugreifen, soweit ihm dies gestattet ist. Die Managementsysteme sind damit die primären Schnittstellen der jeweiligen Akteure.

Die Vermittlungsfunktion umfasst das Sammeln, Registrieren und Aufbereiten von durchsuchbaren Daten. Dies können Metadaten zu Komponenten sein, um aus dem Angebot der verfügbaren Elemente passende zu identifizieren oder auch andere Informationen, die im Rahmen des Komponentensystems eine Rolle spielen. Die Vermittlung wird normalerweise vom Transfersystem übernommen. Da es das Bindeglied zwischen allen Entwicklungs- und Anwendungssystemen ist.

3.4 Wartung von Komponenten

In diesem Abschnitt soll die Situation betrachtet werden, wenn ein Komponentensystem für eine Reihe von Softwaresystemen genutzt wurde und der Bedarf entsteht, Komponenten zu überarbeiten.

Die Entwicklung eines Softwaresystems ist stets mit einem erheblichen Aufwand verbunden, den man möglichst nicht erneut investieren möchte. Daher wird man bei kleinen Änderungen der Anforderungen oder erkannten Fehlern nicht wieder von vorne anfangen, sondern versuchen das gegenwärtige System anzupassen. Hatte man bei der ursprünglichen Entwicklung noch die freie Auswahl aus allen verfügbaren Komponenten, so ist es normalerweise bei einem bestehenden System nicht mehr so einfach einzelne Elemente ohne weiteres auszutauschen, da sie sich oft in einem komplexen Beziehungsgeflecht zu den anderen Bestandteilen befinden. In diesem Fall ist eine Überarbeitung der betreffenden Komponenten i.d.R. die beste Option.

Änderungswünsche sind bei einem längerfristigen Einsatz keine Ausnahme, sondern die Normalität. Es ist daher bei einer Verwendung auf vielen Systemen mit regelmäßigem Anpassungsbedarf von unterschiedlichen Nutzern zu rechnen. Es stellt sich daher die Frage, ob, wann und wie diese Änderungen durchgeführt werden können und wie genau man diesen Vorgang organisiert.

Dafür dient die Wartung einer Komponente. Mit diesem Begriff ist ihre Überarbeitung zur Annäherung an die Anforderungen in ihrem Nutzungskontext gemeint. Es handelt sich dabei stets um Komponenten, die sich bereits im Einsatz befinden. Der Ausgangspunkt ist ein konkreter Änderungswunsch (request), der eine Anpassung der gegenwärtigen Version mit einem konkreten Ziel beinhaltet.

3.4.1 Wartungszyklus

Ein Wartungszyklus besteht aus vier Prozessen (vgl. Abb. 3.3). Er wird angestoßen durch Erfahrungen mit einer Komponente während der Verteilung und Verwendung. Es handelt sich dabei meistens um unvorhergesehene Veränderungen, die durch dabei gewonnene Erkenntnisse ausgelöst werden. Damit diese überhaupt genutzt werden können, müssen sie jemandem mitgeteilt werden, der in der Lage ist, die Komponente zu verändern. Diese Person muss die rechtlichen und technischen Möglichkeiten sowie das passende Hintergrundwissen besitzen, um entsprechende Anpassungen vornehmen zu können. Normalerweise sind dies die ursprünglichen Entwickler.

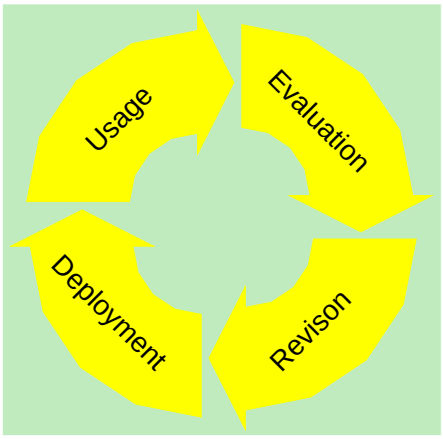

Abbildung 3.3: Wartungszyklus einer Komponente

Zunächst findet eine Auswertung (evaluation) statt, in der gemachte Erfahrungen formuliert, übermittelt, gesammelt und begutachtet werden. Anschließend werden entsprechende Anpassungen für die ausgewählten Änderungsvorschläge entwickelt (revision), die dann wieder verteilt (deployment) und genutzt (usage) werden. Es kann sich bei diesem Zyklus einerseits um die in Abschnitt 3.3.2 beschriebene Verteilung mit anschließender Nutzung und vorausgehender Entwicklung (hier: Überarbeitung) handeln, die durch eine zusätzliche Auswertung angestoßen wurde. Damit wird die überarbeitete Komponente wie eine neue Version oder sogar eine völlig neue Komponente behandelt. Im letzteren Fall ist sie "glücklicherweise" ein guter Ersatz für die alte. Der konkrete Ablauf hängt davon ab, ob das Komponentensystem im Rahmen der Verteilung einen besonderen Upgrade-Mechanismus besitzt und damit eine neue Version als Nachfolger der alten Komponente erkannt wird und so nur eine Aufrüstung durchgeführt werden muss. Sollte dies nicht der Fall sein, muss der entsprechende Akteur diesen Umstand erkennen und diesen Mechanismus durch manuelle Arbeiten ersetzen.

In manchen Fällen kann diese Anpassung bereits durch eine erneute Konfiguration erreicht werden, wie sie evtl. schon während des Transfers für die Anpassung an die lokale Umgebung durchgeführt wurde (vgl. Abs. 3.3.4). Reicht dies jedoch nicht aus, ist eine Modifikation notwendig, die entweder lokal oder global durchgeführt werden kann, wie es in Abschnitt 3.4.2 und 3.4.3 beschrieben ist. Bei der lokalen Variante handelt es sich um einen gesonderten Wartungsmechanismus, der nur eine lokale Überarbeitung auf dem jeweiligen einzelnen Anwendungssystem umfasst und diese überarbeitete Komponente zukünftig unabhängig von der ursprünglichen weiterentwickelt wird. Dies schließt jedoch eine spätere Rückkehr zu der allgemeinen, einheitlichen Komponente nicht aus, wenn die entsprechenden Probleme in einer neuen Version behoben wurden.

3.4.2 Lokale Modifikation von Komponenten

Findet die Wartung der Komponenten im Bereich der Systementwicklung statt, so führt dies zu unabhängigen Überarbeitungen auf jedem System, wie es Abb. 3.4 zeigt und im Folgenden erläutert wird:

Die Systeme S1und S2 sind beide aus der Installation (installation) des gemeinsamen System

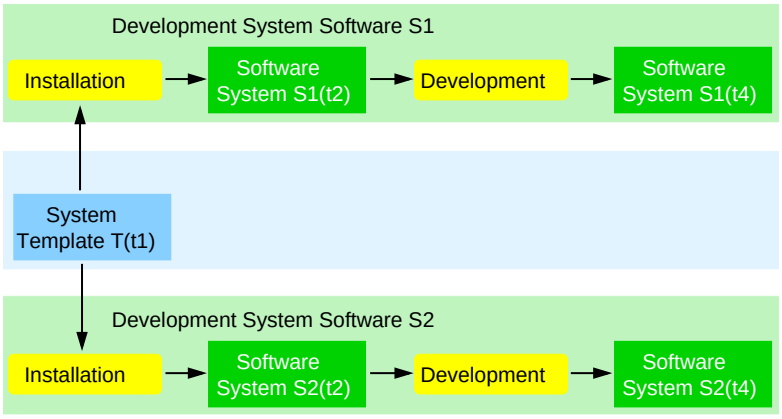

Abbildung 3.4: Unabhängige Entwicklung von zwei Systemen aus einem System Template

Templates T entstanden, das man sich als ein Sortiment aus aufeinander abgestimmten Komponenten vorstellen kann. Es ist den Betreuern der jeweiligen Softwaresysteme überlassen, welche Elemente sie aus diesem Sortiment auswählen, daher kann man davon ausgehen, dass sich in S1 und S2 teilweise unterschiedliche Komponenten befinden. Alle Komponenten, die jedoch in beiden Systemen vorhanden sind, sind sich aufgrund des gemeinsamen Ursprungs relativ ähnlich. Trotzdem sind sie nicht gleich, da sie durch die Anpassung an die jeweilige Systemumgebung während der Installation verändert wurden, die bei dieser Betrachtung entsprechende Konfigurations-, Anpassungs- und Integrationsarbeiten umfasst. In dem betrachteten Beispiel sollen die beiden Systeme und damit die enthaltenen Komponenten an die jeweilige Systemumgebung und die daraus resultierenden Anforderungen besser angepasst werden. Die durch das Komponentensystem bereitgestellten Konfigurationsmöglichkeiten wurden ausgeschöpft und haben kein zufriedenstellendes Ergebnis geliefert. Die weitere Anpassung erfordert daher die Veränderung bzw. Erweiterung der Komponenten selbst. Dies geschieht in der darauf folgenden Entwicklung (development), die jedoch jeweils unabhängig vom anderen Systemen abläuft. Das Ergebnis sind die angepassten Softwaresysteme S1(t4) bzw. S2(t4).

Dieses Vorgehen bringt durch die Unabhängigkeit ähnliche Vor- und Nachteile, wie sie bereits eine Neuentwicklung gegenüber der Verwendung von Komponenten hat: Zwischen der Entwicklung von S1 und S2 ist keine Koordination notwendig, d.h. entsprechender Aufwand entfällt und die Betreuer der Systeme haben mehr Entscheidungsspielraum. Der Nachteil dieses Vorgehens ist, dass nach der anfänglichen Installation keine weiteren Synergieeffekte mehr genutzt werden können, wenn z.B. beide Systeme das selbe Problem haben. Dadurch kann es mit der Zeit hunderte unterschiedlicher Versionen eines Elements geben, die eine Sicherstellung der Konsistenz, Kompatibilität und Qualität durch das Komponentensystem erschweren oder unmöglich machen.

3.4.3 Globale Modifikation von Komponenten

Die Alternative dazu ist die Wartung der Komponenten im Bereich der Komponentenentwicklung, die zu einer globalen Bearbeitung von Problemen führt, wie sie Abb. 3.5 zeigt.

Das System-Template T ist die Grundlage der Systeme S1 und S2 und enthält damit auch die

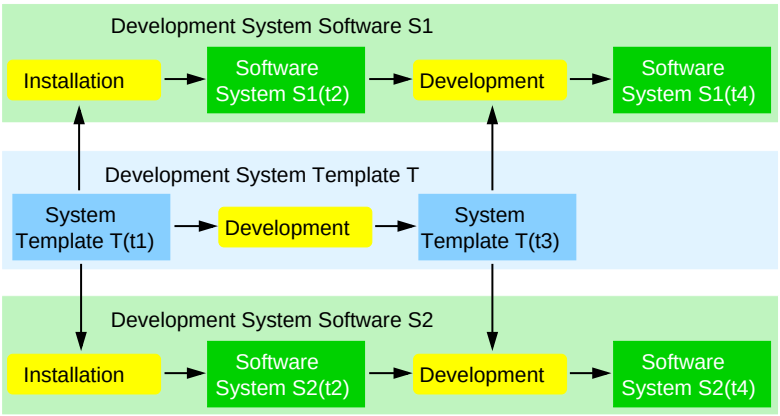

Abbildung 3.5: Kooperative Entwicklung von zwei Systemen mit Hilfe von System Templates

entsprechende Komponente, die überarbeitet werden soll. Wird eine Änderung im Rahmen der Komponenten-Entwicklung gemacht, bedeutet es das Template T entsprechend zu überarbeiten und damit den Betreuern der Softwaresystemen die Möglichkeit zu geben, entsprechende Änderung zu übernehmen und ihre System anzupassen.

Die entsprechenden Vorgänge auf Komponentenebene sind in Abb. 3.6 dargestellt. Aus Platzgründen wurde das System S2 hier weggelassen.

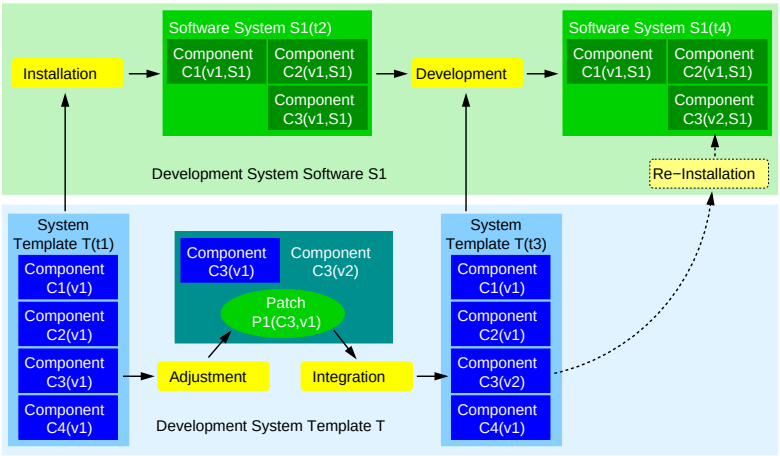

Abbildung 3.6: Vorgänge auf Komponenten-Ebene bei einer kooperativen Entwicklung

Aus dem System Template T werden zum Zeitpunkt t1 die Komponenten C1, C2 und C3 in Version v1 entnommen und in das Softwaresystem S1 installiert. Durch entsprechende Arbeiten werden sie zu lokalen, auf S1 angepassten Komponenten. Aufgrund eines entsprechenden Änderungsbedarfs wird für C3(v1) eine Änderung (patch) P1(C3,v1) entwickelt, die anschließend in Template T integriert wird, das dann zum Zeitpunkt t3 Komponente C3 in der Version v2 enthält. Diese überarbeitete Fassung wird im Rahmen von Entwicklungsarbeiten (development) durch eine Re-Installation integriert. S1 enthält nun die überarbeitet Version v2 von C3.

Ein grundlegendes Problem dieses Vorgehens ist die Sicherstellung, dass durchgeführte Änderungen niemandem schaden und zu allen Systemen kompatibel sind, auf denen die Kompo-

nente bereits eingesetzt wird, was aufgrund der allgemeinen Problematik einer Komponenten-Entwicklung schwierig ist (Kontextproblem). Daher gibt es unterschiedliche Entwicklungszweige und Versionen, mit denen man dann den Nutzern die Wahl lässt zwischen maximaler Kompatibilität, was normalerweise als Wartungsversion (maintenance release) bezeichnet wird, und umfangreicher Verbesserung.

3.5 Modell der Open-Source-Entwicklung

Debian GNU/Linux 4.0 als Beispiel für eine Linux-Distribution umfasst 18200 Pakete [Debian 07, Etch Release Note]. Diese Pakete sind fertige Komponenten, die für eine Systementwicklung verwendet werden können. Ein daraus erstellte Installation auf einem Computersystem besteht typischerweise aus über tausend Paketen. Da die Elemente vielfach verwendet werden können, handelt es sich um eine Methode zur Wiederverwendung. Booch/Brown sprechen sogar von der evtl. erfolgreichsten Form von Wiederverwendung im großen Stil in der Softwareindustrie [Brown 02, S. 124]. Die Betrachtung der Vorgänge legt daher nahe, dass es sich bei Open-Source um ein Paradebeispiel für eine komponentenbasierte Entwicklung handelt. Auch Brügge et al. sehen entsprechende Parallelen (vgl. [Brügge 04, Kapitel 4]).

Im Folgenden wird gezeigt, dass die Open-Source-Entwicklung ein vollständiges Komponentensystem umfasst, in dem die in Abschnitt 3.1.1 geforderten drei Elemente (Komponentenmodell, Komponenten-Infrastruktur und Komponenten-Wartung) identifiziert und die wesentlichen Besonderheiten dieses Systems charakterisiert werden.

3.5.1 Komponenten-Infrastruktur

Komponentensysteme beinhalten als eine systematische Form der Wiederverwendung eine Menge von wiederkehrenden Aufgaben, die eine entsprechende Unterstützung von einer passenden, technischen Infrastruktur geradezu erfordert. Die erfolgreiche Identifikation von einer entsprechenden Infrastruktur ist ein Beleg dafür, dass entsprechende Abläufe im Open-Source-Bereich tatsächlich stattfinden und damit ein Komponentensystem umfasst.

Im Abschnitt 3.3 wurden folgende Aktivitäten identifiziert, die durch eine Komponenten-Infrastruktur unterstützen werden können: Informieren, Anpassen, Transferieren und Organisieren. Im Folgenden werden nun Werkzeuge benannt und eingeordnet, die entsprechende Aufgaben im Open-Source-Bereich erfüllen.

3.5.1.1 Transferieren

Der Transfer ist die Übertragung einer Komponente von der Entwicklungsumgebung in die Ausführungsumgebung (vgl. 3.3.2). Für diesen Vorgang wurde im Open-Source-Bereich das Paketsystem entwickelt. Das Software-Release eines Open-Source-Projekts (raw component) wird mit Hilfe von Werkzeugen für die Paketerstellung in den Kontext einer Distribution eingepasst und für den Transfer in die Ausführungsumgebung vorbereitet (prepared component). Das endgültige Paket im Transportformat (z.B. DEB- oder RPM-Format) wird dann entweder bereits in

diesem Prozess erstellt oder durch die Infrastruktur des Verwaltungssystems für die Paketsammlung oder auch -Distribution (transfer system), in dem das Paket abgelegt wird. Von dort kann es von dem entsprechenden Paketverwaltungssystem eines Systemknotens (z.B. apt-get, dpkg, etc. bei Debian) abgeholt werden und befindet sich dann zunächst auf dem lokalen System (delivered component). Durch den Integrationsprozess wird diese Komponente mit entsprechenden Hilfsmitteln in das lokale Informationssystem integriert (integrated component). Dabei wird normalerweise mit passenden Werkzeugen (z.B. deb-conf bei Debian) eine Konfiguration durchgeführt oder sogar wie bei Gentoo eine für den lokalen Kontext optimierte Version erzeugt.

3.5.1.2 Informieren

Alle drei involvierten Systemkategorien (Application-, Development- und Transfersystem) benötigen umfangreiche Informationen zur Wahrnehmung ihrer Rolle in einem Komponentensystem (vgl. Abschnitt 3.3.3). Im Open-Source-Bereich wird dafür eine umfangreiche Infrastruktur zur Verfügung gestellt, die das Bereitstellen, Vermitteln und Abrufen von Informationen unterstützt. Dabei handelt es sich im Wesentlichen um folgende Infrastruktur:

Portale Die meisten Open-Source-Projekte und Distributionen haben im World Wide Web eine Internet-Präsenz, in der wesentliche Informationen dazu ausgetauscht, gesammelt, strukturiert und bereitgestellt werden. Da ein Großteil der Open-Source-Projekte ähnliche Anforderungen an diese Infrastruktur stellen, haben sich sogenannte Hosting-Plattformen herausgebildet, die eine geeignete, generische Infrastruktur bereitstellen (z.B. Sourceforge, BerliOS). Größere Open-Source-Projekte und Distributionen haben spezifische Portale, die für ihre jeweiligen Bedürfnisse optimiert sind. Diese Portale stellen jedoch i.d.R. ähnliche Hilfsmittel für informative Aufgaben bereit: Mailinglisten, Newsgroups, Foren, unterschiedliche Tracker (Defekte, Anregungen, etc.), Wiki, Dokumenten-Sammlung und Informationen zur Änderungshistorie.

Meta-Informationen Die Werkzeuge zum Erstellen von Paketen umfassen normalerweise auch eine Möglichkeit umfangreiche Meta-Informationen darin abzulegen, die dann für die Vermittlung, Auswahl und andere Vorgänge innerhalb des Komponentensystems genutzt werden können. Auf der anderen Seite gibt es Hilfsmittel, die diese Informationen anzeigen, durchsuchen und auswerten können, um entsprechende Vorgänge zu unterstützen (z.B. Auswahl, Erfüllung von Paket-Abhängigkeiten, etc.).

Feedback Treten bei der Benutzung Fehler oder unerwünschte Effekte auf oder hat der Nutzer einfach eine Idee zur Verbesserung einer Software, so sollten diese Informationen dann auch zu den passenden Entwicklern gelangen. Teilweise werden dafür spezielle Werkzeuge bereitgestellt (z.B. reportbug bei Debian oder Talkback bei Firefox), die das Melden von Fehlern erleichtert. Anderseits gibt es umfangreiche Bug-Tracking Systeme, die zugehörige Informationen verwalten und zur Verfügung stellen.

Verzeichnisse Sowohl Projekt-Hoster (z.B. Sourceforge) wie auch andere Anbieter (z.B. Freshmeat oder Ohloh) stellen Verzeichnisse bereit, mit denen man nach Open-Source-Projekten und damit letztendlich auch nach Komponenten suchen kann.

Internet Das Internet selbst, mit den Möglichkeiten zur Kommunikation, Veröffentlichung, Abruf und Suche von Informationen, stellt eine Infrastruktur für die informativen Aufgaben dar, die im Open-Source-Bereich ausgiebig genutzt wird.

3.5.1.3 Anpassen

In Abschnitt 3.3.4 wurden drei Vorgänge benannt, die der Anpassung dienen:

Konfiguration Die Konfiguration findet i.d.R. während der Integration einer Komponente statt und wurde hier bereits als Teil des Transfers behandelt.

Modifikation Modifikationen stellen Anpassungen dar, die über die Konfiguration hinaus gehen. Sie sind damit eine Weiterentwicklung der jeweiligen Komponente für den entsprechenden Kontext. Dies kann man auch als Wartung bezeichnen. Dabei kann man lokale und globale Wartung unterscheiden, je nachdem, ob entsprechende Änderungen nur lokal wirksam werden oder für das ganze Komponentensystem (vgl. Abs. 3.4). Bei einer globalen Wartung benötigt man neben der Infrastruktur für den Transfer und die informativen Aufgaben keine weiteren Werkzeuge. Bei einer lokalen Modifikation muss jedoch ein Mechanismus vorhanden sein, um entsprechende, lokale Komponenten einbinden zu können. Dies wird im Open-Source-Bereich durch einen zusätzlichen Bereich gelöst, der gesonderte lokale Systemdaten enthält, die nicht von dem Komponentensystem kontrolliert werden (z.B. im GNU/Linux-Dateisystem /usr/local/). Dafür werden hier jedoch keine besonderen Werkzeuge benutzt, wenn man von den regulären Entwicklungswerkzeugen absieht.

Kontexterfassung Die Kontexterfassung ist in Bezug auf die Infrastruktur größtenteils ein Problem der Informationsbeschaffung, die bereits durch die informativen Aufgaben abgedeckt wurde.

Es gibt also diese drei Vorgänge und auch entsprechende, unterstützende Werkzeuge, die jedoch nicht ausschließlich für die Anpassung benutzt werden.

3.5.1.4 Organisieren

In den jeweiligen Bereichen, in denen Entwicklungsarbeit im Open-Source-Bereich stattfindet, müssen zugehörige Ergebnisse und die dabei verwendeten Informationen verwaltet werden. Dafür haben sich entsprechende Versionsverwaltungssysteme etabliert (z.B. CVS, Subversion, Bitkeeper, git), die zugehörige Änderungen und Entwicklungszweige verwalten können.

Auf den Systemknoten werden die verfügbaren und installierten Komponenten durch passende Paket-Management-Systeme verwaltet (z.B. dpkg, rpm).

3.5.1.5 Zusammenfassung

Insgesamt kann man feststellen, dass es im Open-Source-Bereich alle Werkzeuge gibt und diese auch intensiv eingesetzt werden, die man als Bestandteil einer Komponenten-Infrastruktur erwarten würde. Es finden folglich entsprechende Vorgänge statt, die zur Verwendung eines Komponentensystems gehören. Es stellt sich jetzt nur noch die Frage, ob sie tatsächlich Teil eines solchen sind.

3.5.2 Komponentenmodell

Für die Identifikation eines Komponentenmodells im Open-Source-Bereich muss zunächst einmal geklärt werden, was man hier unter einem Komponentensystem versteht und was genau die enthaltenen Komponenten sind.

Software-Releases von Open-Source-Projekte weisen oft eine gewisse Ähnlichkeit auf, wenn sie sich mit Hilfe der Kombination "./configure; make; make install" installieren lassen, aber selbst dann eignen sie sich nur wenig als Bestandteile eines ausgereiften Komponentensystems. Es ist zwar möglich, daraus ein vollständiges Softwaresystem zusammen zu setzen, aber es ist sehr mühevoll, weil man viele Vorarbeiten leisten muss, die ansonsten durch Distributionen bereits geleistet wurden. Diese "Rohdaten" als Komponentensystem zu verwenden ist daher ein Sonderfall, der hier nicht weiter betrachtet wird. Anders sieht es mit den Distributionen aus, die dieses Rohmaterial in standardisierte Pakete transformieren, die auf die zugehörige Infrastruktur zugeschnitten sind.

Schon die unterschiedlichen Paketformate, die bekanntesten sind das DEB- und das RPM-Format, lassen jedoch erkennen, dass es im Open-Source-Bereich nicht *ein* einheitliches Komponentensystem gibt, sondern viele unterschiedliche. Diese vielen Systeme weisen jedoch große Ähnlichkeiten zueinander auf. Es stellt sich die Frage, wie sich diese Komponentensysteme voneinander abgrenzen lassen. Ein Kriterium ist sicher das Paketformat, es reicht jedoch nicht aus, da unterschiedliche Distributionen das Gleiche benutzen, aber die Pakete nicht austauschbar sind. Nehmen wir nun eine Distribution als Komponentensystem, kann es immer noch Probleme zwischen den einzelnen Versionen geben. Pakete aus Debian 4.0 (Etch, 2007) lassen sich nur selten auf einem Debian 2.0 (Hamm, 1998) System installieren oder umgekehrt. Wahrscheinlich hat sich in diesen zehn Jahren selbst das Paketformat schon zu sehr verändert. Auch die Tatsache, dass es normalerweise für jede Version einer Distribution eigene Sicherheitsupdates gibt, legt die Vermutung nahe, dass jede Version ein eigenes Komponentensystem darstellt. So wurden z.B. bei Debian mit jeder Version auch die Vorschriften verändert, die in dem "Debian Policy Manual" festgehalten sind. Sie stellen die Regeln dar, die von der zugehörigen Distributionsversion und den enthalten Paketen eingehalten werden müssen.

Wenn man einem Komponentensystem jedoch eine Weiterentwicklung und damit auch mehrere Versionen zugesteht, dann kann man dies auf die Distributionen übertragen. Im Folgenden wird daher davon ausgegangen, dass jede Distribution ein eigenes Komponentensystem mit unterschiedlichen Versionen darstellt. Diese einzelnen Versionen kann man jedoch vereinfacht ebenfalls als eigenständig betrachten. Die im Softwarepool der Distribution enthaltenen Pakete sind dann die Komponenten. Erweitert der Nutzer diesen Pool durch Pakete aus externen Quellen, die zu der jeweiligen Distributionsversion kompatibel sind, können diese ebenfalls als Komponenten angesehen werden.

Nun ist noch zu klären, wie das jeweilige Komponentenmodell aussieht. In Abschnitt 3.1.2 wurde erörtert, dass ein Komponentenmodell festlegt, wie eine Komponente beschaffen sein muss, um einwandfrei in einem Komponentensystem zu funktionieren. Für Open-Source-Projekt-Ergebnisse existieren zunächst mal keinerlei verpflichtende Vorschriften, es sei denn der entsprechende Entwickler akzeptiert sie oder stellt sie auf. Da dies jedoch für jedes Element unabhängig entschieden wird, gilt dies nicht für das gesamte Komponentensystem. Distributionen legen im Gegensatz dazu eine ganze Reihe an Regeln fest, wie alle enthaltenen Pakete beschaffen sein müssen. Zudem etablieren sich zunehmend distributionsunabhängige Standards wie der "File

Hierarchy Standard" oder die "Linux Standard Base", die ebenfalls Regeln festlegen, welche dann normalerweise von den Distributionen übernommen werden. So legt z.B. das Debian Projekt in seinem Richtlinien-Handbuch detaillierte Anforderungen fest, die ein Paket und auch die Distribution selbst erfüllen muss [Debian 07, Debian Policy Manual (V3.7.3.0, 2007-12-02)]:

> This manual describes the policy requirements for the Debian GNU/Linux distribution. This includes the structure and contents of the Debian archive and several design issues of the operating system, as well as technical requirements that each package must satisfy to be included in the distribution.

Andere Distributionen haben ähnliche Richtlinien, auch wenn sie nicht in einem derartigen Dokument zusammengefasst sind. So gibt es auch bei Fedora eine ganze Reihe von Dokumenten, die bei der Erstellung eines Pakets beachtet werden müssen: "Packaging Guidelines", "Package Naming Guidelines", "Forbidden Items" und "Package Review Guidelines" [Fedora 08]. Zudem werden durch das verwendete Paket-Format RPM bereits viele Regeln festgelegt, die man z.B. im "RPM Guide" nachlesen kann [Fedora 08, RPM Guide].

Man kann also davon ausgehen, dass es für jede Distributionsversion ein zugehöriges Regelwerk gibt, das ein mögliches Komponentenmodell darstellt. Es ist jedoch noch zu klären, ob sie tatsächlich die in Abschnitt 3.2 benannten drei Aspekte einer Komponente ausreichend standardisieren: Metadaten, Komposition und Anpassungsmechanismus.

Dazu lässt sich zunächst feststellen, dass die in den Paketen enthaltenen und durch entsprechende Richtlinien standardisierten Informationen das Kriterium für die Metadaten im Allgemeinen erfüllen.

Ein Großteil der Komposition von Paketen wird bereits im Rahmen der Paketerstellung vorgenommen und in Form von Beziehungen zwischen den einzelnen Paketen in das Komponentensystem codiert. Dies wird durch die Vorgabe von möglichen Abhängigkeiten in Paketrichtlinien standardisiert. Zudem findet im Rahmen der lokalen Konfigurationsarbeiten dann oftmals noch eine weitere Komposition statt bzw. wird präzisiert, in dem z.B. aus unterschiedlichen Optionen eine Auswahl getroffen wird. Auch diese Vorgänge werden normalerweise durch entsprechende Vorgaben in Bezug auf das Komponentensystem durch Paket-Richtlinien festgelegt.

Der Anpassungsmechanismus besteht i.d.R. aus drei Teilen:

1. Die Distributoren führen schon in der Vorbereitungsphase eine ganze Reihe von Anpassungen, Optimierungen und Verknüpfungen an den Paketen durch, so dass auf dem lokalen Systemknoten nur noch systemspezifischen Details geklärt werden müssen. Je enger der Fokus einer Distribution ist, um so mehr Anpassungen können vorab durchgeführt werden. So wäre es z.B. denkbar, dass man eine eigene Distribution für einen Konzern erstellt, um die konzernspezifischen Details in die entsprechenden Pakete gleich zu integrieren.
2. An die Paketverwaltung ist normalerweise ein Konfigurationssystem gekoppelt, welches automatisch und/oder durch Interaktion mit dem Administrator erste grundlegende Anpassungen vornimmt.
3. Viele OSS-Komponenten besitzen zudem spezielle Konfigurationswerkzeuge oder Dateien, die einer weiteren Integration in und Adaption an die lokale Umgebung dienen.

Die bisher dargestellten Anpassungsmechanismen basieren auf einer Black-Box-Nutzung. Dies ist auch nicht weiter verwunderlich, da bei Distributionen normalerweise nur binäre Pakete ohne

Änderungsmöglichkeiten verwendet werden. Dies stellt die reguläre Verwendung einer Distribution dar. Es besteht jedoch im Open-Source-Bereich aufgrund der Lizenz die Möglichkeit in eine andere Nutzungsart zu wechseln. Dies wird näher in Abschnitt 3.5.3 behandelt.

Eine weitere Besonderheit dieser Komponentensysteme ist die Kopplung mit dem Betriebssystem und die damit verbundene Ausdehnung auf das gesamte Informationssystem. Distributionen überwachen i.d.R. die gesamten Systemdaten, wie sie in Abschnitt 2.2.3 charakterisiert wurden.

Zusammenfassend lässt sich sagen, dass die Richtlinien für Distributionspakete auf den ersten Blick den Ansprüchen eines Komponentenmodells entsprechen. Für eine klare Aussage müssten einerseits die Anforderungen an ein Komponentenmodell deutlich präzisiert werden und anderseits konkrete Distributionen und ihre Richtlinien untersucht werden. Dies geht jedoch über den Fokus dieser Arbeit hinaus.

3.5.3 Erweiterter Anpassungsmechanismus und Wartung

In Abschnitt 3.5.2 wurde im Rahmen des Komponentenmodells nur die Black-Box-Variante als die übliche Nutzungsart eines Komponentensystems behandelt. Aufgrund der durch ihre besonderen Lizenzen gewährten Rechte (Nutzung, Studium, Änderung, Weitergabe) kann man jedoch bei Open-Source-Komponenten stets auch die anderen drei Nutzungsarten verwenden (vgl. Abschnitt 3.2.4): Glass-Box, Gray-Box und White-Box. Dies ermöglicht, die Komponente durch eine Modifikation anzupassen bzw. ihre Umgebung entsprechend dem Wissen um ihre internen Details zu optimieren. Da die unterschiedlichen Formen der Nutzung, wie veranschaulicht, mit unterschiedlichem Aufwand verbunden sind, ist jedoch immer zwischen Aufwand und Anpassungsgrad abzuwägen.

Der größte Teil des Aufwands besteht in dem Erwerb der notwendigen Kenntnisse über die Komponenten und ihre relevante Umgebung. Zwar ist dieses Wissen aufgrund der ständigen Veränderung nur begrenzt gültig, aber trotzdem erleichtert es für eine gewisse Zeit zukünftige Analysen, Arbeiten und damit entsprechende Arten der Nutzung. Hat man also z.B. eine Komponente schon einmal an die eigenen Bedürfnisse angepasst, sind spätere Änderungen deutlich einfacher, weil man auf bereits erworbenes Wissen zurückgreifen kann.

Daraus ergibt sich eine gewisse Verbindung zwischen Nutzungsarten und Rollen in der Open-Source-Gemeinschaft. Kombiniert man dies noch mit den zwei Bereichen der System- und Komponentenentwicklung (vgl. Abs. 3.1.3), dann ergibt sich die Zuordnung in Tabelle 3.3.

Rolle	Nutzungsart	Entwicklung
normaler Nutzer	black-box	System
Profi-Nutzer	glass-box	System
Gelegenheitsentwickler	gray-box	Komponente
Voll-Entwickler	white-box	Komponente

Tabelle 3.3: Verbindung von Open-Source-Rollen und Nutzungsarten

Damit kann man im Open-Source-Bereich das Nutzungsverhalten in diese vier Rollen unterteilen. Der normale und der Profi-Nutzer betätigen sich ausschließlich in der Entwicklung ihres

Systems, wobei der Profi-Nutzer teilweise noch die Internas der genutzten Komponente für die Optimierung nutzt. Der Gelegenheitsentwickler und der Voll-Entwickler verändern durch Modifikationen die Komponenten selbst. Jedoch beschränkt sich der Gelegenheitsentwickler auf kleinere Änderungen wie Fehlerkorrekturen oder minimale Anpassungen. Wesentliche Überarbeitungen, die tief in die Struktur der Komponente eingreifen oder die sensiblen Schnittstellen verändern, sind den Voll-Entwicklern vorbehalten, da sie ein umfassendes Wissen über die Komponente und ihren relevanten Kontext erfordern. Diese Modifikationen beschränken sich jedoch zunächst nur auf die *lokalen* Komponenten eines Systemknotens. Sie haben dadurch keine Auswirkungen auf die im Komponentensystem hinterlegte Komponente und führen damit zu den Nachteilen einer lokalen Modifikation: primär die Gefahr von resultierenden Konflikten mit der Umgebung und der Zusatzaufwand für die Übertragung der Änderungen, wenn man ein Upgrade auf eine neue Version durchführt (s. Abs. 3.2.4, 3.3.4, 3.4.2). Um diese Probleme zu vermeiden, ist es für die Gelegenheits- und Voll-Entwickler wichtig, dass ihre Änderungen in zukünftige, allgemeine, globale Versionen aufgenommen werden.

Die Nutzer können versuchen, durch die Beschreibung ihrer Probleme eine entsprechende Änderung an der problematischen Komponente auf globaler Ebene durch andere (z.B. die ursprünglichen Entwickler) zu bewirken, da sie selbst dazu nicht in der Lage sind (z.B. aufgrund fehlenden Wissens). Auf diese Weise würden dann ihre Probleme bei einer neuen Version verschwinden.

Die Einwirkung auf neue Versionen einer globalen Komponente finden durch Rückmeldungen an die Verantwortlichen statt, die in Abhängigkeit von der Rolle unterschiedlich ausfallen, da sie jeweils unterschiedliches Wissen besitzen. Es ergibt sich der in Tabelle 3.4 gezeigte Zusammenhang zwischen Rollen und (möglichen) Rückmeldungen.

Rolle	Beschreibung der Rückmeldung
normaler Nutzer	einfacher Erfahrungsbericht mit einer Beschreibung des Verhaltens
Profi-Nutzer	Bericht mit zusätzlicher Analyse der Interna der Komponenten
Gelegenheitsentwickler	kleinere, interne Änderungen
Voll-Entwickler	Schnittstellen-Anpassungen und größere Änderungen

Tabelle 3.4: Verbindung von Open-Source-Rollen und Rückmeldungen

Die Rückmeldungen gehen jedoch bereits über die Anpassung hinaus und stellen bereits Wartungsarbeiten dar. Die Wartung einer Komponente ist ihre Überarbeitung zur Annäherung an die Anforderungen in ihrem Nutzungskontext, d.h. es handelt sich dabei stets um Komponenten, die sich bereits im Einsatz befinden. Ein Wartungszyklus wird angestoßen durch einen konkreten Änderungswunsch (request), der eine Anpassung der gegenwärtigen Version mit einem konkreten Ziel beinhaltet.

Wartungsvorgänge allgemein können einerseits auf den lokalen Kontext begrenzt sein, dann handelt es sich nur um eine (erneute) lokale Anpassung. Anderseits können sie auch die Erstellung einer neue Version der globalen Komponente mit einschließen, wie es in Abschnitt 3.4.3 beschrieben wurde. Im Open-Source-Bereich gibt es beide Abläufe. Bemerkenswert ist hier die Verkettung dieser beiden Vorgänge. Oftmals werden Probleme zunächst lokal behoben und nach einer Bewährungszeit in den globalen Bereich übertragen. In diesem Fall findet die Entwicklung

zunächst im speziellen Kontext einer Systementwicklung statt und wird dann in den abstrakten Kontext der globalen Komponentenentwicklung übertragen. Dies stellt in gewisser Hinsicht eine Richtungsumkehr der üblichen Herangehensweise dar.

Zudem findet auf diese Weise eine implizite Erfassung des Kontexts einer Komponente durch die Rückmeldungen statt.

4 Prozess-Perspektive

In der Prozess-Perspektive wird die Aufteilung eines Arbeitspakets in mehrere Teilaufgaben, die damit verbunden Strukturen und Vorgänge und die daraus resultierenden Konsequenzen betrachtet.

Zunächst werden die klassischen Prozessmodelle studiert und daraus der Einheitsprozess als Basisstruktur einer zielorientierten Entwicklung abgeleitet. Der Einheitsprozess wird als die grundlegende Repräsentation des Verlaufs einer Aufgabenbearbeitung verstanden. Damit werden dann die unterschiedlichen Möglichkeiten der Aufgabenteilung untersucht. Insbesondere wird dabei die klassische Spaltung der notwendigen Arbeiten in (initiale) Entwicklung und (anschließende) Wartung betrachtet. Es zeigt sich, dass dies letztlich ebenfalls als eine Zerlegung der Gesamtarbeit in einen umfangreichen Teil am Anfang und viele kleine, anschließende Teile aufgefasst werden kann.

4.1 Strukturierung der Entwicklungsarbeit

4.1.1 Eingrenzung des klassischen Entwicklungsprozesses

Der Begriff “Entwicklungsprozess” ist nicht so eindeutig, wie es zunächst scheint. Zwar ist klar, dass damit eine Entwicklung gemeint ist, die in Form eines Prozesses beschrieben wird, aber es handelt sich dabei eben nicht um einen klar definierten Begriff, der unabhängig vom Kontext benutzt werden kann. Dies gilt selbst dann, wenn man sich auf den Bereich der Informatik beschränkt. Ein weiteres Problem stellt die Übersetzung der Begriffe vom Englischen ins Deutsche oder umgekehrt dar, da dies zu Verschiebungen in der Bedeutung führen kann. Der wohl treffendste englische Begriff “software development process” wird vom IEEE folgendermaßen definiert:

> The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. *Note:* These activities may overlap or be performed iteratively. [IEEE610.12 90, software development process]

Diese Definition basiert auf dem zugrundeliegenden Lebenszyklus, und beschreibt den Begriff einerseits durch seine Funktion in diesem größeren Kontext (Übersetzung der Nutzerbedürfnisse in ein Softwareprodukt) und die dazugehörigen Abschnitte des Lebenszyklus bzw. deren Funktion. Dies wird noch deutlicher, wenn man diese Definition mit der des zugehörigen Begriffs “software life cycle” vergleicht:

> The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. *Note:* Theses phases may overlap or be performed iteratively. *Contrast with:* software development cycle. [IEEE610.12 90, software life cycle]

Es wurde also der Entwicklungsprozess definiert durch seine Identifikation in einem breiteren Kontext, der bereits erfasst ist. So wird der Entwicklungsprozess hier definiert als ein bestimmter Abschnitt des Lebenszyklus. Im Abgleich der beiden genannten Definitionen umfasst der Entwicklungsprozess den gesamten Lebenszyklus bis auf Betrieb, Wartung und Stilllegung. Installation und Endprüfung werden als Grenzfall angesehen. Noch klarer wird es in der Definition des "software development cycle" der mit der Entscheidung beginnt, die Software zu entwickeln, und mit der Auslieferung endet:

> The period of time that begins with the decision to develop a software product and ends when the software is delivered. [...] [IEEE610.12 90, software development cycle]

Der Entwicklungsprozess lässt sich also vom restlichen Lebenszyklus durch die Auslieferung abgrenzen, der ja auch Betrieb, Wartung und Stilllegung vorangegangen sein muss. Selbst der Grenzfall Installation und Endprüfung passt in dieses Bild, da er als Teil der Auslieferung angesehen werden kann. Diese Einteilung macht sowohl bei der Regalsoftware für den Massenmarkt Sinn wie auch bei Auftragsarbeiten. Man kann also davon ausgehen, dass mit Entwicklungsprozess normalerweise folgendes gemeint ist:

> Der Entwicklungsprozess einer Software ist der Teil des Lebenszyklus, der zeitlich gesehen ihrer Auslieferung voran geht.

Was ist aber mit nachträglichen Änderungen, die z.B. durch übersehene Fehler oder Änderung der Umgebung notwendig werden? Diese Definition behält auch in diesem Fall ihre Gültigkeit, da derartige Änderungen Teil der Wartung sind. Auch größere Änderungen, die gerne nur als eine "neue Version" der Software angesehen werden, sind abgedeckt, wenn man sie stattdessen als eine andere, neue Software begreift, was ja auch der Realität entspricht, da man bei kommerzielle Produkten in der Regel die Software neu erwerben muss, d.h. sie als ein anderes Produkt angesehen wird.

Die Entwicklungsarbeit, die im Rahmen der Wartung gemacht wird, behandelt ISO12207 wie folgt: Durch die Wartung wird im Rahmen der Änderungserstellung ein Mini-Entwicklungsprozess durchgeführt, um entsprechend neue Anforderungen zu erfüllen [IEEE12207.0 96, S. 25, S. 50]. Dieser gehört *nicht* zu dem vorangegangen Entwicklungsprozess, sondern wird als eigenständig angesehen.

4.1.2 Der klassische Lebenszyklus

Es gibt mehrere Standards, die für die Strukturierung, Beschreibung, Präzisierung und Einordnung des Software-Lebenszyklus herangezogen werden können:

ISO/IEC 15288 Systems Engineering - System Life Cycle Processes [IEEE15288 04]: ISO15288 stellt einen Baukasten aus Prozessen zur Verfügung, mit dem man den Lebenszyklus von Systemen beschreiben kann.

ISO/IEC 12207.0 Standard for Information Technology - Software Life Cycle Processes [IEEE12207.0 96][1]: ISO12207 stellt einen Baukasten aus Prozessen zur Verfügung, mit dem man den Lebenszyklus von Software beschreiben kann.

IEEE Std 1074 IEEE Standard for Developing Software Project Life Cycle Processes [IEEE1074 06]: IEEE1074 stellt einen Baukasten aus Aktivitäten zur Verfügung, mit dem man den Lebenszyklus von Softwareprojekten beschreiben kann.

PMBOK A Guide to the Project Management Body of Knowledge (ANSI Standard) [PMBOK 00]: Es identifiziert und beschreibt die anerkannten Grundlagen des allgemeinen Projekt-Managements und liefert damit auch definierte Begriffe zur Beschreibung von Projekten.

SWEBOK Guide to the Software Engineering Body of Knowledge [Abran 04]: Die Softwaretechnik befasst sich laut ihrer Definition[2] mit der Erstellung, dem Betrieb und der Wartung von Software. Der SWEBOK-Leitfaden versucht durch seine Ausführungen eine konsistente Sicht zu etablieren und damit die wissenschaftliche Disziplin inhaltlich zu charakterisieren und abzugrenzen.

Diese Standards und Leitfäden beschreiben den Lebenszyklus als Ganzes oder Teile davon jeweils aus unterschiedlichen Perspektiven und auf verschiedene Abstraktionsniveaus. So befasst sich der IEEE1074 mit den Prozessen, die innerhalb eines Softwareprojekts stattfinden und der PMBOK-Leitfaden mit der Frage, wie das Management von Projekten im Allgemeinen aussehen muss. ISO15288 beschreibt die Prozesse im Lebenszyklus von allgemeinen Systemen und ISO12207 ist auf Software beschränkt.

Es gibt inhaltliche Unterschiede, aber auch große Überlappungen zwischen diesen Werken, die jedoch teilweise schwer zu erkennen und auch durch die unterschiedlichen Begrifflichkeiten nur mühsam nachzuvollziehen sind. Die Harmonisierung der unterschiedlichen Standards ist jedoch in Arbeit und wird in zukünftigen Versionen klarer werden. Die folgenden Prozesse können jedoch in allen diesen Werken in der einen oder anderen Form wieder gefunden werden:

Anforderungsanalyse Es wird erarbeitet und festgehalten, was genau entwickelt werden soll. Dies umfasst einerseits die Untersuchung der zugehörigen Umgebung (Kontexterforschung) und die Festlegung auf eine Aufgabe (Zielfestsetzung).

Design Entsprechend der vorgegebenen Zielsetzung wird ein Lösungskonzept für die gestellte Aufgabe erarbeitet.

Realisierung Es werden dazu benötigte Ressourcen beschafft (Beschaffung) und das Konzept umgesetzt (Realisierung).

Verteilung (oder auch Auslieferung) Das Ergebnis wird aus dem Bearbeitungskontext extrahiert (Vorbereitung) in den Nutzungskontext übertragen (Übertragung) und dort integriert (Integration).

[1] Für diesen Standard wurden in den letzten Jahren zwei Erweiterungen veröffentlicht: [ISO12207-0A1 02, ISO12207-0A2 04]. Zudem gibt es zwei ergänzende Standards: [IEEE12207.1 97, IEEE12207.2 97]

[2] The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. [IEEE610.12 90, software engineeering]

Mit der Verteilung ist die Entwicklung abgeschlossen und die Verwendung beginnt. Diese Prozesse bilden zusammen den Kern der Entwicklungsarbeit. Ihre Reihenfolge ist funktional bedingt und kann daher nicht geändert werden. Man kann nicht die Anforderungsanalyse nach dem Entwurf oder die Realisierung nach der Verteilung durchführen. Die anschließende Verwendung des Ergebnis ist nicht mehr Teil davon. Das integrierte Ergebnis wird entsprechend dem Zweck verwendet, aus dem die Aufgabenstellung entstanden ist.

Es gibt nun mehrere natürliche Fortsetzungen dieses Vorgangs:

Wartung Gibt es unvorhergesehene Probleme mit der Software, ist es die Aufgabe der Wartung, sich damit zu befassen. Evtl. wird dabei eine neue Entwicklungsarbeit notwendig, die auf die gleiche Weise abläuft wie der ursprüngliche Vorgang.

Überarbeitung Fast jede Software braucht früher oder später eine "Generalüberholung", um sich an die veränderte Umgebung anzupassen, z.B. veränderte Nutzerwünsche, neue Technologien oder auch andere interne Abläufe in dem betreffenden Computersystem.

Ausmusterung Irgendwann hat eine Software ausgedient und wird durch eine neue Komponente ersetzt bzw. ersatzlos aus dem System entfernt. Auch dies ist ein Vorhaben, das ein System von einem Ist- in einen Soll-Zustand überführt.

Dies alles zusammen genommen stellt einen generischen Lebenszyklus von Software dar.

4.1.3 Vorgehensmodelle

Der Ablauf bei der Softwareentwicklung folgt immer dem in Abschnitt 4.1.2 beschrieben Muster: Anforderungsanalyse, Design, Realisierung und Verteilung. Diese Prozesse finden sich daher auch in den Vorgehensmodellen wie dem Wasserfallmodell, Spiralmodell, Varianten des Prototyping, Rational Unified Process und anderen wieder. Selbst wenn sie nicht darin beschrieben sind, finden sie trotzdem bei einer entsprechenden Unternehmung statt. Teilweise haben sie andere Namen oder werden anders aufgeteilt, aber sie sind stets vorhanden.

Die Prioritäten und Randbedingungen können jedoch bei der Softwareentwicklung unterschiedlich sein. So kann z.B. die Erfüllung eines Vertrags mit den entsprechenden wirtschaftlichen Umständen oder die perfekte Lösung eines konkreten Problems im Vordergrund stehen. Der Kontext einer Software kann konstant und bereits spezifiziert oder dynamisch und noch völlig unbekannt sein. Die Folgen von enthaltenen Fehlern können tolerierbar oder fatal sein. Alle diese Faktoren stellen Anforderungen an die Abläufe dar, die bei der Auswahl einer Vorgehensweise in Betracht gezogen werden müssen.

Entsprechende Unterschiede gibt es auch bei den Vorgehensmodellen. Sie sind jedoch für jeweilige Vorhaben unterschiedlich gut geeignet. Zur Anpassungen an ein bestimmtes Szenario haben sie verschiedene Prioritäten, Strategien und Mechanismen. Dies kann z.B. die Präzision der Planung, den Umfang der Kontrolle, die Wiederholung von Vorgängen oder die Vermeidung von Fehlern betreffen.

4.2 Der Einheitsprozess

In diesem Abschnitt wird ein generisches Modell für Entwicklungsprozesse dargestellt. Es wird in Zukunft als der Einheitsprozess der Entwicklung bezeichnet.

4.2.1 Betrachtete Prozesse und ihre Gemeinsamkeiten

Softwareentwicklung befasst sich für die Erstellung eines Programms mit verschiedenen Prozessen, die in den unterschiedlichen Systemen ablaufen: Nutzungssystem, Systemknoten und Entwicklungssystem. Die dabei erstellten oder veränderten Artefakte können z.B. Anforderungen, Entwürfe, Dokumentation oder die Codierung sein, die jeweils einen anderen Aspekt eines Prozesses erfassen. Je nach dem, in welchem System der fokussierte Prozess abläuft und welcher Aspekt bearbeitet wird, hat der zugehörige Arbeitsschritt eine andere Funktion im Gesamtablauf der Softwareentwicklung. Man kann nun diese Arbeitsschritte, die selbst wieder als Prozesse aufgefasst werden können, entsprechend dieser Funktion kategorisieren. Daraus ergeben sich Prozess-*Klassen*, die man bei der Betrachtung von Entwicklungsarbeit allgemein identifizieren kann: Anforderungsanalyse, Entwurf, Beschaffung, Realisierung, Vorbereitung, Konfiguration, Integration, Wartung.

Alle diese Prozess-Klassen haben wesentliche Gemeinsamkeiten:

Zielgerichtetes Handeln Theoretisch ist es denkbar, dass man handelt, ohne ein Ziel zu verfolgen. Es wird hier aber davon ausgegangen, dass beteiligte Akteure Ziele verfolgen.

Datenverarbeitung Es geht hier um reine datenverarbeitende Prozesse, d.h. alle Ein- und Ausgaben des Prozesses sind ausschließlich Daten und keine materiellen Güter, die ausschließlich Teil der Infrastruktur sind, soweit sie in die betrachteten Vorgänge involviert sind.

Kontext-Abhängigkeit Der Kontext spielt eine zentrale Rolle. Die angestrebten Ziele beziehen sich stets auf Veränderungen im Kontext und nicht nur auf den Entwicklungsgegenstand selbst, d.h. die Bedürfnisse, Aufgaben, Erfolge, Misserfolge und die Qualität ergeben sich aus dem Kontext.

Kontextmodell Der tatsächliche Nutzungszusammenhang steht während der Entwicklungsarbeit nur begrenzt zur Verfügung. Aufgrund der Kontextabhängigkeit sind jedoch für das zielgerichtet Handeln ein entsprechendes Wissen über den Kontext notwendig, das sich die beteiligte Akteure aneignen und sich damit ein Modell von den möglichen Umgebungen für ein erfolgreiches Handeln schaffen.

Fokus Die Informationen, die während der Entwicklung genutzt werden, sind niemals vollständig und kein perfektes Abbild der tatsächlichen Umgebung, sondern erfassen nur immer einen begrenzten Bereich. Er repräsentiert jeweils den Teil der Realität, welcher für die entsprechenden Arbeiten wesentlich ist.

Transienz Betrachtet man das System, in dem das Ergebnis eingesetzt werden soll, so gibt es immer schon einen gegenwärtigen Zustand, der berücksichtigt werden muss. Die Entwicklung stellt also immer die Veränderung eines Systems dar und nicht die Erschaffung eines Neuen, selbst wenn der Gegenstand selbst neu erstellt wird. Zudem wird hier davon ausgegangen, dass es auch in Zukunft Änderungen an dem System geben wird, die auch den relevanten Kontext des Gegenstands betreffen. Sowohl der Kontext wie auch die Ergebnisse werden daher als transient angesehen.

Aufgrund dieser Gemeinsamkeiten lässt sich eine einheitliche Struktur ableiten, die sich primär auf zwei Bereiche bezieht: Ablauf und Prozess-Umgebung. Diese beiden Aspekte werden im Folgenden näher betrachtet.

4.2.2 Ablaufstruktur

Es wird davon ausgegangen, dass eine einheitliche, natürliche Abfolge von Abläufen innerhalb eines solchen Prozesses existiert, die sich aus den genannten Faktoren und insbesondere der Ausrichtung auf ein vorgegebenes Ziel ergibt. Diese Vorgänge können in Phasen aufgeteilt werden, die stets zumindest minimal vorhanden sind und in Funktion und Reihenfolge bei allen derartigen Prozessen gleich sind.

Alle diese Prozesse sind eine zielorientierte, kreative Entwicklungsarbeit und haben daher eine ähnliche Struktur. Den Anfang bildet eine zu erfüllende Aufgabe (z.B. Erstellung eines Entwurfs aus dem Anforderungskatalog), sie wird dann in dem Prozess bearbeitet und am Ende steht jeweils ein konkretes Ergebnis. Dabei bildet das tiefere Verstehen der Problemstellung, die Identifikation des richtigen Lösungsweges, seine Umsetzung und schließlich auch die Übertragung in eine übergabefähige Form jeweils eine natürliche Phase dieser Arbeit. Die konkreten Vorgänge, ihr Umfang, Abstraktionsgrad und Beschaffenheit können völlig unterschiedlich sein. Die Funktion und Abfolge dieser Phasen bleibt jedoch stets gleich:

Analyse Es wird geklärt, was genau die Aufgabe umfasst (Zielfestsetzung), wie die zugehörige Umgebung beschaffen ist und welche Lösungsmöglichkeiten es gibt (Kontexterforschung).

Konzeption Aus den erkannten Lösungsmöglichkeiten wird eine ausgewählt und zumindest so weit zu einem Konzept ausgearbeitet, dass der Akteur die Gewissheit der richtigen Wahl hat (Entwurf).

Umsetzung Das Konzept wird durch die Erstellung einer vollständigen Lösung umgesetzt (Realisierung).

Ausgabe Das Ergebnis wird aus dem Entwicklungskontext gelöst, für eine Übergabe an den entsprechenden Empfänger vorbereitet (Vorbereitung) und übertragen (Übertragung).

Ein Beispiel für diese Aufteilung ist die in Abschnitt 4.1.2 beschriebene Einteilung für das gesamte Entwicklungsvorhaben mit den spezifischen Bezeichnungen für die jeweiligen Phasen (Anforderungsanalyse, Design, Realisierung, Verteilung).

4.2.3 Prozessdomäne

Eine andere Sichtweise ist die Betrachtung dieser Vorgänge als Ganzes. Es wird nun eine Grenze zwischen internen und externen Vorgängen angenommen, die den Prozess von seiner Umgebung trennt. Dadurch wird ein Informationsaustausch zwischen diesen beiden Bereichen notwendig und der Prozess besitzt damit drei Bereiche: eine Ein- bzw. Ausgabe und die eigentliche Bearbeitung. Dieser innere Bereich stellt einen lokalen Kontext dar, in dem die zugehörigen Vorgänge ablaufen und der primär aus der entsprechenden Infrastruktur und den für die Abläufe notwendigen Informationen besteht und im Folgenden Prozessdomäne genannt wird. Die darin enthaltenen Informationen repräsentieren zum großen Teil externe Vorgänge und Gegenstände, die im Zusammenhang mit dem Entwicklungsgegenstand stehen. Es ist nun wichtig diese Daten konsistent zu halten. Dies gilt sowohl für ihr Verhältnis zueinander wie auch zu den externen Entitäten.

Für die Sicherung dieser Konsistenz ist die Eingabe besonders wichtig. Daher handelt es sich dabei nicht nur um eine einfache Aufnahme von Informationen, sondern um einen Import-Prozess, in dem eintreffende Daten überprüft, an die lokale Umgebung angepasst und in sie integriert werden. Die initiale Eingabe ist normalerweise ein Anliegen (z.B. Änderungswunsch), das eine entsprechende Entwicklung erfordert und einen zugehörigen Prozess in Gang setzt. Alle weiteren Importe werden dann durch diesen Prozess ausgelöst.

Aber auch für die Ausgabe hat diese Trennung Konsequenzen. Der gesamte Entwicklungsvorgang dient der Bereitstellung eines Ergebnisses, das im internen Kontext erstellt und anschließend im externen Bereich verwendet wird. Daher müssen Beziehungen zu der lokalen Umgebung (z.B. Abhängigkeiten) getrennt, für die Übertragung in den externen Bereich vorbereitet und evtl. die dortige, erneute Verknüpfung unterstützt werden. Dies wird hier als Export-Prozess bezeichnet.

Aus diesen Überlegungen ergibt sich die folgende Aufteilung solcher Entwicklungsprozesse:

Import Sammlung, Übertragung und Integration aller wesentlichen Informationen in den Entwicklungskontext (Kontexterforschung), die zur Erfüllung des Anliegens benötigt werden, sowie der für die Bearbeitung notwendigen Ressourcen (Beschaffung).

Bearbeitung Präzisierung des Anliegens zu einer konkreten Aufgabe (Zielfestsetzung), Erstellung eines Lösungskonzepts (Entwurf) und seine Umsetzung (Realisierung).

Export Die Ergebnisse des Prozesses werden aus dem lokalen Kontext heraus gelöst, für die Übertragung in einen fremden Kontext transformiert (Vorbereitung) und anschließend zu einem Übergabepunkt exportiert (Übertragung). Dabei kann es sich um den endgültige Anwendungsbereich, eine Sammelstelle oder einfach einen anderen Prozess handeln.

Daraus ergibt sich für eine Prozessdomäne der in Abbildung 4.1 dargestellte Aufbau.

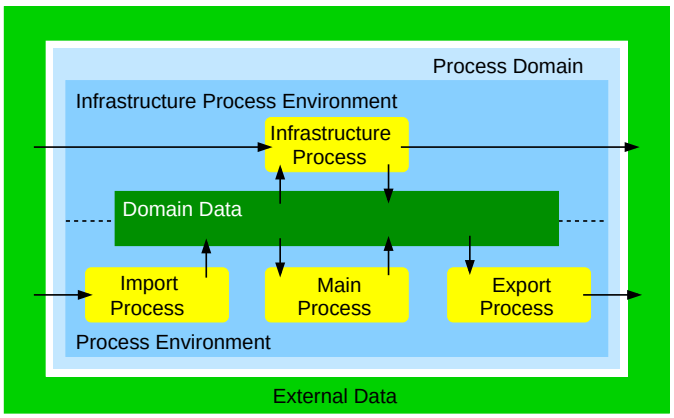

Abbildung 4.1: Aufbau der Prozessdomäne im Entwicklungsbereich

Die Domäne (process domain) stellt den Kontext der Entwicklungsarbeit dar, in den externe Informationen (external data) durch den Import-Prozess übertragen werden (import process) und aus dem Ergebnisse nach der Bearbeitung (main process) wieder exportiert werden (export process). Die technische Arbeitsumgebung, in dem die Entwicklung stattfindet, wird durch den Infrastruktur-Prozess bereitgestellt (infrastructure process), der für die Bewältigung dieser Aufgabe ebenfalls auf externe Daten zurückgreift und auch Ergebnisse teilweise nach außen weiter

gibt. Er dient nur zur Unterstützung der anderen Prozesse und sein primäres Ergebnis ist die Bereitstellung eines entsprechenden Infrastruktur-Dienstes. Man kann damit die Prozessdomäne in zwei Teile gliedern: einerseits die Prozess-Umgebung (process environment), die aus den Domänedaten (domain data) und der bereitgestellten Infrastruktur besteht, und andererseits die Umgebung des Infrastruktur-Prozesses (infrastructure process environment), in dem er abläuft.

Die hier dargestellten Teilprozesse laufen nicht mehr notwendigerweise sequentiell ab, sondern können auch parallel sein. Die Aufteilung ist rein funktional begründet. So kann im Laufe der Bearbeitung der Import weiterer Informationen notwendig sein, oder es können auch Teilergebnisse vor Ende der Bearbeitung exportiert werden. Dies bezieht sich zunächst nur auf den internen zeitlichen Ablauf. Dieser Prozess hat trotzdem einen Anfang und ein Ende. Er benötigt bestimmte Eingaben und hat bestimmte Ausgaben. Da Ausgaben und Eingaben jedoch während des gesamten Verlaufs stattfinden können, ist es theoretisch möglich, dass eine Ausgabe dieses Prozesses einen direkten oder indirekten Einfluss auf eine Eingabe hat bzw. selbst als Eingabe verwendet wird. In diesem Fall handelt es sich um eine Interaktion während des Verlaufs.

Vereinfacht gesehen, ist der Import eine Beschaffung und Integration von benötigten Informationen, der Export eine Vorbereitung und Veröffentlichung von Ergebnissen und die Bearbeitung dient der eigentlichen inhaltlichen Arbeit. Diese Sichtweise ist weniger für die Betrachtung der internen Abläufe hilfreich, sondern für die Einordnung der Prozesse und ihrer Beziehung zu der Umgebung. Dazu gehört insbesondere die Kopplung mit anderen Prozessen. Man kann damit die verschiedenen Schritte einer Entwicklungsarbeit, die in unterschiedlichen Kontexten durchgeführt werden (z.B. verschiedene Personen, Organisationen oder Werkzeuge), durch eine Verkettung von Prozessdomänen abbilden. Dies lässt sich dann sogar bis zur eigentlichen Nutzung fortführen, in der dann jedoch die Entwicklungsprozesse in der Infrastruktur-Prozess-Umgebung stattfinden und nicht mehr in der Prozess-Umgebung (vgl. Abb. 4.2).

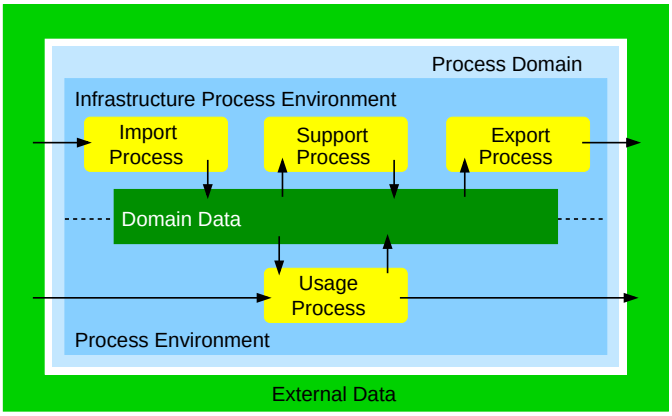

Abbildung 4.2: Aufbau der Prozessdomäne im Nutzungsbereich

4.3 Aufgabenteilung

4.3.1 Motivation und Probleme

Ein grundlegendes Problem in der Softwareentwicklung ist die Frage, wie eine Aufgabe aufgeteilt werden soll, die aufgrund ihres Umfangs nur schwer als Ganzes zu bewältigen ist. Eine Voraussetzung dafür ist eine entsprechende Größe des Problems, die eine Aufteilung in mehrere kleinere Pakete sinnvoll oder sogar notwendig macht. Diese Aufteilung kann unterschiedlich motiviert sein:

Kontrolle Durch Festlegung von Aufgaben-Paketen und Abschätzung ihres Aufwands, kann der Fortschritt der Arbeit gemessen, kontrolliert und gesteuert werden.

Beschleunigung Gelingt es voneinander unabhängige Teilaufgaben zu definieren, können diese von verschiedenen Teams parallel bearbeitet werden. Auf diese Weise kann die Entwicklungszeit reduziert und das Projekt damit beschleunigt werden.

Koordination Fertige Teilergebnisse können zur Abstimmung mit externen Prozessen genutzt werden, die nicht Teil des Projekts sind.

Komplexitätsreduzierung Durch die Verkleinerung des zu berücksichtigenden Bereichs bei den Teilaufgaben werden überschaubare Arbeitspakete geschaffen, die weit weniger komplex und damit leichter zu bearbeiten sind.

Umfasst die Aufteilung mehrere voneinander unabhängige Arbeiten, können diese einfach getrennt voneinander durchgeführt werden. Dies ist jedoch nicht so leicht, wenn der gesamten Aufgabe ein gemeinsamer Kontext zugrunde liegt, der die einzelnen Arbeiten und/oder ihre Ergebnisse voneinander abhängig macht. Zerlegt man eine solche Aufgabe in mehrere Teile, die dann getrennt bearbeitet werden, entstehen daraus folgende Probleme:

Konsistenz Aufgrund der Trennung der Abläufe können sich unterschiedliche Vorstellungen der Sachverhalte entwickeln, die zu Ergebnissen führen, die nicht zusammenpassen.

Erfahrung Während der Entwicklung wird laufend Wissen zu der gestellten Aufgabe, ihrem Kontext und möglichen Lösungen gesammelt. Das neue Wissen stellt frühere Ergebnisse in Frage und kann ein Überdenken der Lösungsstrategien bewirken. Die Aneignung der relevanten Informationen gestaltet sich bei einer Aufteilung der Aufgaben (insbesondere bei einer Sequenz) deutlich schwieriger.

Zusatzaufwand Das Aufteilen einer größeren Arbeit in kleinere, sinnvolle Teile, die nachher zusammenpassen, erfordert zusätzliche Tätigkeiten (z.B. Koordination).

4.3.2 Einbindung der Teilaufgaben

Teilaufgaben werden bei der Bearbeitung durch Subprozesse abgearbeitet, die eine gewisse Autonomie besitzen und daher einen Einbindungsmechanismus benötigen, der sie an den Hauptprozess ankoppelt. Diesen kann man in vier Aktivitäten unterteilen:

Definition Es wird festgelegt, welche Teilaufgabe im Rahmen eines Subprozesses bearbeitet werden soll.

Verteilung Das entsprechende Arbeitspaket wird von einem Entwickler übernommen bzw. ihm zugewiesen.

Bearbeitung Die gestellte Teilaufgabe wird von den zuständigen Entwicklern mit einer gewissen Unabhängigkeit vom Hauptprozess und anderen Subprozessen bearbeitet.

Integration Das Ergebnis des Subprozesses wird in den Hauptprozess integriert.

Setzt man diese Aktivitäten in Bezug zu dem Einheitsprozess in Abschnitt 4.2, dann können Verteilung, Bearbeitung und Integration als Teil des Realisierungsprozesses angesehen werden. Während die Definition eher im Bereich der Anforderungsanalyse und der Konzeption anzusiedeln ist.

Bei der Initiierung von Teilaufgaben sind grundsätzlich zwei Methoden denkbar:

zentral Eine zentrale Instanz zerlegt die Gesamtaufgabe in einzelne Arbeitspakete, die dann an verfügbare Entwickler verteilt oder von ihnen belegt werden.

dezentral Die verfügbaren Entwickler wählen sich Elemente aus der Gesamtaufgabe aus, die sie dann bearbeiten.

Ist die Bearbeitung einer Teilaufgabe abgeschlossen, so muss das Ergebnis wieder in den Hauptprozess integriert werden. Aufgrund der begrenzten Unabhängigkeit kann es Unstimmigkeiten zwischen den Vorstellungen der Beteiligten des Hauptprozesses und den Ergebnissen geben (Inhalt, Architektur, Form, Qualität, etc.). Beim Umgang mit diesen Differenzen sind nun unterschiedliche Möglichkeiten denkbar:

Nachbesserung Die Ergebnisse müssen so lange durch den Subprozess überarbeitet werden, bis sie den geforderten Ansprüchen genügen.

Toleranz Trotz der festgestellten Unzulänglichkeiten, werden die Resultate aufgenommen und evtl. notwendige Änderungen und Konsequenzen von dem Hauptprozess getragen.

Selektion Präsentierte Ergebnisse werden als Angebote verstanden, die angenommen werden, wenn sie einen Fortschritt in Richtung der Zielsetzung des Hauptprozesses bedeuten und keine bessere Option in Aussicht ist. Eine Ablehnung kann mit Nachbesserungsvorschlägen und anderen Einschätzungen verbunden sein, die für eine Überarbeitung und anschließende Akzeptanz der Ergebnisse hilfreich sein können.

4.3.3 Schaffung von Schnittstellen

Das natürliche Entstehen und bloße Erkennen von völlig unabhängigen Aufgaben ist in einem Softwareprojekt eher die Ausnahme, stellt jedoch den Idealfall dar, da man sich damit den hohen Aufwand für Koordination und Zusammenführung der Ergebnisse sparen kann. Für diesen Zweck werden sie daher künstlich erzeugt, in dem man Sachverhalte festlegt und damit die wechselseitigen Abhängigkeiten auf Elemente höherer Ordnung verschiebt. Dadurch ist eine Aufgabe nicht mehr von einer anderen abhängig, sondern nur noch von den festgeschriebenen Randbedingungen, die man als Schnittstellen bezeichnen kann. Schnittstellen regulieren damit Berührungspunkte zwischen zwei Elementen. Bei diesen Elementen kann es sich um Entwicklungsprozesse oder auch ihre Ergebnisse handeln.

Die Berührungspunkte existieren unabhängig von den Schnittstellen. Sie ergeben sich aus dem Kontext bzw. der eigenen Funktionalität der Elemente. Ändert sich eines von beiden können sich auch die Berührungspunkte ändern. Legt man nun jedoch eine verbindliche Schnittstelle fest, definiert man damit gleichzeitig auch das was jeweils auf den beiden Seiten davon ist. Ähnlich einem Vertrag regelt die Schnittstelle die Rechte und Pflichten der beteiligten Elemente. Man kann daher von einer Schnittstellenspezifikation sprechen, die dann genau das besagte Element höherer Ordnung darstellt. Auf diese Weise wird von dem realen Kontext abstrahiert und durch die Spezifikation ersetzt. Da jedoch das Ergebnis später im tatsächlichen Kontext eingesetzt wird, ist sie nicht frei definierbar, sondern muss ein passendes Modell der Vorgänge sein. Dabei ist zu berücksichtigen, dass jeweils das eine Element Bestandteil vom Kontext des anderen ist.

Schnittstellen sind daher nicht frei wählbar, sondern müssen sich an der Aufgabenstellung, dem Kontext, den Lösungsansätzen für die Bewältigung der Teilaufgaben und der Struktur ihrer Ergebnisse orientieren. Auch wenn gewisse Abweichungen von einer optimalen Schnittstelle tolerierbar sind, verursacht dies bereits zusätzlichen Aufwand. Im Extremfall macht eine unpassende Schnittstelle das zugehörige Element nicht ausreichend zugänglich und damit die Erstellung einer redundanten Funktionalität notwendig. Es ist daher wichtig eine passende Schnittstelle zu definieren und sie nicht willkürlich zu wählen.

Eine solche Spezifikation wäre daher am leichtesten nach Vollendung der Elemente zu erstellen, könnte aber dann nicht mehr der Entkopplung der ja bereits abgeschlossenen Entwicklungsarbeit dienen. Wird sie vor Beginn der Entwicklung formuliert, beschreibt sie teilweise etwas, was noch gar nicht da ist: die zu erstellenden Elemente. Es besteht daher eine wechselseitige Abhängigkeit zwischen Schnittstellen und dadurch verbundenen Elementen. Dieses Problem lässt sich nur auflösen, wenn man die Schnittstellenspezifikation als Teil des Bauplans für die beteiligten Elemente ansieht und damit als Teil der Aufgabenstellung versteht. Um so detailreicher eine solche Schnittstellenspezifikation ist und um so mehr sie die Internas der Elemente festlegt, um so mehr stellt ihre Erstellung eine Entwicklungsarbeit auf einer höheren Ebene dar.

4.3.4 Gruppierungen von Teilaufgaben

Ein wesentlicher Aspekt bei Teilaufgaben ist ihr Verhältnis zueinander, was primär durch ihre Abhängigkeiten voneinander geprägt ist. Daraus ergeben sich drei mögliche Gruppierungen:

Sammlung Damit wird hier eine Menge von Aufgaben bezeichnet, die voneinander unabhängig bearbeitet werden können. Gleichgültig ob dies durch entsprechende Schnittstellen oder durch eine natürliche Trennung entsteht.

Sequenz Eine Sequenz zeichnet sich durch das notwendige Aufeinanderfolgen der Bearbeitungsprozesse aus, weil das Ergebnis des einen Prozesses als Eingabe für den folgenden Prozess dient.

Cluster Ein Cluster ist eine Menge von Teilaufgaben, die ein gewisses Maß an Unabhängigkeit voneinander besitzen, jedoch trotzdem in einer gewissen Wechselwirkung stehen, die eine Interaktion sinnvoll oder sogar notwendig machen, d.h. die Lösung der einen erfordert auch die Lösung der andern Aufgabe. Daher werden sie parallel und interaktiv bearbeitet.

4.3.5 Iteratives Arbeiten

Man kann stets versuchen eine gegebene Aufgabe bereits beim ersten Anlauf perfekt zu lösen. Aufgrund der Komplexität und minimalen Fehlertoleranz gelingt dies bei der Erstellung von Software jedoch i.d.R. nicht und eine nachfolgende Überarbeitung wird notwendig. Zwar gibt es Techniken (z.B. formale Spezifikationen, Verifikation), die einen bei diesem Versuch unterstützen, aber ihre Verwendung erhöht den Aufwand erheblich. Zudem können nicht alle Fehler damit erkannt bzw. verhindert werden, z.B. konzeptionelle Fehler oder eine fehlerhafte Vorstellung vom späteren Anwendungsszenario sind nur schwer vor einem Test oder sogar der tatsächlichen Nutzung zu erkennen. Eine Garantie für echte Fehlerfreiheit lässt sich daher nicht erreichen, sondern nur eine gewisse Korrektheit und Konsistenz. Dazu kommt die stete Veränderung der Welt, die eben oftmals auch zu einer Änderung des zugehörigen Kontexts und damit zu weiteren Unzulänglichkeiten führt.

Letztendlich sollte man davon ausgehen, dass Software niemals perfekt, sondern mit Fehlern und Unzulänglichkeiten behaftet ist, selbst wenn in bestimmten Bereichen für eine gewisse Zeitspanne mit einem entsprechenden Aufwand ein hoher Grad an Perfektion erreicht werden kann. Hinzu kommt der Funktionsumfang einer Software, deren Erweiterung die Komplexität, die Fehleranfälligkeit und damit auch den Aufwand erheblich erhöht. Man muss daher immer abwägen zwischen Umfang, Perfektion und Aufwand.

Konkrete Entwicklungsvorhaben haben nun i.d.R. für Umfang und Perfektion gewisse Mindestanforderungen und für die verfügbaren Mittel (Aufwand) bestimmte Obergrenzen. Zudem sind unterschiedliche Prioritäten bezüglich dieser drei Faktoren denkbar. Bei der einen Software soll eine maximale Perfektion des fest definierten Funktionsumfangs im Rahmen des möglichen Budgets (Aufwand) erreicht werden. In einem anderen Fall sind Qualitätskriterien (Perfektion) und verfügbare Mittel (Aufwand) festgelegt und ein Nutzungsprozess soll in diesen Grenzen möglichst umfangreich unterstützt werden (Umfang).

Es stellt sich dabei stets die Frage wie viel Arbeit man in welche Tätigkeit investiert, um die gewünschten Ziele zu erreichen. Sind nun alle Faktoren bekannt, die auf den Entwicklungsvorgang einen wesentlichen Einfluss ausüben, und ihre maßgebliche Änderung kann sicher ausgeschlossen werden, so kann man tatsächlich den gesamten Verlauf entsprechend den vorgegebenen Prioritäten präzise im Voraus planen. Empirische Untersuchungen in der Praxis haben jedoch gezeigt, dass die erstellte Planung in vielen Fällen nicht eingehalten werden kann. Die Erfolgsquote der weltweiten Projekte, die von der Standish Group 2003 untersucht wurden, war 34% im Vergleich zu 16% im Jahre 1994 [ChaosReport 03]. In der SUCCESS-Studie, die sich auf Projekte in Deutschland beschränkte, wurde eine Erfolgsquote von 50,7% festgestellt [Buschermöhle 06]. Man sollte daher grundsätzlich bei der Softwareentwicklung mit unerwarteten Problemen rechnen und sich nicht allein auf die gemachte Planung verlassen.

Ein sicherer Weg, um verdeckte Probleme zu erkennen, ist entsprechende Arbeiten durchzuführen. Um so besser man den entsprechenden Kontext kennt und um so klarer die Gesamtstruktur der zu erstellenden Lösung ist, um so unwahrscheinlicher ist das Auftauchen von neuen Problemen, die bisherige Ergebnisse in Frage stellen. Auch wenn nur Teile der notwendigen Arbeiten gemacht werden, können damit wesentliche Unsicherheitsfaktoren beseitigt werden. Es lassen sich aufgrund des Verhältnisses zum Endergebnis drei Varianten dieses explorativen Arbeitens unterscheiden:

Prototyp Die Software wird nur provisorisch erstellt. Dabei werden Aspekte bewusst vernachlässigt, die erst in einem späteren Zyklus nachgeholt werden. Dabei kann es sich z.B. um die innere Struktur, Benutzeroberfläche, Fehlerbehandlung oder ähnliches handeln.

Komponente Es wird ein Teil der angestrebten Funktionalität ausgewählt und in einem Durchlauf erstellt.

Konzept Anstatt die Software selbst zu erstellen, werden Konzepte entworfen oder auch getestet, die später für die endgültige Erstellung der Software verwendet werden.

Diese Varianten lassen sich auch kombinieren. So kann das Ziel eines Iterationsschritts z.B. das Konzept zu einem Prototypen von einer Komponente sein.

Man kann iteratives Vorgehen auch in Phasen aufteilen, die eine funktionale Änderung der einzelnen Zyklen in den jeweiligen Phasen kennzeichnet. Das folgende beschreibt eine derartige Einteilung in drei Phasen:

Exploration Die Aufgabe, der zugehörige Kontext und die zu erstellende Software werden erkundet und auf diese Weise notwendigen Kenntnisse für die weitere Entwicklung gewonnen.

Konstruktion Es werden mögliche Strukturen, Strategien, Lösungsvarianten, Bauelemente, etc. entworfen, gesammelt, getestet oder gebaut.

Perfektion Bereits vorhandene Ergebnisse werden verbessert, überarbeitet oder vervollständigt.

Wird bei einer Iteration für jeden Durchlauf angegeben, was darin stattfinden soll, ist es keine echte Iteration mehr, sondern eine Sequenz mit ähnlichen Teilaufgaben. Eine echte Iteration ist die Wiederholung ohne klare inhaltliche Vorgaben, die bei der tatsächlichen Ausführung Freiräume für adaptives Verhalten lässt. Insgesamt kann man sagen, dass ein iteratives Vorgehen dem Verlauf mehr Freiheit bei der Arbeit gibt, da es zwar gewisse Strukturen vorgibt, aber einen flexiblen Umgang mit neuen Erkenntnissen ermöglicht, die nach Abschluss der Planung gewonnen wurden.

4.3.6 Konkurrierendes Vorgehen

Innerhalb eines Projekts ist es eher unüblich (Teil-)Aufgaben mehrfach bearbeiten zu lassen, da man letztendlich nur ein Ergebnis braucht und damit die Arbeit für die anderen Lösungen am Ende evtl. umsonst sind, wenn man nicht in einem weiteren Schritt versucht, aus den vorhandenen Alternativen die besten Elemente zu extrahieren und daraus eine gemischten Lösung zu bauen. Doch selbst wenn man die alternativen Ergebnisse nicht weiterverwendet, kann in Situation, in denen das richtige Vorgehen nicht klar erkennbar ist, das parallele Verfolgen von alternativen Ansätzen durch eine derartige Überproduktion und die anschließende Selektion gewisse Vorteile bringen, die evtl. den Zusatzaufwand rechtfertigen. Im schlechtesten Fall, wäre bei einer einfachen Entwicklung die ungünstigste Variante aus der Menge der erstellten Alternativen entstanden. Im Vergleich mit diesem ungünstigsten Szenario einer eingleisigen Entwicklung hat eine konkurrierendes Vorgehen folgende Vorteile:

Zeit Entwickelt man nur eine Lösung für ein bestimmtes Problem und ist anschließend nicht mit dem Ergebnis zufrieden, benötigt man zusätzliche Zeit für die Erstellung einer Alternative, die man sich evtl. sparen kann, wenn man von Anfang an mehrere Ansätze verfolgt.

Qualität Sind alle verfügbaren Alternativen akzeptabel, bleibt immer noch der Vorteil, dass man sich für die beste Variante entscheiden kann, was einen gewissen Qualitätsvorteil bringt.

Zuverlässigkeit Stellt sich ein verfolgter Lösungsansatz unerwartet als nicht machbar bzw. sinnvoll heraus, führt dies zu keinen größeren Problemen, da es immer noch die konkurrierenden Alternativen gibt. Sollten alle verfolgten Ansätze nicht realisierbar sein, hat man evtl. auf diese Weise eine Menge Zeit gespart und viel Wissen für weitere Versuche gesammelt.

Vielfalt Durch das zulassen von alternativen Bestrebungen wird in Bezug auf die Aufgabenstellung und ihren Kontext ein Vielfalt an Wissen und Konzepten erzeugt, die grundsätzlich bei der Entwicklung und Überarbeitung von Vorteil ist.

Diese Vorteile beziehen sich primär auf den Entwicklungsgegenstand, aber auch in Bezug auf die beteiligten Entwickler können redundante Teams ähnliche Vorzüge haben. So führt der Ausfall von einzelnen Entwicklern oder sogar eines ganzen Teams nicht zwangsläufig zu maßgeblichen Verzögerungen oder Ausfällen.

4.3.7 Schrittgröße

In den vorangegangen Abschnitten wurde aufgezeigt wie Arbeitsschritte strukturiert sind und wie sie aufgeteilt werden können. Bisher wurde jedoch noch nicht die Größe von Teilaufgaben und die Konsequenz daraus betrachtet. Hier soll der Einfluss dieses Aspekts auf die Entwicklungsarbeit betrachtet werden, daher ist es wesentlich, was die teilnehmenden Akteure und nicht etwa außenstehende Beobachter unter der "Größe eines Arbeitsschritts" verstehen, denn nur das hat einen wesentlichen Effekt.

Da eine derartige Einschätzung nur relativ zu der zugehörigen Umgebung gesehen werden kann, ist es wichtig sich auf eine Granularitätsebenen festzulegen, die man betrachten will. Gilt es eine Gesamtaufgabe zu bewältigen, stellt sich die Frage, in wie viele Teile sie zerlegt wird, z.B. in zwei, zehn oder hundert. Eine weitere interne Zerlegung während der Bearbeitung dieser Teile ist davon unberührt, da sie auf einer anderen Granularitätsebene stattfindet.

Die Grenze zwischen der Bearbeitung der Hauptaufgabe und der Teilaufgaben kann nun unterschiedlich festgelegt sein und die einbindenden Aktivitäten (z.B. Koordination, Integration) können zentral durch den Hauptprozess oder dezentral durch die Subprozesse abgedeckt werden. In beiden Fällen steigen jedoch diese Aktivitäten, um so größer die Zahl der Teile wird. Damit steigt auch ihre Sichtbarkeit und Kontrollierbarkeit. Zudem können leichter Kurskorrekturen durchgeführt werden, ohne dabei in die Autonomie der Teilprozesse einzugreifen. Grundlage für solche Korrekturen können wiederum Ergebnisse aus bereits abgeschlossenen Teilprozessen sein.

Die Teilprozesse besitzen eine gewisse Autonomie, d.h. der Hauptprozess greift nur beschränkt in ihren Ablauf ein. Wird die Anzahl der Teilaufgaben erhöht bzw. die Schrittgröße verkleinert, verkürzen sich die Intervalle, in denen die Subprozesse laufen und der Hauptprozess keine Eingriffsmöglichkeit besitzt. Dies führt dann indirekt zu einer Erhöhung des Einflusses des Hauptprozesses auf die Subprozesse.

4.3.8 Parallelisierung der Entwicklungsaktivitäten

In Abschnitt 4.2 wurde der Einheitsprozess mit seinen vier Phasen dargestellt und mit der funktionalen Abhängigkeit ihre notwendige Reihenfolge begründet. Durch die in Abschnitt 4.3 dargestellt Aufgabenteilung ist es jedoch mit einer Iteration oder Sequenzbildung möglich abschnittsweise vorzugehen und damit zwischenzeitlich Kurskorrekturen und eine dadurch bedingte Änderung der Aufgabenstellung in den Teilschritten zu erreichen. Obwohl also der Einheitsprozess in seiner Struktur erhalten bleibt, kann man durch eine entsprechend kleine Schrittgröße (vgl. Abschnitt 4.3.7) die Aktivitäten nahezu Parallelisieren wie es Abbildung 4.3 zeigt (1:Analyse, 2:Konzeption, 3:Umsetzung, 4:Ausgabe).

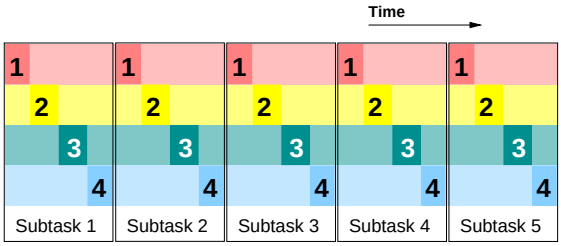

Abbildung 4.3: Parallelisierung der Entwicklungsaktivitäten durch kleine Teilaufgaben

Durch die Parallelisierung können die Aktivitäten interaktiv durchgeführt werden, d.h. Erfahrungen, Probleme und Ergebnisse aus der einen können nicht nur für die nachfolgenden, sondern für alle Aktivitäten genutzt werden. Handelt es sich um einen komplexen, unbekannten Kontext kann dies erheblich Vorteile haben.

So kann die relativ aufwendige Erfassung des Kontexts im Rahmen der Anforderungsanalyse durch die Kopplung mit den anderen Aktivitäten auf den Teil beschränkt werden der wirklich für eine Lösung notwendig ist und auch Korrekturen führen zu keinem erheblichen Mehraufwand.

Findet im Rahmen dieser Vorgänge auch eine gewisse Verteilung statt, d.h. die Ausgabe dient der Überführung in ein neues Kontextmodell (vgl. Kapitel 6), in dem mehr über die konkrete Umgebung bekannt ist, so kann man auf diese Weise die Software schrittweise spezialisieren. Dies bringt besondere Vorteile, wenn man die Software parallel auf mehrere unterschiedliche Kontextmodelle mit dem selben Abstraktionsgrad spezialisieren möchte.

4.4 Wartung: Entwicklung in der Nutzungsphase

4.4.1 Stufenmodell der Softwareentwicklung

Softwareentwicklung endet in den meisten Fällen nicht mit der ersten Auslieferung an den Endnutzer. Es gibt immer den Bedarf nach Veränderungen. Letztendlich beginnt die Softwareentwicklung mit der ersten Idee und endet mit der Stilllegung auf dem letzten Rechner, auf dem sie eingesetzt wurde. Dabei handelt es sich jedoch nicht um einen fortlaufenden Prozess, sondern um voneinander weitgehend unabhängige Entwicklungsschritte, denen jeweils die Verwendung der Ergebnisse folgt, die sie trennen. Bennett/Rajlich unterteilen diesen Verlauf in fünf Etappen und nennen diesen Verlauf das einfache Stufenmodell (simple stage model) [Rajlich 00]:

Initiale Entwicklung (initial development) In dieser Phase wird die Software neu entwickelt, um die anfänglichen Anforderungen zu erfüllen. Ein wesentlicher Teil dieser Arbeit ist das Erforschen und Verstehen des Problembereichs und dem zugehörigen Kontext. Zudem müssen grundlegende Entscheidungen getroffen werden, die sich in der Systemarchitektur widerspiegeln und eine zentrale Bedeutung für die Effizienz der späteren Etappen haben.

Evolution (evolution) Im Anschluss beginnt eine Etappe der iterativen Entwicklung, in der es fortlaufende Anpassungen der Funktionalität gibt, die u.a. durch den durchlaufenen Lernprozess, veränderte Anforderungen und Konkurrenzdruck entstehen. Evtl. beginnt diese Phase bereits vor der ersten Auslieferung z.B. zur Erreichung von bestimmten Qualitätsstandards. Sie kann aber auch während der Nutzung im Rahmen von Wartungsarbeiten fortgesetzt werden.

Instandhaltung (servicing) Für eine effektive Überarbeitung der Software ist eine passende Architektur und ein ausreichend befähigtes und großes Team notwendig. Ist eine dieser Anforderungen nicht mehr gegeben, werden Änderungen aufwendig, ineffizient und behelfsmäßig. Dies reduziert die Qualität und die zukünftige Wartbarkeit immer weiter. Im Grunde handelt es sich dabei um den Anfang vom Ende einer Software.

Abwicklung (phaseout) Werden die aktiven Wartungsarbeiten zu einem Programm seitens der eigentlichen Entwickler eingestellt, müssen die Nutzer der Software selbstständig Wege finden, um Probleme zu beheben, in dem sie z.B. entsprechende Änderungen an der Umgebung vornehmen bzw. sie künstlich konstant halten. Das Programm wird in dieser Phase nur noch ohne weitere Wartung zur Verfügung gestellt.

Stilllegung (closedown) Nach einer Weile werden die Probleme zu zahlreich und folgenschwer. Die Software wird für "unbrauchbar" erklärt, nicht mehr weiter zur Verfügung gestellt und von einer weiteren Nutzung abgeraten. Falls vorhanden, wird das Ausweichen auf ein Folgesystem empfohlen und dazu evtl. entsprechende Migrationsstrategien bzw. -hilfen angeboten.

Bennett/Rajlich haben diese Aufteilung auch auf einen Ablauf übertragen, in dem immer wieder aus der Evolutionsphase Versionen abgezweigt werden, die keine größeren Veränderungen mehr erfahren und dann die besagten Phasen durchlaufen. Währenddessen läuft die Evolution als paralleler Entwicklungszweig weiter. Diese Aufteilung nennen sie das versionierte Stufenmodell (versioned stage model), das in Abb. 4.4 gezeigt ist.

In beiden Modellen werden in den ersten drei Phasen (initiale Entwicklung, Evolution, Instandhaltung) Entwicklungsarbeiten durchgeführt, die grob dem Schema aus Abschnitt 4.1.2 folgen. Bei den initialen Arbeiten handelt es sich um ein großes Entwicklungsprojekt während Evolution und Instandhaltung iterativen Charakter haben.

4.4.2 Verhältnis der Entwicklung zur Wartung

Wie am Ende von Abschnitt 4.1.1 beschrieben, ordnet ISO12207 die Erstellung von Änderungen im Rahmen von Wartungsarbeiten als einen Entwicklungsprozess ein (s. a. [ISO14764 06]). Auch Kitchenham et al. sehen Wartung als eine besondere Form der Entwicklung [Schneidewind 99, Kitchenham 99]. Sie unterscheiden zwei Szenarien:

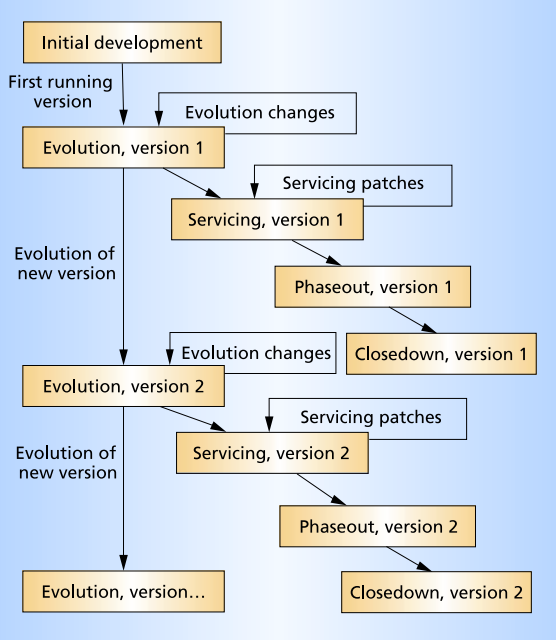

Abbildung 4.4: Versioniertes Stufenmodell eines Lebenszyklus (Quelle: [Rajlich 00])

Evolutionäre Entwicklung In diesem Fall sind die selben Entwickler sowohl für die Korrektur von Mängeln in alten Versionen wie auch für die Entwicklung von neuen Versionen zuständig. Es wird hier implizit von einer ähnlichen Struktur ausgegangen, wie sie in dem versionierte Stufenmodell dargestellt wurde (vgl. Abschnitt 4.4.1). Damit ist die Wartung hier eine natürliche Fortsetzung der initialen Entwicklungsarbeit.

Unabhängige Wartungsgruppe Wartungsarbeiten werden von einem unabhängigen Team übernommen, das an der ursprünglichen Entwicklung nicht beteiligt war, keine tiefgreifenden Änderungen mehr vornehmen kann bzw. soll und nur für einen akzeptablen Betrieb sorgt. Es handelt sich dabei um eine typische Instandhaltung im Sinne von Abschnitt 4.4.1. In diesem Fall ist jede Änderung eher als ein eigenständiges, kleines Entwicklungsvorhaben zu sehen, das die Beseitigung der entsprechenden Problematik zum Ziel hat.

In beiden Fällen handelt es sich jedoch um Entwicklungsarbeiten. Trotzdem bedarf nicht jeder Wartungsfall auch einer Änderung der Software. Manchmal sind Modifikationen an dem zugehörigen Softwaresystem z.B. in Form von Konfigurationsänderungen oder Kontextanpassungen eine zufriedenstellende und weniger aufwendige Lösung.

Obwohl es sich also wie beschrieben bei der Wartung in vielen Fällen um Entwicklungsarbeit handelt, gibt es trotzdem eine Besonderheit, die stets beachtet werden sollte: Die zu überarbeitende Software ist bereits im Einsatz. Dadurch müssen die aktuellen Systeme bei der Anforderungsanalyse und Verteilung mit einbezogen werden.

In Bezug auf die Verteilung erfordert dies eine Migrationsstrategie, die erstellte Änderungen auf vorhandene Softwaresysteme mit möglichst wenig Aufwand und Problemen überträgt.

Da i.d.R. Software nicht nur auf einem System eingesetzt wird, muss bei der Anforderungsanalyse zunächst geklärt werden, ob das betreffende Anliegen ein allgemeines oder spezielles Pro-

blem ist und ob die zugehörige Anpassung und ihre Folgen bei den anderen Systemen zumindest toleriert werden kann. Je nachdem ist es sinnvoller das Anliegen allgemeine oder individuelle zu lösen. In beiden Fällen wird die Erfassung der Anforderungen im Vergleich zur initialen Entwicklung vereinfacht, da man sich auf die gegenwärtige Version der Software beziehen und sie dadurch in Form von kleinen Änderungsbeschreibungen übermitteln kann, statt eine vollständige Beschreibung zu verfassen. Dadurch wird auch das Risiko geringer, an den Bedürfnissen der Nutzer "vorbei zu entwickeln".

4.4.3 Wartung in der Praxis

Wartungsarbeiten sind bezüglich der Gesamtkosten, die im Lebenszyklus einer Software anfallen, der größte Kostenfaktor. Die Zahlen variieren je nach Untersuchung zwischen 50 und 90 Prozent[Koskine 04]. Da sie sich über den gesamten Nutzungszeitraum erstreckt, hat sie auch zeitlich gesehen den größten Anteil.

Man unterscheidet vier Kategorien von Wartungsarbeiten wie sie in Tabelle 4.1 beschrieben sind [Abran 04, S. 6-3].

	Korrektur	Verbesserung
proaktiv	preventive: Beseitigung potentieller Fehlerquellen	perfective: Verbesserung von Wartbarkeit und Leistungsfähigkeit
reaktiv	corrective: Korrektur von entdeckten Problemen	adaptive: Anpassung an veränderte Umgebung

Tabelle 4.1: Einteilung der Wartungsarbeiten

Entgegen der allgemeinen Auffassung haben Untersuchung gezeigt, dass 80% der Wartungsarbeiten "non-corrective" sind und damit keine aufgetretenen Probleme beheben, sondern präventive Maßnahmen oder Verbesserungen darstellen [Abran 04, S. 6-3]. Dies ist nicht verwunderlich, bedenkt man Lehmanns erstes Gesetz: ein System in Nutzung wird verändert werden [Endres 03, S. 163]. Studien belegen diese Theorie in Bezug auf die Anforderungen, die eine durchschnittliche Änderungsrate von 2% monatlich haben [Jones 98] und auch durch den Lernvorgang während der Entwicklung wird eine Änderung der Funktionalitätsanforderungen von 30% angenommen [Rajlich 00]. Ein großer Anteil der Wartung ist damit auch im Bereich der proprietären Software evolutionäre Entwicklungsarbeit.

Da es sich hauptsächlich um *keine* Fehlerkorrekturen handelt, hat das Unterlassen oder das Aufschieben dieser Arbeiten normalerweise keinen katastrophalen Folgen, reduziert aber die Benutzbarkeit, Effizienz und Produktivität der Software, was ebenfalls erhebliche Kosten nach sich zieht, wenn man den Aufwand einbezieht, diese Unzulänglichkeiten auszugleichen. Der Nutzer hat daher abzuwägen zwischen Wartungsarbeiten einschließlich den damit verbunden Kosten einerseits und den Problemen des gegenwärtigen Zustands und dem damit verbundenen Kompensation-Aufwand andererseits. Letztlich ist es damit eine Kosten-Nutzen-Rechnung. Wartung kann entweder mit den entsprechenden Kenntnissen und Lizenzen durch einen selbst bzw. eigenes Personal durchgeführt oder von extern als Dienstleistung eingekauft werden.

4.4.4 Verlagerung von Entwicklungsarbeiten in die Wartung

Die vorangegangen Abschnitte haben gezeigt, dass die Wartung eine Möglichkeit ist die Entwicklungsarbeit nach der initialen Auslieferung fortzuführen. Man kann sich damit zunächst für anstehende Teilaufgaben entscheiden, ob man sie bei der initialen Entwicklung oder später während der Wartung bearbeitet. Damit muss jedoch nicht mehr der Abschluss der Entwicklung, die Vollständigkeit der Funktionalität und die Erfüllung von Anforderungen notwendigerweise bis zur Auslieferung erfüllt sein und der Auslieferungstermin kann bis zu einem gewissen Grad von der Entwicklungsarbeit entkoppelt werden. Dies nenne ich das *Evolutionsprinzip*. Es basiert auf der Auslieferung und Verwendung einer provisorischen Lösung und ihrer schrittweisen Verbesserung im Rahmen eines umfangreichen Wartungsprozesses.

Auch wenn die Entwicklungsmethodik selbst keine endgültige Fertigstellung bei der Auslieferung fordert, kann dies durchaus durch andere Umstände erzwungen werden. So kann es einen entsprechenden Vertrag zwischen Auftraggeber und Entwickler geben, der die perfekte Fertigstellung bei Auslieferung verlangt. Eine anderer Hinderungsgrund kann die Intoleranz des Anwendungssystem sein, das eine Nutzung der Software nur erlaubt, wenn es alle Anforderungen exakt erfüllt. Dies gilt z.B. für Steuerungssoftware in der Luft- und Raumfahrt oder in der Medizintechnik. Bei derartigen Anwendungen sollte es eigentlich überhaupt keine Wartung geben, da Fehler inakzeptable Konsequenzen haben und daher auch ein entsprechend hoher Aufwand bei Analyse, Kontrolle und Endprüfung betrieben wird, um Fehler zu vermeiden, der zudem bei jeder Wartungsarbeit wiederholt werden müsste. Es gibt also eine Menge Software, die mit Auslieferung eine maximal Perfektion besitzen soll und eine minimale Wartung angestrebt wird. Dies nenne ich das *Perfektionsprinzip*. Es ist gekennzeichnet durch die Forderung nach einer möglichst perfekten Lösung vor der ersten Nutzung und der Minimierung der anschließenden Wartung bzw. ihr gänzliches Fehlen.

Die Entscheidung für eines der beiden Prinzipien hat erhebliche Auswirkungen auf die in Abschnitt 4.1.2 beschriebenen Phasen. Kombiniert man sie mit der Wartung ergeben sich die in Abbildung 4.5 veranschaulichten Unterschiede. Dabei sind die unterschiedlichen Phasen farblich voneinander abgegrenzt und durch Nummern gekennzeichnet (1:Anforderungsanalyse, 2:Design 3:Realisierung, 4:Auslieferung, 5:Wartung).

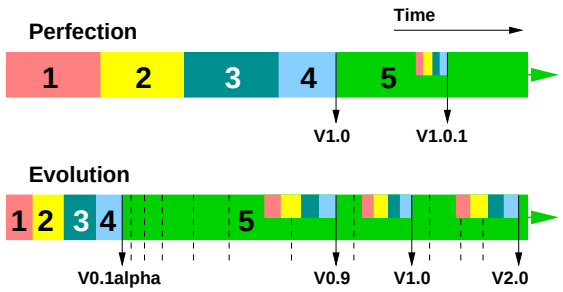

Abbildung 4.5: Vergleich Perfektions- vs. Evolutionsprinzip

Die Erfahrung zeigt: eine Systemevolution durch Wartungsarbeiten ist fast immer notwendig und vor der Wartungsphase muss es eine initiale Entwicklung geben, bevor sich etwas zunehmend verbessern kann. Für welches Prinzip man sich entscheidet, hängt von vielen Faktoren ab (rechtliche, technische, wirtschaftliche, gesellschaftliche, strategische, ...).

4.5 Modell der Open-Source-Entwicklung

4.5.1 Verlagerung der Entwicklung in die Wartung

Man kann im Open-Source-Bereich zwischen zwei Prozessen unterscheiden: die initiale Entwicklung einer ersten lauffähigen Version der Software (initialer Prototyp) und der darauf folgenden sukzessiven Weiterentwicklung der Software entsprechend dem Feedback der Benutzer, die am Anfang auch nur die Entwickler sein können (vgl. [Dietze 04, S.42]). Jeder neue Benutzer wird sich dann erstmal die Software besorgen und sie ausprobieren, bevor er daran arbeitet, Anforderungen formuliert oder etwas anderes damit macht. Sieht man von der initialen Prototypenentwicklung ab, so sind demnach *alle* Entwicklungsaktivitäten der Auslieferung *nachgelagert.* Dies widerspricht aber der in Abschnitt 4.1.1 gefunden Eingrenzung von Entwicklungsprozessen auf die Vorgänge *vor* der Auslieferung. Es handelt sich daher um Abläufe, die im klassischen Lebenszyklusmodell der Wartung zugeordnet werden und nicht dem primären Entwicklungsprozess. Diese umfasst jedoch wie dort beschrieben Mini-Entwicklungsprozesse zur Erstellung der zugehörigen Änderungen. Dies entspricht auch der üblichen Vorgehensweise im Open-Source-Bereich, da Patches i.d.R. in solchen Mini-Prozessen entwickelt und dann integriert werden. Für die Erstellung eines Patches ist wiederum wegen des geringen Umfangs und der wenigen Beteiligten keine besondere Vorgehensweise notwendig.

Bisher wurde allerdings noch nicht die Entwicklung des initialen Prototypen betrachtet. Nach Dietze wird diese “in einem geschlossen Entwicklungsprozess durchgeführt” und “unterliegt somit nicht den spezifischen Anforderungen der Softwareentwicklung durch eine OSS-Community” [Dietze 04, S. 42]. Es handelt sich dabei also *nicht* um einen besonderen Open-Source-Entwicklungsprozess.

Verbindet man diese beiden Erkenntnisse, drängt sich die These auf, dass es gar keinen besonderen Open-Source-Entwicklungsprozess gibt, wenn wir im Rahmen der üblichen Begrifflichkeit der Informatik bleiben. Betrachten wird jedoch die Vorgänge in der Open-Source-Gemeinschaft, die man üblicherweise als “Wartung” bezeichnet, wird dieser Begriff unpassend, da es sich in Wirklichkeit um die eigentliche Entwicklungsarbeit in Form eines Evolutionsprinzip handelt, wie es in Abschnitt 4.4.4 beschrieben wurde. Dies ist grundsätzlich nichts Neues, denn auch im proprietären Bereich findet in der Wartungsphase noch eine wesentliche Softwareentwicklung statt (vgl. Abs. 4.4.3). Im Open-Source-Bereich wurde jedoch die Trennung zwischen Entwicklung und Wartung fast völlig aufgehoben, da die initiale Entwicklung derart minimiert wurde, dass sie relativ gesehen zu den nachfolgenden evolutionären Vorgängen keine wesentliche Rolle mehr spielt.

Dies hat auch erhebliche Auswirkungen auf die initiale Entwicklungsarbeit. Da zunächst nur ein rudimentärer Prototyp gebaut wird, kann man sich die extrem aufwendigen Abschlussarbeiten zunächst sparen. Veranschlagt man hier die typische 80/20 Regel (die letzten 20% der Entwicklung erfordern 80% der Zeit) kann man schon alleine deswegen mit einer Reduzierung des Initialen Aufwands auf weniger als 20% des Üblichen rechnen. Zudem wird durch eine Reduzierung von begleitenden Tätigkeiten (Management, Planung, Vertragsverhandlungen, Marketing, etc.) und der damit verbunden Koordination und Kommunikation der Aufwand und die Komplexität der Abläufe auf das absolut Notwendige reduziert. Dadurch sind einerseits die geleisteten Arbeiten und damit die Wertschöpfung zunächst gering und andererseits der Prototyp für einen effizienten Einsatz noch nicht geeignet. Ist jedoch ein entsprechender Bedarf vorhanden,

reicht dies unter günstigen Umständen aus, ein erfolgreiches Open-Source-Projekt zu etablieren, das dann in einem evolutionären Prinzip entwickelt werden kann. Anderenfalls halten sich die fehl-investierten Ressourcen in Grenzen. Man kann auf diese Weise ohne die Investition von umfangreichen Mitteln den Bedarf für eine solche Software feststellen.

Sie umfasst zusätzlich auch die Auslieferung und Installation der Software, da es im Rahmen eines intensiven Wartungsprozesses regelmäßig notwendig ist, die Software durch eine neue Version zu ersetzen.

Für diese ausgedehnte Wartungsphase hat sich im Laufe der Zeit ein eigener Lebenszyklus herausgebildet, der mit dem klassischen nur noch wenig zu tun hat. Dies steht auch in Verbindung mit dem sukzessiven Vorgehen wie es z.B. Dietze beschreibt (s. [Dietze 04, S. 43]), das sich mit der frühen Auslieferung einer halbfertigen Software begründet. Die Folge davon ist das häufige Erscheinen von neuen Versionen führt. Damit bekommen dann die Auslieferungs- und Upgrade-Mechanismen eine zentrale Bedeutung für den effizienten Betrieb einer Software. Da diese Mechanismen bereits etabliert sind besteht auch kein großer Bedarf einer Änderung der Vorgänge, wenn die Software schließlich ausgereift ist.

4.5.2 Open-Source-Lebenszyklus

Wie bereits erläutert handelt es sich beim Open-Source-Lebenszyklus im klassischen Sinne um einen Wartungsvorgang, d.h. er beginnt und endet mit der Nutzung einer Open-Source-Softwarekomponente. Angestoßen wird der Vorgang durch ein konkretes Problem oder wie Raymond es nennt: „A developer's personal itch" [Raymond 99, S. 32] und mündet in einer neuen Komponentenversion. Betrachtet man ihren Weg, so lassen sich primär folgende Prozesse identifizieren, die sie durchläuft:

Erstellung (creation) Die auf einem Computersystem eingerichtete Komponente entspricht nicht den Vorstellungen des betreffenden Nutzers. Hat er ausreichende Kenntnisse und Interesse, so passt er sie nach seinen Bedürfnissen an. Durch diese Veränderungen wird eine neue, angepasste Komponente (adjusted component: aCP) erstellt. Da sie sich jedoch nur in einem kleinen Teil von der ursprünglichen Version unterscheidet, wird normalerweise nur ein Patch (patch) weitergereicht.

Vereinheitlichung (unification) In einem Open-Source-Projekt, das eine entsprechende Komponente betreut, werden diese aCPs bzw. die zugehörigen Modifikationen gesammelt, wenn sie der Autor dort einreicht. Entsprechen sie den Vorstellung des Projekts, werden sie in die Code-Basis des Projekts aufgenommen. Für eine sachgerechte Integration können kleine Änderungen notwendig sein, die durch das Projekt oder in Kooperation mit dem Autor vorgenommen werden. Von Zeit zu Zeit werden aus der Code-Basis mit den eingearbeiteten Modifikationen konsistente, abgerundete, neue Versionen gebaut, von denen man annimmt, die bestmögliche Lösung für die Bedürfnisse der Zielgruppe zu sein. Diese vereinheitlichte Komponente (unified component: uCP) wird dann in Form eines Release (release) zugänglich gemacht.

Vorbereitung (preparation) Distributoren wählen sich aus den vorhandenen Versionen einer uCP die für ihre Zwecke am besten geeignete, passen sie an ihre Distribution und die zugehörigen Richtlinien an, stimmen sie mit den ebenfalls enthaltenen Elementen ab und bereiten sie für eine effiziente Installation auf ein passendes Computersystem vor. Das

Ergebnis ist eine vorbereitete Komponente (prepared component: pCP), die in Form von Paketen (package) transportiert wird.

Installation (installation) Das vom Distributor erstellte Paket wird auf dem Computersystem installiert und an die lokalen Gegebenheiten angepasst. Die daraus resultierende, installierte Komponente (installed component: iCP) ist damit für die Nutzer des Systems verfügbar. Es bedarf daher auch keiner Transportform.

Anpassung (customization) Die auf dem lokalen Computersystem eingerichtete iCP ist zwar bereits verfügbar, aber nicht auf das persönliche Profil des Benutzers angepasst. Zudem ist evtl. noch eine Optimierung auf die vorgesehenen Nutzungsprozesse sinnvoll. Ist auch dieser letzte Vorgang abgeschlossen kann man von einer personalisierten Komponente (customized component: cCP) sprechen, die nun für die Nutzung bereit ist.

Diese typischen Prozesse, die in der Open-Source-Gemeinschaft zu beobachten sind, haben alle eine ähnliche Struktur: Sie nehmen eine Komponente, passen sie einem Kontextmodell an und erstellen dadurch eine andere Komponente. Zudem bilden sie in der aufgezählten Reihenfolge eine Prozesskette, in der die Ausgabe des letzten Prozesses die Eingabe des nächsten bildet. Man kann daher von einem Lebenszyklus sprechen und die jeweiligen Prozesse als Lebensphasen einer Komponente auffassen. Die bei den jeweiligen Prozessen angegebenen Komponentenklassen (aCP, uCP, pCP, iCP, cCP) sind damit Stadien einer Komponente in ihrem Lebenszyklus.

Auch hier kann man die Zuordnung bezüglich der komponentenbasierten Entwicklung zur Komponenten- bzw. Systementwicklung vornehmen:

Komponenten Im Erstellungs- und Vereinheitlichungsprozess geht es primär um die Komponenten, da hier eine Änderung der Funktionalität im Vordergrund steht.

Systeme In den verbleibenden drei Prozessen (Vorbereitung, Installation, Anpassung) geht es um eine zunehmende Anpassung an ein System.

Es gibt noch einen weiteren wichtigen Prozess: die Evaluation (evaluation). Er ist der komplexeste und umfangreichste Prozess in der Open-Source-Welt. Hier werden Änderungswünsche (issues oder requests) erstellt, gesammelt, verteilt und ihr Status verwaltet. Zudem sind durch ihn alle Prozesse miteinander verbunden und sie erhalten Rückmeldungen zu ihren Ergebnissen. Dies gilt insbesondere für die Nutzung und Erstellung, da ohne Evaluation zwischen diesen beiden Prozessen keine Verbindung wäre und über diesen Umweg der Kreislauf geschlossen wird. Änderungswünsche können jedoch von jedem Prozess kommen und alle können diese entgegen nehmen. Ursprung ist der Prozess, in dem ein Problem entdeckt wurde. Ziel ist der Prozess, der am besten geeignet ist, eine Lösung zu liefern. Da es nicht immer klar ist, in welchem Prozess bzw. Kontextmodell ein Problem zu lösen und welche Komponente am besten für eine Änderung geeignet ist, kann die Vermittlung eine schwierige Aufgabe sein. Da es sich hier um eine globale Aufgabe handelt, die über allen anderen Prozessen steht, wäre zumindest für die Verteilung die naheliegende Lösung ein zentraler Prozess, der diese Aufgabe für alle Prozesse und Komponenten erledigt. Bisher wird diese Aufgabe jedoch dezentral von den jeweiligen Prozessen selbst erledigt.

Zum besseren Verständnis nochmal alle betrachteten Prozesse in der Übersicht in Abbildung 4.6. Die Prozesse sind gelb und ihre jeweilige Umgebung mit dem zugehörigen Kontextmodell, auf das sie sich beziehen, ist grün dargestellt. Entlang der Pfeile fließen Informationspakete (blau): Änderungswünschen (issues) und die jeweiligen Komponenten (patch: Pt/aCP, release: Re/uCP,

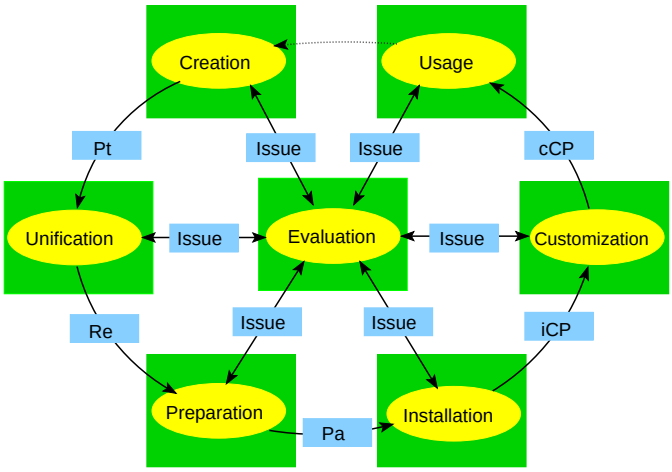

Abbildung 4.6: Open-Source-Lebenszyklus

package: Pa/pCP, iCP, cCP). Der gepunktete Pfeil zwischen *usage* und *creation* veranschaulicht die Schließung des Kreislaufs über den Umweg *evaluation*.

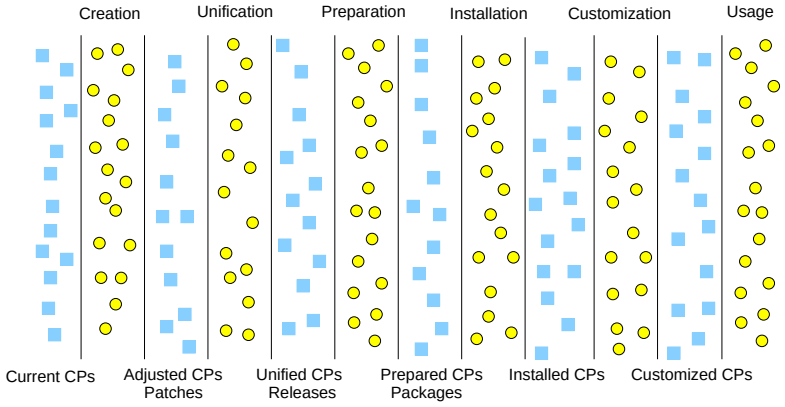

Abbildung 4.7: Kombination von vielen Open-Source-Lebenszyklen ohne Kontext

Auf einem Systemknoten befinden sich viele unterschiedliche Komponenten und jede ist das Ergebnis eines derartigen Lebenszyklus, wenn man von der Ausnahme absieht, dass es sich um eine initiale Version handeln könnte. Abstrahiert man von den genauen Zeitpunkten und betrachtet nur die Tatsache, dass jede Komponente jeden Prozess durchläuft, so ergibt sich eine Wechsel aus Prozessen (gelb) und Ergebnissen (blau), wie in Abbildung 4.7 gezeigt. Dabei wurde von den Kontexten zur Vereinfachung abstrahiert.

4.5.3 Struktur der Prozessdomänen

In dem Open-Source-Lebenszyklus, wie er in Abschnitt 4.5.2 erläutert wurde, stellen Patch, Release und Paket Transportformen einer Komponente dar, die einen Kontextwechsel zwischen den Prozessen markieren. Zudem wird in Abschnitt 6.4.1 gezeigt, dass jeder Prozess ein anderes Kontextmodell verwendet, das jeweils ein klar abgegrenzten Geltungsbereich haben sollte (vgl.

Abs. 6.3.4). In diesem Sinne ändert sich auch die Prozessumgebung zwischen Installation und Anpassung, weil sich u.a. letzteres auf die Bedürfnisse eines bestimmten Benutzers konzentriert. Zudem werden die jeweiligen Prozesse oftmals von unterschiedlichen Akteuren durchgeführt. So wird ein Patch z.B. von einem Gelegenheitsentwickler für die Verbesserung seines Systemknotens entwickelt. Das Release, in dem er integriert wird, stellt evtl. der zugehörigen Projektleiter zusammen, Das Paket wird von dem zuständigen Paketverwalter der jeweiligen Distribution erstellt. Der Administrator installiert dieses dann auf seinem Systemknoten und hat als Ergebnis ein iCP. Dieses kann dann z.B. vom Benutzer selbst auf seine besonderen Bedürfnisse angepasst werden, was zu einer cCP führt, die dann in einem realen Nutzungsprozess eingesetzt wird. Letztlich hat daher jeder der betrachteten Prozesse seinen eigenen Kontext und es findet jeweils zwischen ihnen ein Kontextwechsel statt.

In Abschnitt 4.2.3 wurde das Konzept der Prozessdomänen eingeführt, welche die abgegrenzte Umgebung eines Prozesses darstellt. Nach den vorhergehenden Überlegungen kann man davon ausgehen, dass jeder der benannten Prozesse eine eigene Prozessdomäne hat, wie es in Abbildung 4.8 parallel für drei Komponenten gezeigt wird.

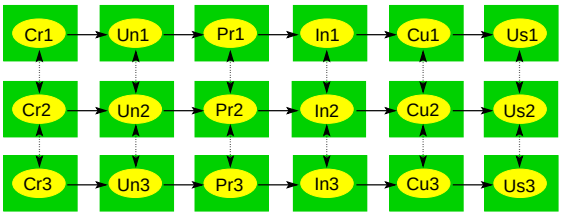

Abbildung 4.8: Gruppierung von Prozessdomänen

Die Prozesse wurden mit gelb markiert (creation=Cr, unification=Un, preparation=Pr, installation=In, customization=Cu, usage=Us). Die zugehörigen Prozessdomänen sind grün. Die durchgezogenen Pfeile stellen die Weitergabe der Komponenten dar. Die gestrichelten Doppelpfeile repräsentieren die Querbeziehungen zwischen den Prozessen und damit auch indirekt zwischen den dabei erzeugten Komponenten.

Eine Prozessdomäne kann zeitlich gesehen unabhängig von dem Ablauf des jeweiligen Prozesses vorhanden sein. Da es i.d.R. immer wieder neue Versionen einer Komponente gibt, werden die zugehörigen Prozesse auch immer wieder durchgeführt. Sie bilden daher so etwas wie einen konstanten Arbeitsbereich, der immer wieder für den selben Prozess verwendet wird. In einer solchen Prozessdomäne können daher auch Meta-Informationen hinterlegt sein, die über mehrere Komponenten-Versionen gleich bleiben. Sie spiegelt mit ihrem Import- und Export-Prozess auch die Notwendigkeit wieder, Eingaben zuerst in den lokalen Kontext zu überführen (Auspacken, Überprüfen, Abhängigkeiten verarbeiten, Inhalt in lokalen Kontext überführen, etc.) und Ausgaben für eine externe Nutzung vorzubereiten (wesentliche Metadaten codieren, Einpacken, etc.).

Es wurde bereits erläutert, warum die unterschiedlichen Prozesse eines Lebenszyklus eine eigene Prozessdomäne benötigen. Es gibt jedoch keinen zwingenden Grund, warum gleichartige Prozesse von unterschiedlichen Komponenten sich ihre Prozessdomäne nicht evtl. teilen sollen. Insbesondere dann, wenn sie aufeinander abgestimmt werden müssen, wie dies z.B. bei der Vorbereitung für eine bestimmte Distribution oder der Installation auf einen bestimmten Systemknoten der Fall ist. Zudem handelt es sich um ähnliche Vorgänge, die Kontextmodelle haben

eine große Übereinstimmung und oftmals werde sie von den selben Akteuren durchgeführt. Man kann sogar soweit gehen, die Prozesse zusammen zu fassen wie es schon in Abbildung 4.9 gemacht wurde.

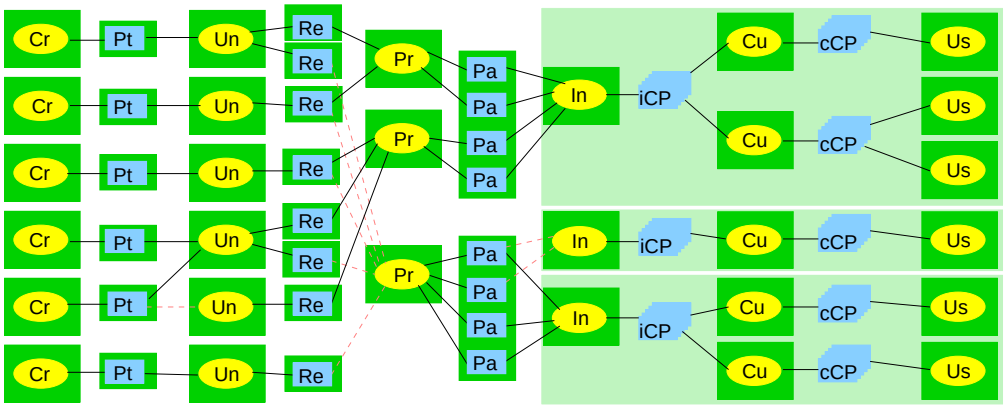

Abbildung 4.9: Kombination von vielen Open-Source-Lebenszyklen mit Kontext

Die Komponenten sind blau dargestellt. Die Prozess-Kontexte und die Komponenten-Kontexte sind alle grün dargestellt. Die blassen grünen Bereiche rechts stellen Systemknoten dar, in denen die jeweiligen Komponenten als Datenobjekte abgelegt sind (iCP, cCP). Die anderen Komponenten-Klassen sind durch ihre Transportform repräsentiert: Pakete (package=Pa), die durch die Vorbereitung in das Kontextmodell (grün) einer Distribution transformiert wurden, Releases (release=Re) und die Patches (patch=Pt). Es handelt sich dabei um drei Systemknoten und zwei Distributionen. Die schwarzen und rosa Linien verbinden die Prozesse jeweils mit ihrer Ein- und Ausgabe.

Doch selbst zwischen den Erstellungs- und Vereinheitlichungsprozessen bestehen oftmals enge Verwandtschaft, da sich z.B. die Abläufe ähnlich sind. Es ist daher nicht verwunderlich, dass sie oftmals die selbe Infrastruktur nutzen (z.B. Hosting-Plattformen wie Sourceforge).

Im Open-Source-Bereich findet man folglich eine Menge Querbeziehungen und Synergieeffekten zwischen den benannten gleichartigen Prozessen, was aufgrund ihrer Lizenz auch möglich ist. Dies ist im Bereich der proprietärer Softwareentwicklung wegen der zentralen Bedeutung des Eigentumsaspekts nicht möglich, der eher für eine Nutzung der Beziehungen entlang des Lebenszyklus geeignet ist. Hier gibt es jedoch wie erläutert eine natürliche Abgrenzung der Prozessdomänen, was Synergieeffekte erschwert.

4.5.4 Parallelisierung der Entwicklungsphasen

Eine typische Charakteristik der Vorgehensweise im Open-Source-Bereich ist eine kleine Schrittgröße (vgl. Abs. 4.3.7). Brown/Booch bezeichnen dies als “open releases” (vgl. Abs. 1.3.2). Raymond sieht darin eine Eigenart des von Linus Torvalds geprägten Entwicklungsstil und beschreibst es mit dem oft zitierten Worten “release early and often” [Raymond 99, S. 30]. Dies alles beschreibt das selbe Vorgehen: man setzt sich nur relativ kleine Ziele für einen Entwicklungszyklus, macht das Ergebnis einer breiteren Benutzergruppe zugänglich, sammelt entsprechendes Feedback ein und nutzt dieses bereits für den nächsten Zyklus. Die kleinen Schrittgröße hat den entscheidenden Vorteil eine frühen Rückkopplung mit Personen, die nicht direkt

an der Entwicklung beteiligt sind. Werden die Zyklen nur ausreichend kurz gehalten, so führt dies zu der in Abschnitt 4.3.8 dargestellten Parallelisierung der Entwicklungsphasen, die bei einer risikoreichen Entwicklung gewisse Vorteile bringt:

1. Um so schwieriger die Umgebung einer Software zu erfassen ist und um so weniger bisher über sie bekannt ist, desto größer wird die Wahrscheinlichkeit von Irrtümern und Fehlentwicklungen innerhalb ihres Erstellungsprozesses. Es ist wichtig solche Irrtümer frühzeitig zu erkennen, da sie ansonsten zu einer Basis für weiterführende Arbeiten werden und eine spätere Korrektur dann mit viel Aufwand verbunden sein kann.
2. Die aufwändige Kontexterfassung kann auf das Notwendige reduziert werden, da man sie parallel zur Entwicklung durchführt, und auf diese Weise keine Kontextbereiche erforschen muss, die letztendlich dann doch nicht bei der Entwicklung benötigt werden.
3. Durch die vielen Veröffentlichungen und die damit verbunden Rückmeldungen entsteht eine größere Nähe zwischen der Entwicklung und dem Nutzungskontext, was den Entwicklern das Verständnis des relevanten Kontexts erleichtert.

5 Agenten-Perspektive

Die Agenten-Perspektive beschäftigt sich mit den beteiligten Akteuren, ihren Aktivitäten und der Interaktion zwischen ihnen. Dabei spielt die Organisation der Akteure eine zentrale Rolle.

Die klassische Organisationsform ist die projektbasierte Entwicklung. Es hat sich jedoch im Laufe der Arbeit gezeigt, dass die in der Literatur primär dokumentierten Strukturen, die auf die Bedürfnisse des Projektmanagement ausgerichtet sind, nicht zu den wesentlichen Organisationsstrukturen im Open-Source-Bereich passen (z.B. Open-Source-Projekte). Trotz intensiver Recherche wurde auch keine andere passende Organisationsform in der Literatur gefunden. Daher wurde in Anlehnung an die Werke von Axelrod ein einfaches, allgemeines Modell der kooperativen Entwicklung erstellt (s. [Axelrod 06, Axelrod 97, Axelrod 99]). In diesem Modell wurde im Vergleich zu Projekten von großen Teilen des Managements abstrahiert, indem z.B. die Leitungsfunktion auf die Selektion von Ergebnissen reduziert wurde.

5.1 Projektbasierte Entwicklung

Die projektbasierte Entwicklung ist heute in der Informationstechnologie das übliche Vorgehen, um Software zu produzieren. Es ist in einer Vielzahl von Standards und Büchern beschrieben (z.B. [PMBOK 00, Patzak 04, IEEE1058 98, IEEE1074 06]) und wurde über die Jahrzehnte in Theorie und Praxis erprobt, untersucht, optimiert und standardisiert. Im Folgenden werde ich die wesentlichen Aspekte dieser Methode vorstellen um sie später gegen die kooperative Entwicklung abgrenzen und die Unterschied aufzeigen zu können.

5.1.1 Projekte

Das Project Management Institute unterscheidet zwei Methoden der Bearbeitung von Aufgaben innerhalb von Organisationen: Projekte (projects) und Tätigkeiten (operations). Während Tätigkeiten als fortlaufend und sich wiederholend angesehen werden, sind Projekte als ein “zeitlich begrenztes Vorhaben zur Erstellung eines einmaligen Produktes oder Dienstes” [PMBOK 00, S. 4] definiert. Zudem können Projekte auch in sogenannten “Programmen” gruppiert werden, die das Erreichen eines umfangreicheren Ziels in mehreren Etappen erlaubt (vgl. [PMBOK 00, S. 10]).

Man kann diese drei Begriffe hinsichtlich des Bestimmtheitsgrades in eine Hierarchie rationalen menschlichen Handelns einordnen [Patzak 04, S. 18]:

Routineaufgaben Eine häufig wiederholte Verkettung von Aktivitäten bei denen Ausgangslage, angestrebtes Ergebnis und erforderliche Maßnahmen bereits bekannt und festgelegt sind und man von einem erfolgreichen Bearbeitung ausgehen kann (z.B. Beschaffung eines Zulieferteils).

Projekte Eine einmalige Vernetzung von Aktivitäten bei denen Ausgangslage und angestrebtes Ergebnis festgelegt, die erforderlichen Maßnahmen jedoch noch (teilweise) unklar sind und damit bei der Zielerreichung wesentlich Unsicherheiten bestehen (z.B. Produktentwicklung).

Programme Eine Vernetzung von Projekten, bei der das angestrebte Ergebnis nur grob durch eine Zielvorstellung vorgegeben ist, der Weg dorthin und und die darin enthalten Projekte jedoch teilweise noch völlig offen sind (z.B. Aufbau eines neuen Marktes).

Patazak/Rattay benennen folgende wesentlichen Merkmale für Projekte: zeitlich begrenzt, neuartig, zielorientiert, komplex/dynamisch, interdisziplinär und bedeutend [Patzak 04, S. 19].

Projekte haben ein klares Ziel: das angestrebte Ergebnis. Wurde es erreicht sind die Arbeiten vollendet und das Projekt wird abgeschlossen. Dies steht auch in einem engen Zusammenhang mit der Forderung nach der zeitlichen Begrenzung, was bedeutet sie haben einen klaren Anfang und ein klares Ende. Auch wenn derartige Vorhaben Jahre dauern, vorzeitig oder verspätet enden können, haben sie einen festgelegten inhaltlichen und zeitlichen Abschluss. Fortlaufende Arbeiten werden eben nicht als Projekte angesehen.

Eine weitere Forderung dieser Auffassung ist die Neuartigkeit der Arbeitsleistung. Es handelt sich dabei nicht um Routine, sondern um eine Tätigkeit, die zumindest zu einem gewissen Teil in dieser Form vorher noch nicht durchgeführt wurde und es ist damit ein wesentlicher Teil der Arbeit einen geeigneten Weg zum Erreichen der gegeben Ziele zu finden. Aufgrund dieser Unsicherheit sind Projekte stets mit einem gewissen Risiko behaftet.

Meredith/Mantel sehen in Ergebnis, Kosten und Zeit die drei wesentlichen Aspekte eines Projekts, die für einen Erfolg ausschlaggebend sind[Meredith 00, S. 3]. Sie lassen sich wie folgt beschreiben:

Ergebnis Das Projekt dient der Erreichung von einer Menge von festgelegten Ergebnissen. Sind sie erreicht, ist das Projekt beendet. Es ist evtl. notwendig diese Vorgaben im Laufe des Projekts an eine neue Situation anzupassen.

Kosten Es werden eine bestimmte Menge von Ressourcen zur Verfügung gestellt, die dazu verwendet werden sollen das Ergebnis zu erreichen. Diese Mittel können im Laufe des Projekts angepasst werden, sind aber bekannt und nicht beliebig zu variieren. Zu diesen Mitteln gehören u.a. Arbeitsmittel, Infrastruktur und Arbeitskräfte.

Zeit Es existiert ein festgelegter Zeitrahmen, der angepasst werden kann, aber in jedem Fall beschränkt ist. Das Projekt hat einen klaren Anfang und ein klares Ende.

Das angestrebte Ideal ist ein Projekt, das alle drei Aspekte früh festlegt, möglichst vor Beginn, und bis zum Abschluss nicht mehr ändern muss.

5.1.2 Projekt-Management

Das Projekt Management[1] steht in einem engen Zusammenhang mit dem Projektbegriff. Es beschäftigt sich primär mit der Aufgabe, die genannten drei Faktoren in ihrem Zielbereich zu halten und damit das angestrebte Ergebnis in dem vorgegebenen Zeitrahmen mit den bereitgestellt Mitteln zu erreichen.

[1] Definition Projekt Management: “the application of knowledge, skills, tools, and techniques to project activities to meet project requirements” [PMBOK 00, p. 6]

Patazak/Rattay benennen Planung, Organisation/Kommunikation, Teamführung und Controlling als die wesentlichen Aufgaben des Projekt-Management [Patzak 04, S. 22], die sie in ein ganze Reihe von Teilaufgaben zerlegen, von den hier abstrahiert wird. Inhaltlich haben sie folgende Funktion:

Planung Sie umfasst die detaillierte Festlegung und Abstimmung aller wesentlichen Arbeitsschritte und die Analyse der zugehörigen Einflussfaktoren.

Organisation Die Gestaltung und Optimierung des Umfeldes in dem die Arbeiten durchgeführt werden sollen. Dies umfasst ebenfalls die Struktur des sozialen Systems selbst (Rollen, Verantwortlichkeiten, etc.).

Teamführung Sie ist der Versuch durch direktes Einwirken auf die Projektmitglieder einen positiven Einfluss auf den Projektverlauf zu nehmen.

Controlling Es handelt sich dabei um die Kontrolle und Steuerung von wesentlichen Faktoren bzw. Risiken (z.B. Qualität, Kosten, Ressourcen, Termine) und die Anordnung von korrektiven Maßnahmen bei ungünstigen Entwicklungen.

Der PMBOK Leitfaden unterteilt das Projekt-Management in neun Wissensbereiche (“project management knowledge areas”), die das Wissen und seine Anwendung in Bezug auf bestimmte Teilprozesse des Projekt-Management beschreiben. Diese erfüllen jeweils eine bestimmte Funktion innerhalb des Projekts [PMBOK 00, S. 7]:

Integration Sicherstellung der ordnungsgemäßen Koordination der unterschiedlichen Elemente des Projekts

Fokus Sicherstellung, dass das Projekt genau alle die Arbeiten umfasst, welche notwendig sind, um es erfolgreich abzuschließen. Allerdings soll er auch dafür sorgen, dass es keine unnötigen Arbeiten enthält.

Zeit Sicherstellung der rechtzeitig Fertigstellung des Projekts

Kosten Sicherstellung des Projektabschluss innerhalb des genehmigten Budgets

Qualität Sicherstellung der Abdeckung von den Bedürfnissen, für die das Projekt ins Leben gerufen wurde

Personal Sicherstellung des effektivsten Einsatzes der Projektmitglieder

Kommunikation Sicherstellung der zeitgerechten und angemessenen Erstellung, Sammlung, Verbreitung, Speicherung und Endlagerung von Projektinformationen

Risiko Identifikation, Analyse und Behandlung von Projektrisiken

Beschaffung Besorgung von Gütern und Dienstleistungen von außerhalb der durchführenden Organisation

5.1.3 Interessengruppen in einem Projekt

Der PMBOK Leitfaden definiert die folgenden allgemeinen Haupt-Interessengruppen [PMBOK 00, S. 16]:

Projektmanager Die Person, die für das Management des Projekts verantwortlich ist.

Kunde Die Person oder Organisation, die das Ergebnis des Projekts verwenden wird. Es kann mehrere Ebenen von Kunden geben. So gibt es z.B. bei einem neuen pharmazeutischen Produkt den Arzt, der es verschreibt, der Patient, der es einnimmt, und die Versicherung, die es bezahlt.

Durchführende Organisation Das Unternehmen dessen Angestellte am direktesten in die Erledigung der Projektarbeit eingebunden ist.

Projektmitglieder Die Personengruppe, welche die Projektarbeit ausführt.

Sponsor Die Einzelperson oder Gruppe innerhalb oder außerhalb der durchführenden Organisation, welche die finanziellen Mittel für das Projekt zur Verfügung stellt.

5.1.4 Lebenszyklus eines Projekts

Ein Projekt wird i.d.R. in mehrere Abschnitte eingeteilt, die Projektphasen genannt werden. Jede Projektphase ist gekennzeichnet durch die Fertigstellung einer oder mehrerer Leistungen, die konkrete, nachprüfbare Arbeitsergebnisse sind (z.B. ein funktionierender Prototyp). Am Ende einer Phase findet üblicherweise eine Überprüfung der bisherigen Projektergebnisse statt, um festzustellen, ob das Projekt in die nächste Phase eintreten kann, und um bisherige Fehler zu erkennen und zu beseitigen. Jede Projektphase enthält normalerweise eine bestimmte Menge an Leistungen, die hauptsächlich zur Unterstützung der Management-Kontrolle dienen. Die Mehrzahl davon bezieht sich direkt auf Ergebnisse der primären Phase, in der i.d.R. so etwas wie ein Planung stattfindet. Die jeweiligen Phasen werden typischerweise nach diesen Hauptleistungen benannt (z.B. Anforderungen, Design, etc.) [PMBOK 00, S. 11-12].

Ein Lebenszyklus setzt sich nun aus mehreren solcher aufeinander abgestimmter Phasen zusammen und definiert den Anfang und das Ende eines Projekts.

5.2 Kooperative Entwicklung

Die folgende Vorgehensweise basiert auf dem Zusammenschluss von einer Gruppe von Akteuren, die ähnliche Interessen verfolgen und sich durch das kooperative Vorgehen entsprechende Vorteile insbesondere Synergieeffekte versprechen.

5.2.1 Ansatz der kooperativen Entwicklung

Die in Abschnitt 5.1 umrissene Arbeitsweise dient der Herstellung eines Ergebnis, das durch einen Wunsch oder ein Problem von Außen vorgegeben wird. Die Beteiligten sind dabei primär Erfüllungsgehilfen. Selbst wenn das angestrebte Ergebnis ausschließlich von Projektmitgliedern bestimmt wurde, müssen sie sich bei diesem Ansatz vor Projektbeginn auf ein verbindliches Ziel einigen, das dann nicht mehr ohne weiteres im Laufe der Durchführung geändert werden kann. Daher ist das angestrebte Ergebnis auch in diesem Fall während des Projekts von “Außen” vorgegeben und steht eben nicht mehr zur Disposition. Bei diesem Ansatz gilt die größte Aufmerksamkeit dem Projekt-Management, das für das Erreichen des vorgegebene Ergebnis innerhalb der Zeit mit den bereitgestellten Mitteln sorgen soll.

Das gemeinsame Arbeiten an einem Problem ist jedoch auch in anderer Form denkbar. Haben viele Menschen ein ähnliches Problem bzw. einen ähnlichen Wunsch, so können sie bei einer entsprechenden Zusammenarbeit Synergieeffekte nutzen ohne ihre Autonomie aufzugeben und so trotzdem jeder Einzelne für sich selbst genommen das Aufwand/Nutzen Verhältnis verbessern. Bei einer solchen kooperativen Entwicklung sind angestrebtes Ergebnis, Zeit und Mittel keine statischen Größen, sondern können dynamisch an die jeweilige Situation angepasst werden. Die Festlegung der Aufgaben, eingesetzte Mittel und der zeitlich Rahmen werden damit für das Gesamtvorhaben und jeden Einzelnen der Gegenstand eines fortlaufenden Einigungs- und Optimierungsprozesses. Dehnt man z.B. den Problembereich aus, so steigt zwar einerseits der notwendige Aufwand, aber andererseits auch die Zahl der potentiellen Helfer. Zudem gibt es viele weitere Einflussfaktoren, die bei einem derartigen Arbeiten berücksichtigt werden müssen: So steigt mit der Zahl der Beteiligten der Koordinationsaufwand, oftmals ist es schwierig die Schnittmengen der unterschiedlichen Interessen zu erkennen und der Nutzen eines Teilnahmewilligen für das Vorhaben ist nur schwierig im Voraus abzuschätzen.

Entscheiden sich nun die Beteiligten ihre Autonomie während der Entwicklung zumindest größtenteils zu behalten und den Prozess offen zu gestalten, d.h. es besteht während der ganzen Zeit die Möglichkeit neue Akteure aufzunehmen und alte zu verabschieden, entsteht daraus eine hohe Dynamik. Diese Dynamik erfordert einerseits eine entsprechende Organisationsform und anderseits ein passendes Verhalten der Teilnehmer. Dies kann u.a. eine hohe Flexibilität oder Eigenständigkeit sein. Was konkret das jeweils passende Verhalten ist, kann so nicht pauschal gesagt werden. Es kommt dabei auf die anderen Teilnehmer und die genauen Umstände an. Axelrod hat sich in seinen Büchern mit Kooperationsstrategien beschäftigt und bieten einen ersten Ansatzpunkt für weitere Studien [Axelrod 06, Axelrod 97, Axelrod 99].

Trotz der hohen Dynamik kann man auch bei dieser Entwicklungsform einige allgemeine Strukturen und Vorgänge erkennen, die stets vorhanden sind bzw. stattfinden. Diese sollen im nächsten Abschnitt behandelt werden.

5.2.2 Grundstrukturen der kooperativen Entwicklung

Das Ziel der hier betrachteten Vorhaben ist die Erstellung oder Überarbeitung eines Entwicklungsgegenstands, der sich ausschließlich aus Informationen zusammensetzt. Dabei kann es sich z.B. um Dokumente, Quelltexte oder auch eine Menge von derartigen Elementen handeln. Dieser Gegenstand wird durch die Auffassung der zugehörigen Teilnehmer eines Entwicklungsvorhabens definiert. Sie befindet sich wie der Gegenstand selbst in einem ständigen Wandel. Die konkrete Auffassung zu einem bestimmten Zeitpunkt manifestiert sich in den verschiedenen Datenobjekten, die als Teil des Vorhabens angesehen werden. Die Entwicklung findet durch Akteure statt, die diese Objekte manipulieren. Die Handlungen lassen sich entsprechend ihrer Funktion im Gesamtzusammenhang in mehrere Aktivitäten aufteilen. Damit lassen sich wiederum bestimmten Rollen zuordnen, welche die Funktionen eines Akteurs in einem solchen Vorhaben kategorisieren.

Im Folgenden werden daher die Grundstrukturen der kooperativen Entwicklung von Datenobjekten durch die wesentlichen Objekte, Aktivitäten und Rollen beschrieben.

5.2.2.1 Objekte

Es handelt sich dabei um passive Elemente, die aus Informationen bestehen.

Version Eine Version ist eine vollständige, konkrete Ausprägung eines Entwicklungsgegenstands, die aus der Initialversion und akzeptierten Änderungen besteht. Sie entsteht aus ihrer Vorgänger-Version und zugefügten Änderungen.

Aufgabe Es handelt sich dabei um die Beschreibung einer gewünschten Änderung zu einer bestehenden Version.

Änderung Eine Änderung ist ein modifiziertes Fragment des Entwicklungsgegenstands, das auf einer vorhandenen Version basiert.

Release Ein Release ist eine Version des Entwicklungsgegenstands, die für die Nutzung außerhalb des Entwicklungskontextes vorbereitet wurde.

Von allen diesen Objekten gibt es auch Objekt-Sammlungen.

5.2.2.2 Rollen

Rollen sind Platzhalter für das Verhalten eines aktiven Elements (vgl. [ISO10746-2 96, role]). Sie repräsentieren damit Akteure, die einem bestimmten Verhaltensmuster folgen (Aktivität). Hier ist jeder Rolle genau eine Aktivität zugeordnet.

Leiter lenkt das Entwicklungsvorhaben. Er definiert es, indem er entscheidet, was dazu gehört und was nicht.

Entwickler gestaltet die konkreten Objekte, die den Entwicklungsgegenstand manifestieren.

Aufgabensteller regt durch das Formulieren von Aufgaben neue Veränderungen an.

5.2.2.3 Aktivitäten

Aktivitäten werden hier verstanden als eine Menge von möglichen Aktionen eines Akteurs, die gemeinsam eine bestimmte Funktion innerhalb des Vorhabens erfüllen.

Anregen Durch das Erstellen von Anregungen werden mögliche Verbesserungen und Richtungen der Weiterentwicklung aufgezeigt.

1. Aufgabe zu Version erstellen

Gestalten Aufgezeigte Anregungen werden in konkrete Änderungen umgesetzt.

1. Änderung zu einer vorhanden Version erstellen
2. Änderung(en) in bestehende Version integrieren, woraus eine neue Version entsteht
3. autorisierte Version als Release veröffentlichen

Die Integration einer oder mehrerer Änderungen erzeugt eine neue Version. Bestehende Aufgaben oder Änderungen zur vorhergehenden Versionen werden automatisch zur nächsten Version übertragen, wenn diese Verbindung nicht explizit gelöscht wird.

Lenken Durch die Selektion der Anregungen, Änderungen und Teilnehmer wird dem Gesamtvorhaben eine bestimmte Richtung gegeben.

1. Aufgabe/Änderung aufnehmen/entfernen
2. Version/Release autorisieren
3. Teilnehmer (Leiter/Entwickler/Aufgabensteller) aufnehmen/entfernen
4. Entwicklungsvorhaben starten/beenden

5.2.3 Arbeitsweise

Die in Abschnitt 5.2.2 dargestellten Strukturen passen auf viele Formen der Zusammenarbeit. Man kann auch ein Projekt auf diese Art und Weise sehen. Sie stellen daher keine Möglichkeit zur Abgrenzung dar, sondern ein Grundgerüst und strukturelles Minimum für weitere Betrachtungen. Entscheidend für die Abgrenzung zur projektbasierten Entwicklungsform ist die *Abwesenheit* von vorgegebenen Ordnungsstrukturen, die der Durchsetzung und dem Erhalt von Kontrolle und Steuerung durch ein zentrales Projektmanagement dienen. Fehlen diese Strukturen und Abläufe müssen jedoch für eine produktive Zusammenarbeit andere Mechanismen vorhanden sein, die zumindest ein Teil dieser Aufgaben übernehmen, damit es für die Beteiligten möglich ist wie eine Einheit zu agieren. Zu diesem Zweck muss es einen Mechanismus geben, der zumindest dafür sorgt, dass sich aus den Aktivitäten der einzelnen Beteiligten ein sinnvolles Gesamtwerk entsteht. Diesen Vorgang kann man in drei Bereich aufteilen:

Richtungsgebung Die Arbeiten müssen zumindest im Gesamtergebnis eine gemeinsame Richtung ergeben.

Koordination Die Arbeiten der einzelnen Beteiligten sollte sich möglichst wenig stören und sich stattdessen eher ergänzen.

Integration Am Ende müssen die Ergebnisse aller Beteiligten in irgendeiner Weise zu einem konsistenten Ganzen zusammen gesetzt werden.

Im Folgenden wird nun die Frage untersucht wie weit die vorgestellten Grundstrukturen ausreichen, um diese drei Funktionen für die darin enthalten Aktivitäten bereitzustellen. Dabei soll insbesondere der dafür genutzte Mechanismus herausgestellt werden.

5.2.3.1 Richtungsgebung

Die Festlegung der Richtung der Entwicklung findet in den dargestellten Strukturen durch eine Kombination aus allen drei Aktivitäten statt: Anregen, Lenken und Gestalten. Dabei handelt es sich um ein mehrschichtiges Muster von Produktion und Selektion. Neue Ideen werden von dem Aufgabensteller produziert und durch den Leiter selektiert. Aus den akzeptierten Aufgaben wählen die Entwickler sich wiederum Elemente aus, an denen sie arbeiten wollen. Aus den daraus entstehenden konkreten Änderungen selektiert der Leiter wiederum jene, die er für das Gesamtvorhaben für nützlich hält. Die Entwickler wählen wiederum aus den akzeptierten Änderungen, passende aus und erstellen daraus eine neue Version des eigentlichen Entwicklungsgegenstands, die vom Leiter beim Erfüllen seiner Ansprüche dann als Release autorisiert werden.

Es findet dabei durch alle drei Aktivitäten bzw. Rollen eine Selektion statt. Der Aufgabensteller wählt aus der fast unendlichen Zahl von möglichen Veränderungen eine konkrete Variante. Der Leiter überträgt aus den angebotenen Aufgaben und Änderungen eine begrenzt Menge in das Vorhaben. Der Entwickler wiederum wählt konkrete, akzeptierte Aufgaben und Änderungen aus dieser Menge aus, um sie zu bearbeiten.

Die Leiterrolle hat hier eine ausschließlich selektive Funktion. Nur der Aufgabensteller und der Entwickler üben eine produktive, schöpferische Tätigkeit aus, in dem sie Änderungen bzw. Aufgaben erzeugen.

Die Menge aller akzeptierten Anregungen können dann als die Richtung der Entwicklung angesehen werden. Es liegt dabei in der Verantwortung des oder der Leiter, dafür zu sorgen, dass die einzelnen Aufgaben miteinander harmonieren, eine konsistente Einheit bilden und nicht unnötigerweise den Interessen der Beteiligten entgegenstehen. Die Grundlage der Selektionsentscheidungen ist wiederum eine Strategiefrage, die auf ganz unterschiedliche Weise gelöst werden kann und in der vorgestellten Grundstruktur nicht mehr erfasst ist.

Zur Verdeutlichung soll hier jedoch exemplarisch kurz die Strategiefrage des Leiters diskutiert werden. So kann er z.B. versuchen die Anzahl der akzeptierten Aufgaben zu minimieren, um eine Konzentration der Kräfte zu erreichen, riskiert dabei jedoch den Verlust von Teilnehmern, die ihre Interessen nicht mehr abgedeckt sehen. Eine anderer Ansatz ist die Maximierung der Aufgaben, d.h. alles wird akzeptiert, was nicht im Widerspruch zu bereits übernommenen Anregungen steht. In diesem Fall kann man zwar wahrscheinlich auf eine Menge von potentiellen Teilnehmern zurückgreifen, aber die Kräfte verteilen sich auch auf viele Aufgaben. Dieses Situation könnte z.B. durch Etappenziele (Meilensteine) entschärft werden, die eine Priorisierung von Aufgaben darstellt.

Die Grundstruktur ist in der Lage einer Entwicklungsarbeit durch einen Produktions- und Selektionsmechanismus eine Richtung zu geben. Ob diese Richtungsgebung jedoch effizient, erfolgreich oder gar optimal für das Erreichen eines bestimmten Ziels ist, hängt maßgeblich von den verfolgten Handlungsstrategien und dem Zusammenspiel der verschiedenen Akteuren in ihren jeweiligen Rollen ab, was beides nicht durch die Grundstruktur erfasst ist.

5.2.3.2 Koordination

Malone/Crowston definieren Koordination als “Managen von Abhängigkeiten zwischen Aktivitäten” [Malone 94]. Sie nennen explizit die folgenden Arten von typischen Abhängigkeiten zwischen Aktivitäten, die im Folgenden für die vorgestellte Grundstruktur diskutiert werden:

begrenzte Ressourcen Die einzige gemeinsam genutzte, begrenzte Ressource, die in diesem Zusammenhang von wesentlicher Bedeutung ist, sind die Akteure und ihre Zeit. Insbesondere in den produktiven Rollen Entwickler und Aufgabensteller. Hier sind jedoch die Akteure selbst für die Verteilung ihrer Zeit zuständig bzw. eine zentrale Zuteilung ihrer Zeit wird durch die Strukturen nicht erfasst.

Produzenten/Konsumenten Verhältnis Es gibt eine derartige implizite Abhängigkeit zwischen den Objekten in folgender Reihenfolge: Aufgabe, Änderung, Version, Release. Dies bedeutet, zunächst muss es eine Aufgabe geben, aus der eine Änderung erstellt wird, die zu einer neuen Version führt, die dann wiederum als Release freigegeben werden kann.

Aus diesem Verhältnis folgt, dass der jeweilige produktive Prozess zur Erzeugung des Vorgänger-Objekts abgeschlossen sein muss, bevor der Nachfolge-Prozess beginnen kann. Dies steht in der dargestellten Struktur jedoch außer Frage, da erst die erfolgreiche Erstellung des Vorgänger-Objekts den Folgeprozess auslösen kann. Auch hier gilt, dass eine zentrale planende Instanz nicht erfasst wird, die evtl. schon im Voraus von der baldigen Verfügbarkeit eines solchen Objekts weiß.

Beschränkungen bezüglich der Gleichzeitigkeit Bis auf den Integrationsprozess von Änderungen in die aktuelle Version können alle genannten Aktionen, die im Rahmen der dargestellten Aktivitäten stattfinden, nebenläufig ausgeführt werden.

Aufgabe/Teilaufgabe Beziehungen Die Aufteilung einer größeren Aufgabe in mehrere Teilaufgaben bringt immer gewisse Vor- und Nachteile mit sich (vgl. Abschnitt 4.3). Einer dieser Nachteile ist das Entstehen von möglichen Abhängigkeit zwischen den Teilaufgaben. Dies gilt insbesondere dann, wenn die Teilaufgaben für sich allein genommen keinen Sinn machen. Die vorgestellte Struktur liefert zunächst keinen Mechanismus, der das Managen von derartigen Abhängigkeiten unterstützt. Es liegt also in der Verantwortung der Beteiligten, eine Strategie zu finden um dieses Problem zu lösen. Die einfachste Lösung ist, ausschließlich eigenständige Aufgaben zu formulieren, die nur auf dem gegenwärtigen Stand des Entwicklungsgegenstands aufbauen.

Sonstige Abhängigkeiten In diese Kategorie fallen hier primär inhaltliche Überschneidungen von Aufgaben, ihrer Bearbeitung und ihrer Ergebnisse. Findet hier keine Abstimmung statt, kann dies zu Konflikten und Problemen führen, die eine Ablehnung der erstellten Resultate bei der anschließenden Selektion bzw. eine Überarbeitung erforderlich machen. Die Beteiligten brauchen auch hier eine Strategie, um mit dieser Problematik umzugehen. Es wäre eine Möglichkeit, dafür zu sorgen, dass solche Aktionen nicht parallel ausgeführt werden (z.B. indem derartige Aufgaben gar nicht akzeptiert werden), aber dies ist nur *eine* denkbare Variante. Man kann genauso gut die Entstehung von Konflikten in Kauf nehmen und sie bei Auftreten lösen, es den Akteuren im jeweiligen Einzelfall überlassen sich miteinander abzustimmen oder eine zusätzliche externe Instanz etablieren, die diese Funktion übernimmt.

Insgesamt kann man sagen, dass die Grundstruktur eine Koordination in Bezug auf die benannten Aktivitäten selbst nicht erfasst, sie allerdings auch nicht ausschließt. Es wurde sozusagen von ihr weitgehend abstrahiert. Die immer wieder auftretende Selektion selbst stellt jedoch eine Art von Koordination in Bezug auf den gesamten Entwicklungsvorgang dar. Sie sorgt im schlechtesten Fall, bei der Abwesenheit von anderen Koordinationsmechanismen, für das Funktionieren des Gesamtsystems. Durch intelligente Koordinationsstrategien kann jedoch die Produktivität und Effektivität des Gesamtsystems deutlich gesteigert werden, indem das Aussortieren von Arbeitsleistungen und die damit verbundene Fehlinvestition von Produktivität reduziert werden können. Dabei ist jedoch zu beachten, dass jegliche Form von Koordination wiederum Arbeitsleistung verbraucht, die zunächst wieder erwirtschaftet werden muss.

5.2.3.3 Integration

Alle Aufgaben, daraus resultierende Änderungen und die anschließenden Einpassungsarbeiten bei der Erstellung einer neuen Version müssen zueinander passen und ein konsistentes Ganzes

ergeben. Dies wird ebenfalls primär durch die mehrfache Selektion und die damit verbundene Ablehnung von unpassenden Elementen sichergestellt. Durch das mehrstufige Vorgehen können unpassende Arbeiten bereits vorzeitig also solche erkannt und damit evtl. Entwicklungspotential gespart werden. Aber auch hier kann man ähnlich wie bei der Koordination durch unterschiedliche Maßnahmen für höhere Erfolgschancen bei der Annahme von Elementen sorgen:

Explizite Richtlinien Es können verbindliche Regeln explizit benannt oder referenziert werden. So kann man auf Standards und andere Dokumente verweisen bzw. konkrete Konventionen formulieren, die eingehalten werden sollen.

Feedback Sowohl bei einer Ablehnung wie auch bei aufkommenden Fragen während der Erstellung von Anregungen, Änderungen oder Einpassungen sind für weitere Arbeiten Hinweise hilfreich, welche Aspekte problematisch bzw. unannehmbar sind.

Kleine Schrittgröße Indem man nur kleine Änderungen in einer relativ kurzen Zeit erstellt, kann das Risiko einer konfliktbehafteten Veränderung des Entwicklungsgegenstands minimiert werden (vgl. auch Abschnitt 4.3.7).

Diese Maßnahmen sind jedoch bereits nicht mehr in der Grundstruktur enthalten, sondern wären Ergänzungen, mit denen man bei geeignetem Einsatz eine Steigerung der Produktivität und Effizienz des Gesamtvorhabens erreichen kann.

Es sind auch andere Maßnahmen und Strategien möglich, um die Effizienz zu verbessern. Die projektbasierte Vorgehensweise ist eine davon. Eine weitere Möglichkeit wäre die fortlaufende Besprechung und Anpassung der Aktivitäten durch alle Beteiligten.

5.2.3.4 Zusammenfassung

Die vorgestellte Grundstruktur liefert durch die Selektion rudimentäre Mechanismen für Richtungsgebung, Koordination und Integration.

In den meisten Fällen sind diese jedoch aufgrund ihres hohen Verlusts an Arbeitsleistungen durch Ablehnungen in Bezug auf Produktivität und Effizienz allein unbefriedigend.

Zusätzliche, passende Mechanismen und intelligente Handlungsstrategien, die durch die Grundstrukturen nicht erfasst wurden, können hier deutliche Verbesserungen erzielen ohne die Selektionsvorgänge zu stören. Wie diese Maßnahmen genau aussehen hängt von den jeweiligen Umständen und Beteiligten des konkreten Entwicklungsvorhabens ab. Zudem muss man davon ausgehen, dass diese Strategien und Mechanismen nicht statisch sind, sondern einem fortlaufenden Anpassungsprozess bezüglich der Umgebung und den Beteiligten unterworfen sind.

5.3 Vergleich der beiden Vorgehensweisen

5.3.1 Grundprinzip

Der zentrale Unterschied zwischen diesen beiden Vorgehensweisen liegt in der Organisationsform.

Projekt Der beschriebene Ansatz ist in jeder Hinsicht auf die zentrale Planung, Kontrolle und Steuerung durch das Projekt-Management ausgerichtet. Die beteiligten Akteure sind "nur" Erfüllungsgehilfen, die durch das Management vorgegebenen und aufeinander abgestimmten Aufgaben erledigen. Es wird von den Beteiligten erwartet, dass sie ihre eigenen Wünsche und Ziele dem von Außen vorgegebenen Projektziel unterordnen.

Kooperation Es handelt sich hierbei um den Zusammenschluss einer Gruppe von Akteuren, die ähnliche Interessen verfolgen und sich durch das kooperative Vorgehen entsprechende Vorteile versprechen. Dabei wird davon ausgegangen, dass sie ihre Autonomie während des Vorhabens nicht an eine zentrale Kontrollinstanz abgeben und gemeinschaftliche Ziele nur verfolgen, wenn es ihren eigenen Interessen dient.

Diese unterschiedliche Organisation hat entsprechende Konsequenzen für die gesamte Zusammenarbeit.

5.3.2 Aufgabendefinition

Die Bestimmung des verfolgten Ziels findet bei diesen beiden Ansätzen auf unterschiedliche Weise statt.

Vorabdefinition (Projekt) Man versucht vorab auf einer abstrakten Ebene unter Beteiligung aller relevanten Interessengruppen ein klares Ziel zu definieren. Im Rahmen dieser Anforderungsanalyse und Verhandlungen entstehen dann detaillierte Vorgaben für Ergebnis, Zeit und Kosten. Das Ergebnis ist durch eine Spezifikation, Zeit und Kosten durch einen Vertrag oder sogar einen exakte Planung festgelegt, die ebenfalls klärt, wer welche Kosten trägt bzw. Pflichten zu erfüllen hat. Diese Phase ist der eigentlichen Entwicklungsarbeit vorgelagert, die dann im Idealfall ohne weitere Beeinflussung von externen Interessengruppen stattfindet. Die ausführende Organisation und ihre Akteure sind dann an diese Vorgaben gebunden und müssen in diesem Rahmen das Vorhaben durchführen.

Resultierende (Kooperation) Das angestrebte Arbeitsziel wird nicht vorab festgelegt, sondern ergibt sich im Verlauf. Vor allem anderen können auf diese Weise in die Entscheidungsfindung bereits erarbeitete Teil- bzw. Zwischenergebnisse mit einbezogen werden. Zudem kann man schrittweise entwickeln, mehrere unterschiedliche Ansätze ausprobieren und sich dann erst nach einer Evaluierung der Varianten entscheiden. Damit haben alle Beteiligten mehr Möglichkeiten und können zumindest provisorische Kompromisse leichter eingehen, da es im weiteren Verlauf wieder verändert werden kann. Es handelt sich dabei um eine System der Selbstorganisation wie es in Kapitel 7 geschildert wird. Dabei wird die Entwicklung und Zielfindung miteinander vermischt, weder Ergebnis noch dahin führende Arbeitsschritte stehen vorher fest und nur die Arbeitsweise ist im konkreten Fall festgelegt. Auf diese Weise kann man sich auf lokale Entscheidungen beschränken, die nur eine sehr konkrete Situation berücksichtigen müssen. Dies hat gewisse Parallelen zu der Softsystem Methodik von Peter Checkland, deren Kern die Systematisierung der Vorgehensweise statt die Systematisierung des betrachteten Diskursbereichs ist (vgl. [Checkland 90]). Dieser wird als zu komplex und nicht korrekt erfassbar angesehen. Ein zentraler Aspekt dieser Herangehensweise ist der Wegfall der Notwendigkeit den gesamten Diskursbereich vorab vollständig zu verstehen, alle wesentlichen Fragen vor Beginn

der eigentlichen Arbeiten zufriedenstellend beantwortet zu haben und der Bedarf eines "allwissenden Organisators".

5.3.3 Teambildung

Das Entstehen und die Veränderung des Teams, welches das Entwicklungsvorhaben durchführt ist ebenfalls bei beiden Ansätzen unterschiedlich.

Projekt Die planungsorientierte Durchführung eines Projekts benötigt ein zuverlässiges Mitwirken der beteiligten Akteure. Daher versucht man bei dieser Vorgehensweise ihre Rechte und Pflichten für die Dauer des Projekts zu fixieren und möglichst noch vertraglich festzuschreiben, um gegebenenfalls die Einhaltung dieser Vereinbarungen einfordern zu können.

Kooperation Die Autonomie der Beteiligten bedeutet gerade, dass sie von Verpflichtungen frei bleiben und stattdessen bei konkreten Entscheidungen für sich die individuellen Vor- und Nachteile abwägen. Dies schließt sowohl den Beitritt zu wie auch das Verlassen von dem Entwicklungsvorhaben ein. Es steht ihnen also frei sich bereitgestelltes Wissen anzueignen und entsprechende Arbeiten durchzuführen. Damit sie jedoch auch von den anderen Akteuren wahrgenommen werden, müssen sie vom Leiter in der jeweiligen Rolle anerkannt sein.

5.3.4 Leitungsfunktion

Bei Projekten gibt es das Projekt-Management, bei dem kooperativen Vorgehen gibt es die Leiter-Rolle. Beide Elemente haben eine gewisse Steuerungsfunktion, jedoch werden unterschiedliche Bereiche gesteuert.

Projekt Das Projekt-Management ist verantwortlich für den gesamten Prozess und die daraus entstehenden Ergebnisse. Der Manager kann und soll darin eingreifen und den durchführenden Weisungen erteilen. Er hat diesbezüglich eine Überwachungs-, Kontroll-, und Steuerungsfunktion.

Kooperation Im Gegensatz dazu selektiert der Leiter nur Ergebnisse und nimmt keinen Einfluss auf die zugehörigen Prozesse selbst. Ihre Produktion ist ausschließlich in der Verantwortung der durchführenden Akteure.

5.3.5 Rolle der Arbeitsumgebung

In beiden Fällen werden die Aktivitäten im Normalfall von einer technischen Infrastruktur unterstützt. Bei einem Projekt ist sie jedoch nur der verlängerte Arm des Projekt-Managements während sie bei einem kooperativen Vorgehen eine gewisse Ordnungsfunktion übernimmt.

Projekt In einem Projekt ist es die Aufgabe des Projekt-Management dafür zu sorgen, dass eine geeignete Arbeitsumgebung für alle Beteiligten vorhanden ist, damit die verfügbare Arbeitskraft effizient genutzt wird. Ihre Beschaffung und Administration kann entsprechend an geeignete Teilnehmer delegiert werden, liegt aber wie alles andere auch in der Verantwortung des Projekt-Management. Die Infrastruktur kann dabei auch genutzt werden, um Management-Aufgaben zu automatisieren.

Kooperation Hier gilt für die Arbeitsumgebung für die verschiedenen Akteure letztendlich das gleiche Prinzip wie für die Entwicklungsarbeit selbst. Jeder Teilnehmer ist autonom, aber eine Zusammenarbeit ist bei der entsprechenden Interessenlage hilfreich. Diese Kooperation in Bezug auf die Arbeitsumgebung kann sich auch über mehrere Entwicklungsvorhaben erstrecken, die ähnlichen Abläufen folgen. Umgekehrt hat damit das Annehmen einer typischen Vorgehensweise eine bessere technische Unterstützung zur Folge. Aus diesen Wechselwirkungen zwischen Aktivitäten und Infrastruktur ergeben sich einerseits eine passende Infrastruktur und andererseits typische Verhaltensmuster. Auf diese Weise hat die Arbeitsumgebung eine gewisse Ordnungsfunktion bei dem kooperativen Vorgehen.

5.4 Modell der Open-Source-Entwicklung

5.4.1 Einordnung von Open-Source-Projekten

Der Begriff Open-Source-Projekt (OSP) suggeriert, dass damit eine Kategorie von *Projekten* beschrieben wird. Dies ist jedoch nicht der Fall, wenn man sich auf die gängige Definition von Projekten bezieht wie sie in Abschnitt 5.1 beschrieben wurde. Projekte dienen dem Erreichen von festgelegten Zielen in einer vorgegeben Zeit mit den bereitgestellten Mitteln. Sind die Ziele erreicht, endet das Projekt. Dabei werden diese drei Faktoren früh festlegt, i.d.R. vor Beginn, und sollten idealerweise bis zum Abschluss nicht mehr geändert werden.

Die typischen Merkmale der OSPs in Bezug auf diese Aspekte stehen damit im Widerspruch:

Zeit OSPs bestehen so lange wie es Nutzer und Entwickler gibt, die sich für die Software interessieren. Geht das Interesse zwischenzeitlich verloren, kann es ruhen und später wiederbelebt werden. Es handelt sich also um eine fortlaufende Unternehmung, die kein festgelegtes Ende hat.

Mittel Das benötigte Mittel eines OSPs ist primär die professionelle Aufmerksamkeit (vgl. [Raymond 99, S. 71]), welche für die Weiterentwicklung und Überprüfung von Ergebnissen benötigt wird. Da sie bei OSPs auf freiwilliger Teilnahme beruht, kann sie in hohem Maße variieren und ist nur schwer zu kalkulieren.

Ziel Die Ziele eines OSPs ergeben sich aus der Menge der beteiligten Benutzer und Entwickler (Gemeinschaft). Sie variieren ebenfalls erheblich, da eine Fluktuation in der Gemeinschaft meistens direkt mit einer Verschiebung der Ziele verbunden ist.

Selbst wenn es nun gelingt, OSPs als Ganzes in die Definition klassischer Projekte hinein zu zwängen, so hat man damit nur wenig gewonnen, da man sie damit soweit verfremdet, dass ihre wesentlichen Charakteristiken verloren gehen. Gerade, wenn man versucht OSPs zu anderen Projekten abzugrenzen führt die Diskrepanz zwischen diesen beiden Bedeutungen zu Unklarheiten, Missverständnissen und vermeintlichen Widersprüchen.

Allerdings gibt es auch im Bereich der Open-Source-Welt echte Projekte, die zu dem eigentlichen Projektbegriff passen. Das sind Releases und Ähnliches, für die versucht wird einen Zeitrahmen vorzugeben, Ziele zu definieren und ein gewisses Maß an Verbindlichkeit bei den Entwicklern zu schaffen. Auch wenn derartige Projekte in Realität nicht ideal ablaufen, so ist doch bei den Beteiligten i.d.R. der Wunsch nach einem solchen idealen Ablauf vorhanden. Wenn man

diesem Gedanken folgt, können OSPs als Programme angesehen werden, die als eine Menge von zusammenhängenden Projekten verstanden werden (vgl. Abs. 5.1.1). Evtl. kommt daher auch die begriffliche Unschärfe, da in manchen Bereichen Programm-Management und Projekt-Management als Synonym verwendet werden [PMBOK 00, S. 10].

Die doppelte Bedeutung des Projektbegriffs ist jedoch nicht das einzige Problem. Im Internet, insbesondere auf Hosting Plattformen wie Sourceforge, gibt es inzwischen viele tausend Projekte die man als OSPs bezeichnet. Das einzige, was sie sicher gemeinsam haben, ist der Umstand, dass sie ihre Software unter einer Open-Source-Lizenz zur Verfügung stellen. In dieser Arbeit wird der Begriff jedoch in einem engeren Sinne verwendet: Open-Source-Projekte werden hier angesehen als fortlaufende Unternehmungen, die den oben genannten Aspekten (Zeit, Mittel und Ziel) folgen. Des Weiteren wird angenommen, dass sie alle sechs Kriterien der Offenheit von Brown/Booch erfüllen (open-software, open-collaboration, open-process, open-releases, open-deployment, open-environment) (vgl. Abs. 1.3.2).

Im Rahmen von dieser fortlaufenden Unternehmung werden immer wieder Ergebnisse produziert, bei denen es sich größtenteils um Komponenten handelt, die in Prozessen erstellt bzw. bearbeitet werden wie es in Abschnitt 4.5 beschrieben wurde. Ein OSP stellt dann gewissermaßen den Kontext dar, in dem ein solcher Prozess abläuft und deckt sich damit größtenteils mit dem Konzept der Prozessdomäne (vgl. Abs. 4.5.3). Allerdings wurde im Rahmen der Prozessdomänen nicht die Akteure und ihre Handlungen behandelt. OSPs können daher verstanden werden als Prozessdomänen mit den darin agierenden Akteuren, die im Folgenden den zentralen Betrachtungsgegenstand darstellen.

5.4.2 Open-Source-Projekte als kooperative Entwicklung

In Abschnitt 5.3 wurde die projektbasierte und die kooperative Entwicklung gegenüber gestellt. Im Folgenden sollen die dabei verwendeten fünf Aspekte ebenfalls für OSPs erörtert werden:

Grundprinzip Die Akteur innerhalb eines OSPs sind i.d.R. Freiwillige, die sich aus unterschiedlichen Gründen dem Vorhaben angeschlossen haben (vgl. [Dietze 04, S. 44]). Sie behalten ihre Autonomie und es gibt auch keine zentrale Management-Instanz, die anfallende Aufgaben per Anweisung verteilen kann.

Aufgabendefinition Teilweise werden im Rahmen von OSPs zwar geplante Prozesse durchgeführt, die definierte Ziele, eine relativ stabile Entwickler-Gruppe und einen Zeitplan haben (z.B. Fertigstellung eines Release). Daneben gibt es jedoch eine Menge von Aufgaben, Beiträgen und Vorgängen, die ungeplant ablaufen und sogar teilweise erst der Leitung bekannt werden, wenn sie bereits abgeschlossen sind (z.B. eingereichter Patch). Zudem ist eine Menge der Aufgaben von Faktoren abhängig, die nicht von den Teilnehmern beeinflusst werden können (z.B. relevante externe Resultate). Geplante Ziele und Zeiträume sind oftmals mehr als vage Prognose zu verstehen und stellen keine verbindlichen Vorgaben dar. Teilweise werden Zeitpunkte auch als Abgabefrist für die Berücksichtigung von Resultaten verwendet. In diesem Fall sind jedoch i.d.R. die Ziele nicht klar festgelegt. Zudem werden Pläne verworfen, wenn sie sich als problematische herausstellen. Insgesamt kann man über OSPs sagen, dass der Anteil der länger im Voraus geplanten Vorgänge relativ gering, der Verlauf eines OSPs daher schwer vorhersehbar ist und die erstellten Ergebnisse letztendlich doch eher als die Resultierende eines komplexen Abstimmungsprozesses zu verstehen sind.

Teambildung Aufgrund der Lizenz und dem öffentlich zugänglichen Quelltext hat zunächst jeder die Möglichkeit, Änderungen an einer Open-Source-Software vorzunehmen und sich aus seiner Sicht wünschenswerte Änderungen zu überlegen. Ob diese Änderungen oder Wünsche berücksichtigt werden, hängt von dem jeweiligen OSP ab. Es muss den Urheber in seiner Rolle und damit seine Ergebnisse akzeptieren. Wird ihm der Einfluss bzw. die Gestaltung auf die zukünftige Software gewährt, kann er als Teilnehmer des Projekts angesehen werden. Aus dieser Teilnahme können zwar seitens der andern Projektmitglieder Erwartungen entstehen, die bei einer späteren Rückzug zu Konflikten führen, aber letztlich ist die Teilnahme freiwillig. Sie wird faktisch beendet, in dem derjenige keine weiteren Beiträge mehr einbringt, d.h. keine relevanten Arbeiten mehr leistet.

Leitungsfunktion Da die Teilnehmer in OSPs entsprechende Arbeiten freiwillig durchführen, kann ein Leiter zwar entsprechende Ergebnisse nach Fertigstellung als unpassend oder unzulänglich ablehnen, aber er hat weder die Möglichkeit einem Teilnehmer eine Weisung zu erteilen, noch ihm die Beschäftigung mit einem Thema oder die Durchführung von irgendwelchen Arbeiten zu verbieten. Allerdings kann er aufgrund seiner Möglichkeit der späteren Ablehnung durch entsprechende Beratungen versuchen Einfluss zu nehmen, so dass Arbeiten den Interessen des Projekts entsprechen, welches er repräsentiert.

Rolle der Arbeitsumgebung Im Open-Source-Bereich haben sich Standard-Werkzeuge herausgebildet wie sie in z.B. in Abschnitt 3.5.1 als Teil des Komponentensystems dargestellt wurden. Für OSPs sind dabei insbesondere die Hosting-Plattformen wie z.B. Sourceforge von Bedeutung, welche für viele Tausende von ihnen eine Projekt-Umgebung bietet. Sie sind inzwischen so etwas wie eine standardisierte Infrastruktur, die für alle Beteiligten eine einheitlich Schnittstelle darstellt und damit auch die entsprechenden Vorgänge vereinheitlichen. Dadurch wird es einerseits leichter sich in ein neues OSP hinein zu finden und andererseits wird damit auch die Zusammenarbeit erleichtert. Durch diese Standardisierung wird es auch den Hostern überhaupt erst möglich eine passende Plattform für so viele Projekte zur Verfügung zu stellen. Dabei spielt jedoch das einzelne Projekt und seine Bedürfnisse keine Rolle mehr. OSPs werden schon allein deswegen in eine bestimmte Struktur gedrängt. Selbst wenn nun einige Projekte davon abweichen wollen, werden viele ihrer Teilnehmer trotzdem diese Infrastruktur aufgrund ihrer Gewohnheiten bevorzugen.

Versucht man nun OSPs entsprechend den betrachteten fünf Aspekten einzuordnen, erkennt man den Widerspruch mit der projektbasierten und die Übereinstimmung mit der kooperativen Methode.

5.4.3 Aufgabenbearbeitung in der Open-Source-Entwicklung

5.4.3.1 Handlungsoptionen eines Entwicklers

Es werden zunächst die Handlungsoptionen eines Entwicklers betrachtet, wenn er eine Komponente in seinen Entwicklungsgegenstand einbaut und dabei feststellt, die Konfigurationsmöglichkeiten reichen nicht aus, um die Komponente entsprechend seinen Vorstellungen zu integrieren. Folgende drei Möglichkeiten sind stets möglich, selbst wenn es sich um eine Black-Box-Verwendung handelt (s. Abs. 3.2.4):

Anpassung der Umgebung Er verändert seine Vorstellungen und die Umgebung der Komponente so, dass sie in der bereitgestellten Form in das Konzept passt.

Meldung eines Änderungswunsches Kann oder will er seine Vorstellungen bzw. die Umgebung nicht anpassen und hat er auch nicht die Möglichkeit die Modifikationen selbst vorzunehmen, kann er dem Produzenten der jeweiligen Komponente seine Problematik schildern und versuchen ihn dazu zu bewegen, entsprechende Änderungen durchzuführen. Evtl. handelt es sich ja dabei um einen Fehler in der Komponente, die Anregung gefällt dem Produzenten oder man kann ihn für entsprechende Arbeiten bezahlen.

Austausch Ist weder die Anpassung der Umgebung noch der Komponente möglich, bleibt noch die Möglichkeit sich nach einer Alternativen umzusehen. Dies kann einerseits eine andere Komponente sein, die besser geändert werden kann oder die Anpassung der Umgebung erleichtert. Anderseits kann er sie auch vollständig durch Eigenentwicklungen ersetzen.

Im Open-Source-Bereich sind jedoch wie es in Abschnitt 3.5.3 erläutert wurde auch eine Glass-Box, Gray-Box und White-Box Verwendung einer Komponente möglich, die einem Entwickler noch weitere Handlungsoptionen einräumen:

Analyse Er kann das betrachtete Problem in dem fremden Bereich weiter untersuchen, damit dem Zuständigen Arbeit abnehmen und so die Chance und die Geschwindigkeit einer gewünschten Anpassung durch den Produzenten erhöhen. Dies erfordert Einsicht in den Sourcecode (typisch für Glass-Box).

Modifikation Er kann selbst eine entsprechende (lokale) Änderung vornehmen und zur Reduzierung von zukünftigen Integrationsaufwand bei neuen Versionen versuchen, den Zuständigen von der Übernahme der Änderungen zu überzeugen. Dies erfordert die Freiheit den Sourcecode ändern zu dürfen (typisch für Gray-Box).

Teilnahme Er kann dem Vorhaben, die sich um die entsprechende Komponente kümmern, beitreten, mitarbeiten, die gewollte Änderung einbringen und darüber hinaus dauerhaft seine Interessen bei der Weiterentwicklung des Elements vertreten. Dies erfordert eine offene Projektarbeit zu der Komponente (typisch für White-Box).

Konkurrenz Er kann für das entsprechende Element einen Ersatz entwickeln, über das er wiederum die Kontrolle hat. Somit dehnt er seinen Einflussbereich aus, aber auch seine Verantwortung und den damit verbunden Arbeitsaufwand. Dies kann durch eine Abspaltung (fork) oder auch durch eine komplette Neuentwicklung geschehen. Dies erfordert je nach Vorgehen die Möglichkeit vorhandene Komponenten klonen bzw. Konzepte, Schnittstellen oder ähnliches übernehmen zu dürfen (typisch für White-Box).

5.4.3.2 Betrachtung von Änderungswünschen

Um die damit verbunden Vorgänge besser zu verstehen, sollte man sich darüber klar werden, um welche Anliegen es sich überhaupt dabei handelt, wenn ein Entwickler mit einer Komponente unzufrieden ist. Entsprechende Änderungswünschen werden oftmals als “issue” oder “change request” bezeichnet (s.a. [Dietze 04, S. 49ff.], [Koru 04], [Johnson 03], [Suhr 07]). Viele sogenannte Bug-Tracker-Systeme verwalten solche Anliegen, die eben nicht nur ausschließlich aus Defekten bestehen. Zudem ist es manchmal schwierig, einen Fehler von einem unpassenden Verhalten in Bezug auf den lokalen Kontext zu unterscheiden, was schon in dem scherzhaften Spruch “It’s not a bug, it’s a feature.” ausgedrückt wird. Es gibt verschiedene Möglichkeiten der Kategorisierung. Die folgende Einteilung deckt den größten Teil der möglichen Änderungswünsche ab:

Fehlerbehebung (bug fix) Behebung von Defekten, die sich i.d.R. in unerwünschtem Verhalten äußern (z.B. Absturz)

Verbesserung (improvement) Optimierung von bereits unterstütztem Verhalten (z.B. Beschleunigung der Abläufe)

Erweiterung (enhancement) Erweiterung um neue Funktionalität (z.B. neuer Treiber im Kernel)

Bereinigung (cleanup) Aufräumarbeiten und Entfernung von nun inzwischen unnötiger Funktionalität (z.B. Entfernung von Code der nicht mehr verwendet wird)

Umstrukturierung (restructuring) Dies umfasst die Auftrennung einer Komponente in mehrere andere, die Verschmelzung von mehreren Elementen zu einem und andersartige Umstrukturierungen, die bestehende Komponenten anders aufteilt.

5.4.3.3 Verbindung mit dem Open-Source-Lebenszyklus

Diese genannten Kategorien von Änderungswünschen beschreiben grob die möglichen Aufgaben, die sich ergeben, wenn ein OSP die initiale Entwicklungsphase verlassen hat, die im Open-Source-Bereich eher unwesentlich ist, und die Entwicklungsarbeiten in die für Open-Source typische Wartungsphase übergegangen ist wie es in Abschnitt 4.5.1 beschrieben wurde. Der dabei entstehende Lebenszyklus wurde in Abschnitt 4.5.2 in die dort beschriebenen sieben Prozesse aufgeteilt (vgl. Abb. 5.1).

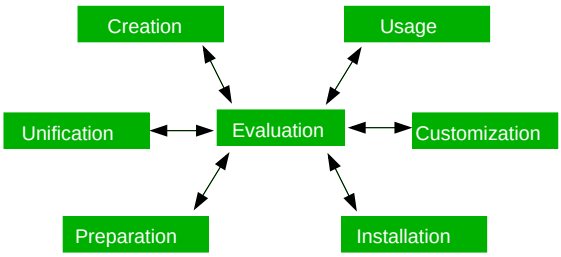

Abbildung 5.1: Meta-Prozess Evaluation im Open-Source-Lebenszyklus

Der Evalutionsprozess spielt dabei eine zentrale Rolle, da er besagte Änderungswünsche aus den anderen Prozessen sammelt und den passenden Prozessen zuführt, die entsprechende Änderungen durchführen können. Obwohl man üblicherweise nur den Vereinheitlichungsprozess als ein OSP ansieht, haben alle Prozesse (bis auf Nutzung und Evaluation) eine ähnliche Struktur, Prozessdomäne und Abläufe.

5.4.3.4 Bearbeitung eines Änderungswunsches

Schaut man sich nun die Abläufe innerhalb einer der entsprechenden Prozessdomäne an, kann man allgemeine Vorgänge identifizieren, die sich laufend wiederholen und alle Prozessarten gemeinsam haben. So folgt die Überarbeitung einer Komponente dem allgemeinen Muster, das in Abbildung 5.2 dargestellt wurde.

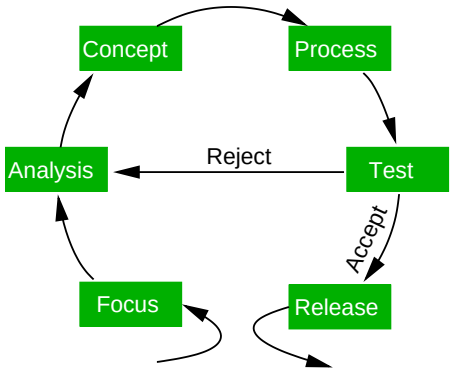

Abbildung 5.2: Überarbeitung von Komponenten

Ein Akteur wählt aus der bekannten Menge von Änderungswünschen einen bestimmten aus (focus), führt eine entsprechende Analyse der zugehörigen Komponente und des relevanten Kontexts durch (analysis), überlegt sich ein passende Lösungsstrategie (concept) und führt anschließend entsprechende Arbeiten durch (process). Dieses Ergebnis wird von ihm, der Infrastruktur oder einer weiteren Person getestet (test). Ist die Prüfung erfolgreich, wird das Ergebnis freigegeben (release). Anderenfalls muss er beginnend mit der Analyse die Arbeiten wiederholen. Vergleicht man diesen Ablauf mit dem Einheitsprozess aus Abschnitt 4.2, so zeigt sich, dass es sich hier um einen einfachen Entwicklungsvorgang handelt, der jedoch vollständig ist. *focus* und *analysis* stellen dabei die Analyse dar. *concept* repräsentiert die Konzeption. Bei *process* handelt es sich um die Realisierung und *release* ist die Ausgabe. *test* wurde hier einbezogen, um den iterativen Charakter des Vorgangs darzustellen. Überprüfungen sind beim Einheitsprozess nur implizit erfasst.

5.4.3.5 Einführung einer neuen Komponente

Die Integration von neuen Komponenten(-Versionen), die normalerweise im Rahmen des Import-Prozesses einer Prozessdomäne stattfindet, kann wie Abbildung 5.3 zeigt in ähnliche Phasen aufgeteilt werden.

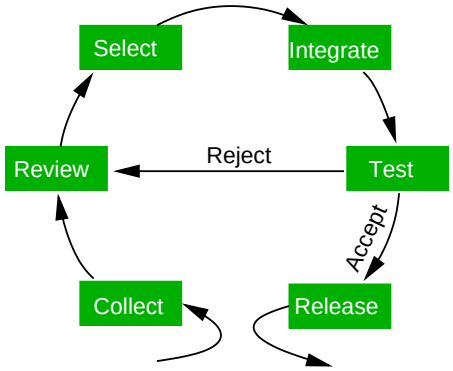

Abbildung 5.3: Integration neuer Komponenten(-Versionen)

Zunächst werden mögliche Kandidaten für eine Aufnahme in den Kontext gesammelt (collect)

und im Rahmen einer Durchsicht die jeweilige Eignung in Bezug auf wesentliche Aspekte (z.B. Nutzen, Qualität und Aufwand) überprüft (review). Entsprechend dieser Eignungsprognose findet dann eine Auswahl statt (select) und das Element wird integriert (integrate). Wird die Komponente bei der anschließenden Prüfung (test) akzeptiert, wird sie in den Bestand aufgenommen (release).

Letztendlich handelt es sich dabei nur um eine weitere Kategorie von Aufgaben, die einerseits den Austausch der gegenwärtigen Komponente mit einer überarbeiten Version und andererseits die Aufnahme einer neuen Komponente in die Prozessdomäne umfasst.

5.4.4 Prozessdomänen als Raum für kooperative Entwicklung

Das wesentlich Merkmal der Open-Source-Methodik ist hier also nicht die Organisation der Prozesse, sondern die Organisation der Prozessdomänen, welche die Einzelprozesse zu den vielen kleinen Aufgaben zu einer großen kooperativen Entwicklung zusammenführen. Die Einzelaufgaben können nicht völlig unabhängig bearbeitet werden, weil ihr Entwicklungsgegenstand durch die anderen Prozesse beeinflusst wird.

Wenn es sich bei diesen Prozessdomänen um eine Organisationsform der kooperativen Entwicklung handelt, dann kann man sie auch mit der in Abschnitt 5.2.2 beschriebenen Grundstruktur erfassen.

5.4.4.1 Prozessdomäne als wesentliche Struktureinheit

Die besagten Prozesse des Open-Source-Lebenszyklus (creation, unification, preparation, installation und customization) werden also immer wieder angestoßen, wenn es eine entsprechende Aufgabe zu erledigen gibt:

Neue Version Dies kann durch das Bereitstellen entsprechender Ergebnisse im Vorgänger-Prozess sein: Die Fertigstellung eines neuen Release durch die Vereinheitlichung stellt implizit der zugehörigen Vorbereitung die Aufgabe, diese neue Version zu bearbeiten.

Änderungswunsch Es kann sich auch um die Bearbeitung von anderen Aufgaben handeln, die am besten an dieser Stelle gelöst werden. So benötigt der Austausch einer Netzwerk-Karte kein neues Release (Vereinheitlichung) oder eine Modifikation des Treibers (Erstellung), sondern nur den Austausch der Treiber-Komponente bzw. eine andere Konfiguration (Installation).

Dabei werden zwar viele gleichartige Prozesse angestoßen, die auch vieles gemeinsam haben, aber trotzdem sind es unterschiedliche Vorgänge. Das Konstante dabei ist ein Großteil der Umgebung inklusive der beteiligten Akteure.

In diesem Sinne sind nicht die Prozesse des Open-Source-Lebenszyklus selbst die wesentliche Struktureinheit, sondern ihre Prozessdomänen und Open-Source-Projekte sind nur eine wichtige Sonderform davon. Es ist daher anzunehmen, dass alle Prozessdomänen der besagten Prozesse als ein Rahmen für eine kooperativer Entwicklung angesehen werden können.

Allerdings sind die Aufgaben teilweise so klein und speziell, dass es dann nur wenige oder sogar nur einen Akteure in einer solchen Prozessdomäne gibt. So befasst sich z.B. meistens nur

der Nutzer selbst mit der Anpassung einer Komponente an seine Bedürfnisse und es gibt oft nur einen Administrator der sich mit der Installation beschäftigt. Die Struktur und die Abläufe bleiben dabei jedoch die gleichen.

5.4.4.2 Objekte

In den Open-Source-Prozessen gibt es zwei wesentliche Objekt-Typen: Aufgaben und Komponenten. In der kooperativen Entwicklung gibt es ebenfalls Aufgaben, die eine übereinstimmende Bedeutung haben. Versionen, Änderungen und Release können jeweils als unterschiedliche Ausprägungen einer Komponente aufgefasst werden.

Dabei müssen entsprechende Versionen nicht in einer einfachen Verkettung zusammen hängen. Im Open-Source-Bereich bilden sie oftmals komplexere Strukturen. Typisch sind mehrere Entwicklungszweige, die z.B. eine unterschiedliche Stabilität der aufgenommen Änderungen kennzeichnen.

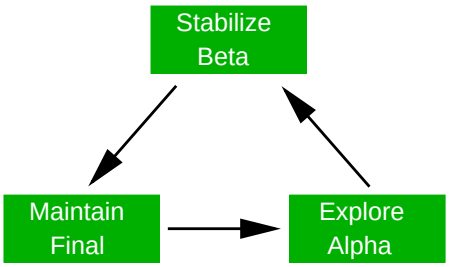

Abbildung 5.4: Unterschiedliche Zweige eines Projekts

Bei diesem Ansatz werden i.d.R. drei Bereiche unterschieden (vgl. Abb. 5.4):

Alpha Änderungen werden zunächst in einem experimentellen Zweig ausgetestet und ihre Eignung für die jeweilige Umgebung ausgetestet.

Beta In einem nächsten Zweig werden alle Änderungen übernommen, die grundsätzlich als geeignet angesehen werde und man versucht Fehler und Probleme aus den daraus erstellten Versionen zu eliminieren.

Final In diesem Zweig werden Änderungen und daraus erstellte Versionen gesammelt, bei denen man davon ausgeht, dass sie keine wesentlichen Mängel mehr enthalten. Die Fortsetzung eines solchen Zweigs kann einerseits die Beseitigung von später entdeckten Fehlern sein oder auch eine neue, bereits stabilisierte Verbesserung darstellen.

5.4.4.3 Aktivitäten und Rollen

Anregen (Aufgabensteller) Darunter kann das Erstellen von Änderungswünschen und auch die Fertigstellung einer neuen Komponentenversion des Vorgänger-Prozessen verstanden werden. Diese Aktivität wird von jenen Akteuren wahrgenommen, die Nutzer der veröffentlichten Komponente (Release) oder erstellter Versionen (Version) sind, d.h. entsprechende Ergebnisse in irgendeiner Weise nutzen. Dabei kann es sich um Akteure in allen nachfolgenden Prozessen inkl. der Nutzung handeln. Die Transformation bzw. Weiterentwicklung ist hier ebenfalls gemeint.

Gestalten (Entwickler) Das Erstellen und die Integration einer Änderung von einer Komponente, die Integration von Ergebnissen aus Vorgänger-Prozessen und die Vorbereitung zur Veröffentlichung einer Komponente werden hier als Gestalten aufgefasst. Als Entwickler wird folglich jeder angesehen, der entsprechende konstruktive Arbeiten in dieser Prozessdomäne ausführt.

Lenken (Leiter) Es gibt in jedem Open-Source-Projekt und auch den anderen Prozessdomänen stets eine Instanz, die bestimmte Entscheidungen trifft. Diese Rolle kann durch eine Gruppe (z.B. alle Kern-Entwickler) oder eine Einzelperson (z.B. Package-Maintainer oder Projektleiter) wahrgenommen werden.

5.4.4.4 Arbeitsweise

Wie in Abschnitt 5.2.3 erläutert, beinhaltet die kooperative Entwicklung mit ihrer Trennung von Produktion und Selektion bereits einen grundlegenden Mechanismus für die Zusammenarbeit. Dieser ist jedoch aufgrund des hohen Verlusts von Arbeitsleistung durch die vielen Ablehnungen alleine nicht effektiv.

Im Open-Source-Bereich können derartige Selektionsmechanismen beobachtet werden, die von der Produktion getrennt sind. So entwickeln die Nutzer einer Open-Source-Software oftmals ihre eigenen Änderungen und schicken sie anschließend an das zugehörige Open-Source-Projekt, das dann über die Integration entscheiden muss. Auch Distributoren und Administratoren entscheiden sich für die Version und zugehörige Modifikationen einer Software, die für ihre Zwecke am besten geeignet sind ohne das deren Entwicklung vorher mit ihnen abgestimmt wurde.

Es können jedoch auch eine Menge zusätzlicher Mechanismen beobachtet werden, welche die Zahl der Ablehnungen reduziert:

Informationsmaterial Es gibt für die meisten Tätigkeiten Dokumente, welche die Sicht der Leitung einer Prozessdomäne widerspiegelt. Dies können z.B. Standards, Richtlinien, Dokumentation oder auch archivierte Äußerungen der Entwickler sein.

Kommunikation Viele Punkte können durch entsprechende Fragen und Diskussionen im Vorfeld geklärt werden ohne vorher größeren Aufwand investiert zu haben.

Gewohnheit Es haben sich im Open-Source-Bereich mit der Zeit einige allgemein akzeptierten Verhaltensmuster und Strukturen herausgebildet, welche die Einarbeitung und Zusammenarbeit in neuen Prozessdomänen erleichtert. Ist man damit zunächst nicht vertraut, erlernt man sie durch das Sammeln entsprechender Erfahrungen. Ein gutes Beispiel dafür ist der Ablauf der Kommunikation über Mailinglisten.

Minimal-Änderungen Es ist üblich Änderungen so klein wie möglich zu halten. Dies gilt bezüglich des Umfangs, der Auswirkungen und auch der Entwicklungszeit. Dies minimiert u.a. das Risiko von Konflikten mit anderen Modifikationen und Komponenten.

Dies alles sind Beispiele für Mechanismen, die für eine Steigerung der Effizienz in der Open-Source-Entwicklung sorgen können und es gibt noch eine Menge anderer Möglichkeiten dafür. Welche Methoden dafür nun tatsächlich in einer Prozessdomäne verwendet werden, hängt vom Einzelfall ab. Dies ist auch sinnvoll, da die Unterschiede in Bezug auf Teilnehmerzahl, Thematik, Nutzerzahl, Größe des Entwicklungsgegenstands und vielen andern Faktoren enorm ist.

5.4.4.5 Lebenszyklus einer Prozessdomäne

Eine derartige Prozessdomäne taucht nicht plötzlich in vollem Umfang aus dem Nichts auf und verschwindet dann wieder auf die gleiche Weise. Stattdessen folgt sie oft dem in Abbildung 5.5 dargestellten Lebenszyklus, der vor allem bei Open-Source-Projekten zu beobachten ist (s. a. [Evers 00, Abs. 2.6.2] u. [Dietze 04, S 41ff.]). Er kann aber letztendlich auf alle Prozessdomänen übertragen werden, wobei die Phasen dort teilweise weniger ausgeprägt sind.

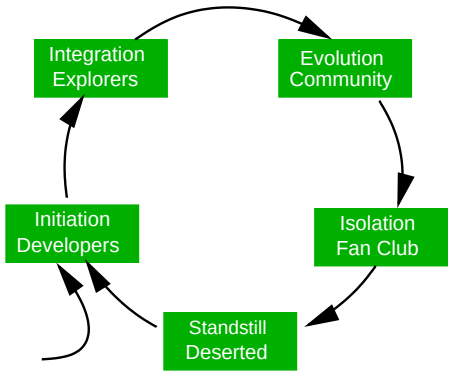

Abbildung 5.5: Lebenszyklus einer Prozessdomäne

Der Lebenszyklus lässt sich in folgende Abschnitte unterteilen:

Initiale Entwicklung (initiation) Zunächst bündelt nur ein Zusammenschluss von Entwicklern ihre Vorstellungen und Ideen und erstellt eine erste Version, die grob das verfolgte Ziel des Vorhabens erkennen lässt.

Vernetzung (integration) Danach wird dieses Ergebnis in das Netzwerk von Prozessdomänen eingebettet und entsprechende Verbindungen hergestellt. In dieser Phase ist es wichtig, Nutzer und Entwickler zu finden, die sich aufgrund ähnlicher Bedürfnisse für die veröffentlichten Ergebnisse so sehr interessieren, dass sie sich zumindest teilweise an der kooperativen Entwicklung beteiligen, um ihre individuellen Interessen zu verfolgen. Dabei kommt es zu einer schrittweisen Verbesserung der fokussierten Komponente und im Idealfall führt dies zu steigendem Engagement und Zahl der Beteiligten in der Prozessdomäne.

Evolution (evolution) Durch eine stetige Verbesserung kann sich so eine ausreichend große Nutzer- und Entwickler-Gemeinde bilden und durch die entsprechenden Anregungen und Gestaltungen eine gebräuchliche Komponente entwickeln. Diese wird dann in einer kooperativen Entwicklung fortlaufend verbessert, an die sich ständig wandelnde Umgebung angepasst und in ihrem Funktionsumfang überarbeitet. So lange ein entsprechendes Interesse vorhanden ist, verharrt eine Prozessdomäne in dieser Phase.

Abseits (isolation) Lässt das Interesse an den resultierenden Komponenten allmählich nach, gerät die entsprechende Prozessdomäne zunehmend ins Abseits und nur der harte Kern der Gemeinde verbleibt. Gründe für ein schwindendes Interesse können alternative Komponenten sein, welche für die Bedürfnisse der früheren Nutzer besser geeignet sind, oder eine konkurrierende Prozessdomäne, die ein angenehmere Arbeiten für die selbe Komponente bietet.

Stillstand (standstill) Verliert auch der harte Kern das Interesse an einer weiteren Nutzung bzw. Entwicklung kommen die Vorgänge in der Prozessdomäne zum Stillstand und die letzten Ergebnisse veralten, d.h. sie verlieren die Möglichkeit in der vorliegenden Form genutzt zu werden. Dies bleibt so lange bestehen, bis sie durch ein erneutes Interesse von einer initialen Entwicklergruppe in einen brauchbaren Zustand gebracht werden und der Zyklus wieder von vorne beginnt.

6 Kontextmodell-Perspektive

Die relevante Umgebung einer Software kann stets nur in Form von Modellen erfasst werden. Die Abgrenzung des wesentlichen Kontexts, das Erkennen der enthaltenen Strukturen, die Repräsentation dieser Erkenntnisse und ihre anschließende Verwendung sind der Betrachtungsgegenstand der Kontextmodell-Perspektive.

In der klassischen Softwaretechnik werden diese Vorgänge primär dem Requirements-Engineering zugeordnet (z.B. [Rupp 07]), das auch als Anforderungsanalyse bezeichnet wird. Diese Phase ist in der klassischen Abfolge der eigentlichen Entwicklung vorgelagert. Bei den resultierenden Modellen handelt es sich um Spezifikationen, die meistens mittels einer formalisierten Sprache dargestellt werden (z.B. Unified Modeling Language) und oftmals den relevanten Kontext vollständig abbilden sollen.

Im Open-Source-Bereich werden i.d.R. keine Spezifikationen angefertigt und die Analyse der Umgebung wird letztlich über die ganze Entwicklung hinweg fortgesetzt. Trotzdem muss man auch hier eine Vorstellung von der Umgebung haben, um Entwicklungsarbeit leisten zu können. Es stellt sich daher die Frage, wie man ohne eine Spezifikation Software erstellen kann, die dann auch noch funktioniert, eine gewisse Qualität aufweist und in die zugehörige Umgebung passt. Letztlich setzt dies voraus, dass man auch hier in irgendeiner Form Kontextmodelle verwendet, auch wenn es sich dabei um keine expliziten, formalisierten Dokumente handelt.

Im Folgenden werden die grundlegenden Probleme von Kontextmodellen und Konzepte für ihre Bewältigung untersucht. Dabei wurde als allgemeine Grundlage die Modelltheorie von Bernd Mahr verwendet, die u.a. in folgenden Arbeiten dargestellt wurde:

- Gegenstand und Kontext - Eine Theorie der Auffassung [Mahr 97]
- Modellieren. Beobachtungen und Gedanken zur Geschichte des Modellbegriffs [Mahr 03]
- Das Wissen im Modell [Mahr 04]
- Ein Modell der Auffassung [Mahr 06]
- Ein Modell des Modellseins - Ein Beitrag zur Klärung des Modellbegriffs [Mahr 08]

Das nachstehende Kapitel ist vor diesem Hintergrund zu sehen.

6.1 Softwareentwicklung und Erkenntnis

6.1.1 Bedeutung der Erkenntnis in der Softwareentwicklung

Programme stellen Prozessmodelle dar. Sie beinhaltet klare Anweisungen, wie ein Computer sich in einem vorher bereits bis ins kleinste Detail durchdachten Szenario verhalten soll. Bis heute ist es immer noch so, dass Computersysteme nur zu Reaktionen fähig sind, die ihnen durch die

Programme vollständig vorgegeben sind. Sie sind nicht wie der Mensch fähig zu improvisieren bzw. sich selbstständig an auch noch so kleine Veränderungen der Umgebung anzupassen. Da bei der Entwicklung aber immer nur ein kleiner Teil aller möglichen Interaktionen mit der Umwelt berücksichtigt werden kann, beschneidet Software die Handlungsoptionen der Umgebung in einem erheblichen Maße, soweit die Nutzung des Computersystems notwendig ist.

Je weiter Computersysteme jedoch in den menschlichen Alltag eindringen und je mehr Dinge mit Hilfe von elektronischen Datenverarbeitungssystemen erledigt werden, um so größer wird das Spektrum an Interaktionen, die unsere Computer bewältigen sollen. Denn es gilt nicht nur, einzelne isolierte Aktionen auszuführen, sondern Vorgänge evtl. über Jahre hinweg zu begleiten, an denen teilweise mehrere tausend Menschen beteiligt sind.

Früher waren es primär kleine, einfache Rechenaufgaben, die von diesen "Rechenmaschinen" erledigt wurden. Heute sind die Aufgaben ganz andere: sie verteilen Geld, überwachen Gebäude, übermitteln Nachrichten, suchen Verbrecher, regeln den Verkehr, erstellen Rechnungen und helfen uns, einen neuen Lebensgefährten zu finden. Zudem sind moderne Computer in viele Gegenstände unseres Alltags integriert: Autos, Haushaltsgeräte, Unterhaltungselektronik, Handys, Flugzeuge, Fertigungsstraßen, etc. Es spielt eher eine nebensächliche Rolle, dass dabei irgendwo auch gerechnet wird. Viel wichtiger sind inzwischen die Prozessmodelle, die in der jeweiligen Software all dieser Geräte festgelegt sind und dem Computer genau sagen, wie er sich in der jeweiligen Situation verhalten soll. Denn ist ein Computersystem für eine Situation, mit der es konfrontiert wird, nicht ausreichend vorbereitet, weil seine Software dies nicht vorsieht bzw. eine unpassende Reaktion erzwingt, so führt dies im besten Fall zu einem frustrierten Nutzer, im schlimmsten Fall zu menschlichen und wirtschaftlichen Katastrophen.

Selbst die Benutzer sind heute andere als früher. Hatte noch vor wenigen Jahrzehnten ausschließlich geschultes Fachpersonal Zugriff auf Computer, dienen heute manche Geräte schon als Spielzeug für Kleinkinder. Aber auch der ganz normale Personalcomputer wird heute von Menschen bedient, die wenig über die Geräte wissen und von denen man keine besondere Anpassung an die Technik erwarten kann. Das Verhältnis zwischen Softwarehersteller und Benutzer wird dadurch verändert. Früher reichte es bei der Entwicklung von Standardsoftware aus, durch entsprechende Studien einen sinnvollen Ablauf in die Software zu kodieren, durch entsprechende Konfigurationsmöglichkeiten eine gewisse Anpassungsfähigkeit zu ermöglichen und das Ergebnis anschließend zur Verfügung zu stellen. Es war dann dem Benutzer überlassen, durch eine sinnvolle Auswahl aus dem vorhandenen Angebot und einer Anpassung der Umgebung an die Bedürfnisse der Software für einen reibungslosen Betrieb zu sorgen. Heutige Benutzer sind dazu oftmals nicht in der Lage, da ihnen das entsprechende Hintergrundwissen fehlt. Für einen effektiven Einsatz von Computersystemen benötigen sie Software, die exakt an ihre Erfordernisse angepasst ist. Waren es früher die Nutzer, die sich an die Geräte anpassten, so wird gegenwärtig mehr und mehr verlangt, dass die Geräte sich nach den Benutzern richten.

Es scheint daher heute eine wesentliche Aufgabe der Softwareentwicklung zu sein, für eine möglichst optimale Anpassung der Computersysteme an ihre Umgebung und vor allem die Bedürfnisse ihrer Benutzer zu sorgen.

Andererseits wird heute durch die zunehmende Globalisierung und weltumspannende Vernetzung der Computersysteme Software in einem weltweiten Kontext entwickelt, und man kann sich nicht mehr in allen Belangen auf eine bestimmte Region beschränken. Dies gilt oft selbst dann, wenn das Programm nur für eine örtlich begrenzte Zielgruppe entwickelt wird, weil es

meistens trotzdem mit global operierenden Systemen zusammenarbeiten muss. Die weltweiten Unterschiede bezüglich der Kultur, Sprache, Klima, Wirtschaft, Bildung, Infrastruktur, etc. erfordern aber dann auch eine entsprechende Vielfalt in der eingesetzten Software, wenn man sie nicht einfach ignoriert und damit wieder die Lösung dieser Probleme dem Nutzer überlässt. Wenn z.B. eine Software nur mit Englischkenntnissen zu bedienen ist, werden damit alle Nutzer gezwungen, Englisch zu lernen. Bei der Entwicklung spart man sich damit aber die Integration mehrerer Sprachen.

Betrachtet man nun die Lebenszeiten von Daten und Systemen, so stellt man fest, dass es seitens der Nutzer eine gewisse Trägheit gibt. Oftmals sind sie nicht dazu bereit, auf neuere Applikationen umzusteigen, sondern überspringen mehrere Entwicklungsschritte, bevor sie ihre Software erneuern. Normalerweise sind die Gründe dafür Vermeidung von zusätzlichen Kosten und Risiken, die mit einer Erneuerung der Software zusammenhängen. Man kann jedoch den Kunden keine Vorschriften machen, *welche* Generationen sie überspringen und als Ergebnis sind zu einem bestimmten Zeitpunkt oft viele unterschiedliche Versionen eines Programms im Einsatz. Dies trägt zu einer weiteren Heterogenität der Systeme bei.

Alle bisher genannten Faktoren machen es schwer, wenn nicht unmöglich, für eine zukünftige Software genau vorherzusagen, welche Funktionalität sie haben soll, wie die Umgebung aussehen wird, in der sie eingesetzt wird, und welche Wünsche und Bedürfnisse der potentielle Nutzer haben wird. Dies gilt schon bereits für den Zeitpunkt der Fertigstellung, aber es wird um so schwieriger, je länger die Software für den Einsatz vorgesehen ist.

Das besagte Unvermögen von Computersystemen sich an neue, unerwartete Umstände anzupassen, lässt jedoch Entwicklern keine Wahl. In der Software muss genau angegeben werden, was später passieren soll. Dazu muss man sehr genau wissen, wie die Umstände zur Laufzeit aussehen werden. Man braucht ein nahezu prophetisches Wissen. Meistens wird versucht, dies durch ausgedehnte, intensive Bedarfsanalysen vor der tatsächlichen Nutzung zu erreichen, d.h. man erstellt vorab mit viel Aufwand ein passendes Modell von der späteren Umgebung und den Abläufen und kodiert diese Erkenntnisse dann in die Software.

6.1.2 Stufenmodell nach Holl

Von der initialen Idee eine Software zu bauen bis zu ihrer Fertigstellung ist also ein erheblicher Erkenntnisprozess erforderlich. Dieser Erkenntnisprozess und die damit verbundene Modellbildung wird jedoch laut Holl/Krach in der wissenschaftlichen Auseinandersetzung mit der Softwareentwicklung unzureichend thematisiert. In ihrer Studie der entsprechenden Literatur kommen sie zu folgendem Ergebnis [Holl 02b]:

> Erkenntnistheoretische Fragestellungen werden in der Literatur wenn überhaupt - oft unsystematisch bzw. unreflektiert dargestellt. Obwohl einige positive Ansätze vorhanden sind, lassen viele Aussagen eine naive Haltung gegenüber erkenntnistheoretischen Fragestellungen vermuten. Eine Entwicklung hin zu einem reflektierten und systematisierten Umgang mit erkenntnistheoretischen Problemen wäre wünschenswert.

Darin belegen sie, dass in der Informatik oftmals (unbewusst) als erkenntnistheoretischer Klärungsansatz ein naiver Realismus verwendet wird, den Holl jedoch “für Fragestellungen der

Wirtschaftsinformatik [als] nicht brauchbar" ansieht [Holl 99]. Dies begründet sich u.a. in dem von Feistner/Holl beschrieben Problem, dass bei der Softwareentwicklung die vielen unterschiedlichen Perspektiven aller Beteiligten zu einem Gegenstand - nämlich der Software - vereinheitlicht werden müssen (vgl. [Feistner 06]).

Holl schlägt als Alternative ein Stufenmodell vor, das folgende drei Ansätze kombiniert [Holl 02b, Holl 99]: naiver Realismus, kritischer Realismus und evolutionäre Erkenntnistheorie. Dabei sieht er die evolutionäre Erkenntnistheorie "als eine kohärente, aufwärtskompatible Erweiterung" des kritischen Realismus und diesen ebenfalls "als eine kohärente, aufwärtskompatible Erweiterung" des naiven Realismus [Holl 99].

An anderer Stelle erklärt er die Verwendung eines solchen Stufenmodells [Holl 02a]:

> Das Verhalten der Natur ist in Teilbereichen analogisch interpretierbar, in anderen nicht, sondern statistisch oder chaotisch erklärbar. Wesentlich für den Menschen in seiner jeweiligen historischen Situation ist nun, zwischen diesen Bereichen erkenntnistheoretisch unterschiedlicher Qualität differenzieren zu können und das eigene Verhalten darauf einzustellen. Erkenntnistheoretisch trägt man diesem Anspruch mit einem Stufenmodell Rechnung [Holl 99, 186-188]. Dazu löst man sich von dem Gedanken, dass eine einzige erkenntnistheoretische Auffassung für alle Erkenntnisgegenstände gleichermaßen gilt, sondern wendet sich mit deren zunehmender Komplexität und abnehmender analogischer Vergleichbarkeit vom naiven Realismus über den kritischen Realismus (gemäßigten Konstruktivismus) zum Solipsismus (radikalen Konstruktivismus) bei singulären Phänomenen.

Laut Holl sollten in der Informatik also alle drei Ansätze (naiver bzw. kritischer Realismus und evolutionäre Erkenntnistheorie) gemeinsam genutzt werden. Wobei man bei einem konkreten Problem in Abhängigkeit von Erkenntnisobjekt und Fragestellung den einfachsten, geeigneten Ansatz verwendet [Holl 99, S. 14].

6.1.3 Stufenmodell nach Vollmer

Vollmer, auf den sich Holl bezieht, charakterisiert in seinem Buch über die Evolutionäre Erkenntnistheorie die verschiedenen Varianten des Realismus mit folgenden Worten [Vollmer 90, S. 35]:

Naiver Realismus "Es gibt eine reale Welt; sie ist so beschaffen, wie wir sie wahrnehmen."

Kritischer Realismus "Es gibt eine reale Welt; sie ist aber nicht in allen Zügen so beschaffen, wie sie uns erscheint."

Streng kritischer Realismus "Es gibt eine reale Welt; aber keine ihrer Strukturen ist so, wie sie uns erscheint."

Hypothetischer Realismus "Wir nehmen an, dass es eine reale Welt gibt, dass sie gewisse Strukturen hat und dass diese Strukturen teilweise erkennbar sind, und prüfen, wie weit wir mit diesen Hypothesen kommen."

"Der hypothetischen Realismus ist hinsichtlich des Geltungsanspruchs seiner Aussagen über das Bestehen und die Struktur der Welt schwächer als die übrigen Realismen" [Vollmer 90, S.

35]. “Die Einsicht, dass die Existenz der Welt ’da draußen’ nicht beweisbar ist” [Vollmer 90, S. 35], führt zur “Realitätshypothese”. Laut dem hypothetischen Realismus ist also *alles* eine Hypothese, auch die Existenz der Realität selbst. Man kann in der vorangegangen Aufzählung von oben nach unten eine zunehmende Infragestellung der Verlässlichkeit der gefunden Erkenntnisse erkennen, d.h. der Zweifel bzw. die Unsicherheit steigen und das Vertrauen in die (direkte) Erkenntnisfähigkeit, in die gemachten Erfahrungen und in die darauf basierenden Aussagen sinkt.

Vernachlässigt man den streng kritischen Realismus, dann handelt es sich bis auf den Austausch des hypothetischen Realismus mit der Evolutionären Erkenntnistheorie um die Elemente von Holls Stufenmodell. Es wird aus seinem Aufsätzen nicht deutlich, warum Holl diesen Austausch vornimmt, der zudem die Systematik durchbricht: die verschiedenen Formen des Realismus stellen unterschiedliche erkenntnistheoretische Grundpositionen in der Philosophie dar, über welche die Evolutionäre Erkenntnistheorie schon begrifflich hinaus geht. Sie umfasst bereits eine ganze Reihe von Hypothesen, wie die Welt beschaffen ist, wie sich der heutige Erkenntnisapparat der Menschen durch die Evolution über Jahrmillionen entwickelt hat und die Begründung seiner (Fehl-)Leistungen beim Erkennen der “Realität” durch seine Funktion als “Überlebenshilfe” der menschlichen Art (vgl. [Vollmer 90]). Die dazugehörige Grundposition ist der hypothetische Realismus, den Holl wiederum dem kritischen Realismus zuordnet [Holl 02b, S. 3]:

> Die von Vollmer vorgenommene Differenzierung in streng kritischen Realismus und hypothetischen Realismus (vgl. [Vollmer 90, S. 34 f.]) ist hier von untergeordneter Bedeutung, wir fassen sie als Ausprägungen des kritischen Realismus auf.

Aus seinen Ausführungen geht daher nicht klar hervor, um was für ein Aufzählung es sich bei seinem Stufenmodell handelt, d.h. was die enthaltenen Stufen/Elemente charakterisiert und worin der Unterschied zwischen kritischem Realismus und evolutionärer Erkenntnistheorie diesbezüglich liegt.

Dies kann jedoch vielleicht mit Hilfe der dahinter liegenden Motivation geklärt werden. Holl schreibt am Anfang seines Aufsatzes [Holl 02b, S. 2]:

> Der Modellbildungsprozess kann dann als erkenntnisgewinnender Prozess interpretiert werden. Daher liegt es nahe, erkenntnistheoretische Fragestellungen auf den Modellbildungsprozess zu übertragen.

Weiter schreibt er in dem zugehörigen Resümee [Holl 02b, S. 13]:

> Da [Wirtschaftsinformatik] und Modellbildung untrennbar miteinander verbunden sind, ist ein erkenntnistheoretisch reflektierter Umgang mit dem Modellbildungsprozess im Spannungsfeld zwischen Realität und Modell als eine der Kernaufgaben der [Wirtschaftsinformatik] anzusehen.

Ist Holls primäres Interesse also die Untersuchung bzw. bessere Fundierung des Modellbildungsprozesses, so liegt die These nahe, dass die Elemente Methodiken repräsentieren, wie man Erkenntnisse über die Welt bzw. einen Betrachtungsgegenstand erhält. Dazu gehören auch die zugrundeliegende erkenntnistheoretische Sichtweisen (die Realismen) und die daraus resultierenden Beschränkungen der Erkenntnisfähigkeit. Die Methodik wird in seinem Stufenmodell jedoch bei dem naiven und kritischen Realismus nur implizit referenziert und erst die evolutionären Erkenntnistheorie umfasst u.a. auch Erkenntnisstrategien.

Eine weitere erkenntnistheoretische Einteilung sind die drei durch Vollmer unterteilten Erkenntnisstufen [Vollmer 90, S. 41, 120, 124]: Wahrnehmungserkenntnis (unbewusst), Alltagserkenntnis (auch Erfahrungserkenntnis; bewusst, unkritisch) und wissenschaftliche Erkenntnis (bewusst und kritisch). "Die unkritische Haltung ist zugleich die des naiven Realisten" [Vollmer 90, S. 124]. Die kritische Haltung bei der wissenschaftlichen Erkenntnis umfasst dann den kritischen (inklusive den hypothetischen) Realisten und damit auch jemanden, der der evolutionären Erkenntnistheorie folgt. Vollmer grenzt Erfahrungs- und wissenschaftliche Erkenntnis wie folgt voneinander ab [Vollmer 90, S. 162]:

> Wissenschaftliche Erkenntnis zeichnet sich also vor der Erfahrungserkenntnis u. a. dadurch aus, dass sie *kritisch* ist, dass sie sich des hypothetischen Charakters ihrer Sätze und Gesetze (wenigstens in unserem Jahrhundert) bewusst ist und dass sie dank ihren verbesserten Beobachtungs-, Experimentier-, und Schlußmethoden eine viel größere induktive Basis für ihre Hypothesen besitzt.

Vollmer stellt direkt im Anschluss an diesen Text folgende Frage:

> Wie ist diese unterschiedliche Leistungsfähigkeit der verschiedenen Erkenntnismethoden zu erklären?

Dabei ist für die Betrachtung hier nicht die Frage oder ihre Antwort von Bedeutung, sondern die darin enthaltene, implizite Verbindung von Erkenntnisarten, Erkenntnismethoden und ihrer Leistungsfähigkeit, die Vollmer herstellt. Er unterscheidet also drei Erkenntnisarten, die auf verschiedenen Erkenntnismethoden beruhen, die wiederum unterschiedliche Leistungsfähigkeit besitzen. Verbindet man die Leistungsfähigkeit noch mit den philosophischen Grundpositionen und ihren Aussagen, dann kann man die Leistungsfähigkeit daran messen, welche Welt bzw. welche Strukturen einer Welt sie erkennen kann.

6.1.4 Erkenntnisstrategien in der Informatik

Sowohl Holls wie auch Vollmers Stufenmodell führen also zu einer Unterscheidung verschiedener Erkenntnismethoden. Die Idee eines solchen Stufenmodells und auch den entsprechenden erkenntnistheoretischen Hintergrund nutze ich im Folgenden für die Aufstellung eines Stufenmodells für die Charakterisierung unterschiedlicher Erkenntnismethoden in der Informatik.

Die erste Stufe in Holls Modell ist der naive Realismus, der in Vollmers Einteilung der Alltagserkenntnis entspricht. Die Wahrnehmungserkenntnis als unbewusster Vorgang ist für meine Zwecke vernachlässigbar. Daher betrachte ich die Alltagserkenntnis als die untere Grenze. Die wissenschaftliche Erkenntnis wird als die obere Grenze angesehen.

Von der alltäglichen Erfahrungserkenntnis bis zur wissenschaftlichen Erkenntnis nimmt der Anspruch erheblich zu, da man zunehmend die offensichtlichen Erfahrungswerte hinterfragt, tieferliegendere Strukturen sucht und entsprechende Anstrengungen unternimmt, um möglichst stabile Aussagen zu erhalten, die dem betrachteten System dauerhaft zugrundeliegen. Daraus resultiert eine Ausdehnung des Erkenntnisgegenstands, eine damit verbundene, höhere Komplexität und eine zunehmende Entfernung zum Erfahrungsbereich, was die Entwicklung und Überprüfung von nützlichen Ergebnissen wesentlich erschwert und auch die Anfälligkeit für Inkonsistenzen und Irrtümer erhöht, deren Vermeidung dann eine höhere Präzision und Skepsis bei der

Erkenntnisfindung erfordert. Dies hat einen erheblichen Mehraufwand als Konsequenz, der sich durch den Umstand potenziert, dass die Organisation der Erkenntnisvorgänge um so schwieriger wird, je umfangreicher sie werden. Dabei geben u.a. folgende Aspekte Anhaltspunkte für den Umfang der zu erbringenden Erkenntnisarbeit:

Zeit Die Zeitspanne, die für den Erkenntnisprozess investiert wird.

Beteiligte Die Anzahl der Personen, die an dem Erkenntnisprozess mitwirken und deren Erfahrungen, Wissen und Sichtweisen bei den Ergebnissen berücksichtigt werden.

Diskursbereich Die Ausdehnung und der Detaillierungsgrad des Betrachtungsgegenstands, über den Erkenntnisse gesammelt werden. Der Gegenstand kann singulär sein (z.B. Warenwirtschaftssystem von einer bestimmten Firma), eine Menge von Elementen enthalten (z.B. Warenwirtschaftssysteme von allen mittelständische Unternehmen in Deutschland) oder auch sehr abstrakt sein (z.B. Warenwirtschaftssysteme).

Betrachtet man nun das Vorgehen in der Informatik, dann kann man sie innerhalb dieser Grenzen in drei wesentliche Vorgehensweisen einteilen:

Direkte Erkenntnis Man geht (wie ein naiver Realist) davon aus, dass der betrachtete Diskursbereich genau so ist, wie man ihn wahrnimmt und handelt mit dieser Gewissheit zielstrebig und fokussiert. Man beschränkt sich auf das Offen-Sichtliche, ohne die dahinter liegenden Strukturen zu untersuchen und die eigene Wahrnehmung bzw. Erfahrung zu hinterfragen. Diese Methode wird oft beim ungeplanten Entwickeln von kleinen Programmen verwendet.

Analytische Erkenntnis Man verlässt sich nicht auf die “einfache” Wahrnehmung, sondern versucht das betrachtete System aus mehreren Perspektiven zu sehen, diskutiert seine Erfahrungen mit anderen, hinterfragt sie, sucht nach grundlegenden Strukturen und versucht gefundene Informationen zu überprüfen. Es handelt sich dabei um die Erledigung von konkreten Aufgaben: die Einrichtung eines Arbeitsplatzes für einen Büroangestellten, die Bereitstellung einer bestimmten Funktionalität für eine Benutzergruppe, die technische Unterstützung eines Produktionsablaufs, etc. Der damit verbundene Erkenntnisprozess ist zweckorientiert und daher begrenzt. Ein typisches Beispiel ist das gewissenhafte Erstellen einer detaillierten Spezifikation.

Evolutionäre Erkenntnis Hier ist die wissenschaftliche Herangehensweise gemeint. Laut Wilkins lassen sich die wissenschaftlichen Abläufe mit einem evolutionären Modell erfassen [Wilkins 95]. Es gibt keine endgültigen Ergebnisse, sondern nur Hypothesen. Das entscheidende ist die Verknüpfung von vielen ansonsten isolierten Erkenntnisprozessen. Die Abläufe auf der Meta-Ebene sind für eine schrittweise Verbesserung und das Entstehen einer konsistenten, strukturierten Gesamtheit der Erkenntnisse entscheidend. Typische Beispiele wären dafür die mathematisch fundierte Bereiche der theoretischen Informatik.

Die drei vorgestellten Handlungsstrategien stellen im Bereich der Informatik differenzierbare, tatsächlich beobachtbare Vorgehensweisen dar, die sich in Aufwand und Leistungsfähigkeit unterscheiden. Ihre umfassende Charakterisierung geht über den Fokus dieser Arbeit hinaus, aber es ist wesentlich die Idee eines Stufenmodells von Erkenntnismethoden festzuhalten, die je nach Anforderung verwendet werden können.

6.2 Kontextmodelle: Erfassung von Kontexten

6.2.1 Kontextmodelle für Komponenten

Entwickler müssen den Kontext kennen, in dem eine Komponente eingesetzt werden soll. Der Kontext umfasst die gesamte Umgebung einer Komponente und damit auch die Aufgabe, der sie dient. Da die angestrebte Nutzung aus der Perspektive des Entwicklers immer in der Zukunft liegt, wird dieser Kontext durch Modelle erfasst. Diese Modelle können explizit in Form von Spezifikationen, Beschreibungen, Diagrammen oder ähnlichem vorliegen oder nur implizit, in gegenwärtigen, beobachtbaren Vorgängen oder im Kopf der beteiligten Personen.

Da eine Softwarekomponente einem bestimmten Zweck dienen soll und sie während der Nutzung stets Teil eines Computersystems sein muss, ist dies auch ihr Kontext. Daher kann dieser relativ abstrakte Begriff hier basierend auf den Überlegungen aus Kapitel 2 für Komponenten präzisiert werden. Die dort dargestellten (Teil-)Systeme bilden zusammen ein einfaches, eigenes Metamodell für Kontextmodelle von Komponenten. Es legt fest, was im Folgenden unter einem Kontext-(Modell) verstanden wird. Es handelt sich dabei im Einzelnen um folgende Systeme: Nutzungssystem (US), technisches System (TS), Informationssystem (IS) und Entwicklungssystem (DS).

Dabei umfasst das Entwicklungssystem den spezifische Lebenszyklus einer Komponente und seine Umgebung, wenn man so will ihre Vergangenheit bzw. Zukunft. In den meisten Fällen dürfte dieses System bei mehreren Komponenten auf einem Systemknoten zumindest einige Gemeinsamkeiten haben, aber nicht identisch sein. Es beinhaltet z.B. die verwendete Infrastruktur, den Ablauf von Fehlerbehebungen, Beschaffungs- und Informationsquellen, Stationen des Verlaufs oder die beteiligten Akteure.

Durch Modelle von diesen vier Systemen lässt sich der wesentliche Kontext einer Komponente erfassen. Die Modelle bestehen aus Objekten und Prozessen. Da nur bestimmte Aspekte dieser Systeme für die Nutzung bzw. Entwicklung einer Komponente von Bedeutung sind, ist die Beschreibung dieser Systeme selbst bereits ein Modell. So spielt im TS die chemische Zusammensetzung der Hardware, der Lieferant, der sie gebracht hat oder der Name des Arbeiters, der sie eingepackt hat, keine Rolle. Äquivalentes gilt für die anderen drei Systeme. Für die Modelle dieser Systeme sind allgemein Strukturen, Regeln, Richtlinien und Standards interessant, die das System in einer prägnanten Form erfassen.

Konkrete Modelle dieser vier Systeme, die diesem Ansatz folgen, haben zunächst einen Fokus: die betrachtete Komponente. Es handelt sich um Modelle *von* den jeweiligen Systemen, die einen bestimmten Ausschnitt der Realität referenzieren, und *für* die Erfassung des relevanten Kontexts einer bestimmten Komponente. Der genaue Zweck für den der Kontext erfasst wird bestimmt, was die Modelle genau beinhalten und wie detailliert sie sind. Die durch eine Anforderungsanalyse erstellten Spezifikationen können ebenfalls als solche Kontextmodelle angesehen werden.

Entwickelt man Software, so muss man sich über diese vier Systeme Gedanken machen. Durch eine entsprechende Kapselung bei der Komponenten-Entwicklung kann man jedoch von vielen Details abstrahieren. So dient das Betriebssystem unter anderem der Aufgabe, für andere Komponenten den tatsächlichen Kontext zu vereinfachen und damit deren Entwicklung deutlich zu erleichtern, in dem diese sich weitestgehend nur mit den Schnittstellen zum Betriebssystem befassen müssen, die auf einem vereinfachten Modell des TS beruhen. Ein anderes Beispiel sind

Komponenten wie Bibliotheken, die ausschließlich interne Aufgaben für andere Elemente abarbeiten, z.B. ein Modul zur internen Speicherverwaltung. Es hat keine direkte Verbindung mit dem US, da es scheinbar nur Aufgaben im IS erfüllt. Trotzdem hat es einen Zweck im US, eventuell ist es nur das Verhindern von Speicherproblemen, die dann wiederum Auswirkungen auf das US hätten.

Diese Überlegungen verdeutlichen zusätzlich einen anderen wichtigen Umstand: Jede Komponente basiert zunächst einmal auf ihrem eigenen Kontextmodell, was nicht weiter verwunderlich ist, da Komponenten i.d.R. von unterschiedlichen Personen für unterschiedliche Zwecke entwickelt wurden. Der reale Kontext ändert sich aber nicht nur bei der Entwicklung, sondern auch bei der Nutzung und so würde man bei der Modellierung bei zwei unterschiedlichen Anwendungsfällen zu unterschiedlichen Ergebnissen kommen. Dasselbe gilt für mehrere beteiligte Personen: Unterschiedliche Vorstellungen über den Kontext (offensichtliche oder verdeckte) sind nicht zu vermeiden. Man kann diese Differenzen nur minimieren. Kontextmodelle sind immer nur eine Auffassung der Realität und können die Situation daher nur näherungsweise erfassen.

Das Ergebnis sind eine Vielzahl von Modellen für dieselbe oder auch unterschiedliche Komponenten. Jedoch haben sie erhebliche Überschneidungen, die diese Zusammenhänge extrem vereinfachen und in der realen Softwareentwicklung zu vereinheitlichten Kontextmodellen führen, aus denen Plattformen, Standards und ähnliches hervorgehen. Diese standardisierten Modelle können dann für ein breites Spektrum an Komponenten genutzt werden, sind aber nur ein Kompromiss zwischen Allgemeingültigkeit und Passgenauigkeit. Für bestimmte Komponenten sind Teile der relevanten Umgebung nicht beschrieben, für andere sind weite Teile des Modells ohne Bedeutung.

6.2.2 Kontextmodelle als Verträge mit der Umgebung

Kontextmodelle können als ein Vertrag zwischen dem betrachten Gegenstand und seiner Umgebung verstanden werden. Es ist allein eine Frage der Auffassung in Bezug auf die Verbindlichkeit. Wird ein Kontextmodell als verbindlich angesehen, dann wird damit gleichzeitig gefordert, dass alle enthaltenen Aussagen über den Kontext, den Gegenstand und ihr Verhältnis zueinander eingehalten werden. Dies stellt dann einerseits eine zu erfüllende Pflicht und anderseits ein einforderbares Recht dar.

Ein Modell des Kontexts einer Komponente erfasst die wesentlichen Elemente ihrer Umgebung und ihre Rolle darin. Dies umfasst insbesondere die Schnittstellen und das erwartete Verhalten und damit auch Anforderungen und Spezifikationen von Komponenten, die selbst als eine bestimmte Form von Kontextmodellen angesehen werden können.

An Komponenten und Systeme werden Anforderungen gestellt, die sie erfüllen sollen. Diese entstehen aus ihrer Umgebung, sie stellen sozusagen etwas wie ein Modell der Umgebung dar. Andererseits benötigen Systeme einen festen Rahmen, in den sie eingebettet sind, um korrekt funktionieren zu können. Dieser feste Rahmen stellt eine Beschreibung der Umgebung dar, die nach ihrer Formulierung, zu einer verbindlichen Festlegung wird, die dann seitens des Systems, soweit sie genutzt werden, als Anforderungen an die Umgebung wieder auftauchen. Beide Aspekte bilden zusammen eine Spezifikation der Umgebung, die so etwas wie ein Katalog von “Rechten und Pflichten” darstellen. Umgekehrt gilt das selbe: Die Spezifikation eines Systems

enthält seine “Rechte und Pflichten” und klärt damit das Verhältnis mit seiner Umgebung. Zusammen genommen handelt es sich dabei um einen Vertrag zwischen System und Umgebung im Sinne von [ISO10746-2 96, contract].

Ein System selbst stellt jedoch wiederum die Umgebung für die darin enthalten Elemente dar und das Verhältnis eines Systems zu seiner Umgebung hat eine gewisse Ähnlichkeit mit dem Verhältnis eines Elements zu diesem System. Aus der Sicht eines betrachteten Elements ist das gesamte übrige System und evtl. auch alles außerhalb davon seine Umgebung. Es wäre nun für die Entwicklung einer konkreten Komponente vorteilhaft, einen einheitlichen Vertrag mit ihrer Umgebung als Ganzes zu haben (restliches System IS/TS und Umgebung US), d.h. das Ideal wäre ein ausschließlicher Vertrag zwischen der Komponente und dem Gesamtsystem. Dies ist jedoch schwierig, da es für einen derartigen Vertrag keine tatsächliche strukturelle Einheit gibt, mit der dieser Vertrag geschlossen werden könnte. Das umliegende System besteht aus vielen anderen Elementen, die strukturell die selbe Situation widerspiegeln. Die enthaltenen Elemente stehen daher primär in einer Wechselbeziehung zueinander und nicht mit einem gesonderten Systemelement. Trotzdem lässt sich in einem System stets eine gewisse Ordnung erkennen, die sich in Form eines Rahmenvertrags charakterisieren lässt.

Unter dem Rahmenvertrag wird hier die Festlegung eines Systems bezüglich seiner inneren Ordnung und seiner Beziehung zur Umgebung verstanden. Dies umfasst primär zwei Bereich: Systemverträge und Metaverträge.

Verträge sind verbindliche Regeln zwischen zwei oder mehr Objekten. Man kann die Systemverträge entsprechend der beteiligten Objekte in folgende Kategorien einteilen. Verträge zwischen ...

1. allen betrachteten Systemen und ihrer Umgebung (z.B.: Ausgaben werden standardmäßig an einen Monitor geschickt; dem System wird vorher mitgeteilt, wenn es ausgeschaltet wird)
2. allen betrachteten Systemen und allen darin enthaltenen Elementen (z.B. Definition von Komponentenformaten, Aufbau des Dateisystems)
3. einem bestimmten System und seiner spezifischen Umgebung (z.B. Ausgaben finden vorzugsweise in englischer Sprache statt; erfolgt x Minuten keine Eingabe wird der Zugriff gesperrt)
4. einem bestimmten System und allen darin enthalten Elementen (z.B. installierte Elemente müssen konform zu Standard XYZ sein; die Ausgabe in englischer Sprache wird unterstützt)

Zudem können in einem solchen Rahmenvertrag auch die Grundsätze vorgegeben sein wie Elementverträge aussehen. Derartige Vorgaben werden hier als Metaverträge bezeichnet. Dies kann z.B. durch Templates, Metamodelle oder Musterverträge geschehen. Es sind Verträge eines Elements mit folgenden Objekten denkbar:

1. andere Elemente des selben Systems
2. Objekte der Systemumgebung
3. dem zugehörigen System

Die aus diesen Vorgaben abgeleiteten, konkreten Elementverträge legen dann fest wie genau das Verhältnis zwischen den beteiligten Objekten ist.

Systemverträge, Metaverträge und konkrete Elementverträge bilden dann zusammen ein ganzes Geflecht von Regeln, die alle Anforderungen umfassen, die an die beteiligten Elemente aus den drei Bereichen eines Systemknotens US, TS und IS gestellt werden. Dazu kommen noch die Regeln zwischen Komponente und seinem jeweiligen Entwicklungssystem.

Man muss dabei jedoch zwischen tatsächlichen Anforderungen/Regeln und den explizit formulierten unterscheiden. I.d.R. wird immer nur ein Teil der tatsächlichen Regeln schriftlich oder in einer anderen Form explizit festgehalten. Zudem kann es auch Abweichungen zwischen tatsächlichen und formulierten Anforderungen geben, wenn sich z.B. die jeweiligen Umstände zwischenzeitlich geändert haben oder es bereits bei der Formulierung zu Irrtümern kam.

Das vollständige Regelwerk zu einer bestimmten Komponente kann sich auf diese Weise aus vielen Teilen zusammensetzen, die jeder für sich einen Vertrag der Komponente mit einem bestimmten Teil der Umgebung darstellen. Da bei der jeweiligen Komponentenentwicklung alle diese Regeln gleichermaßen berücksichtigt werden müssen, ist es wichtig die Struktur dieses Regelwerks klar zu erkennen und evtl. Widersprüche, Redundanzen und andere Problembereiche zu identifizieren und möglichst aufzulösen.

In der Praxis werden vor allem zwei Arten von Kontextmodellen verwendet, die sich in ihrem Gültigkeitsbereich unterscheiden:

Komponenten-Kontextmodell beschreibt ausschließlich die Umgebung einer spezifischen Komponente und beinhalten primär Elementverträge.

System-Kontextmodell beschreibt die Umgebung aller Komponenten, die sich innerhalb eines Systems befinden. Ein derartiges Modell umfasst i.d.R. hauptsächlich System- und Metaverträge.

6.2.3 Umgang mit Kontextmodellen

Regeln, Anforderungen oder Erwartungen im Zusammenhang mit einer Komponente können unabhängig davon existieren, ob sie explizit formuliert wurden oder nicht. Zudem enthält die Komponente selbst implizit entsprechende Festlegungen, wobei dort im Ergebnis nicht mehr zwischen Konsequenzen aus notwendigen Vorgaben und frei gewählten Entscheidungen differenziert werden kann, wenn dies nicht explizit dokumentiert ist. Zudem ist ohne eine entsprechende Kennzeichnung auch nicht erkennbar, welche Teile im Verlauf einer Weiterentwicklung als (un)veränderlich betrachtet werden (sollen). Diese und weitere Meta-Informationen sind ohne zusätzliche Dokumentation dann nur noch von den Entwicklern selbst zu bekommen. Man kann demnach zwei verschiedene Darstellungsweisen unterscheiden, die zwei Extreme bilden zwischen denen es jedoch viele mögliche Kompromisse geben kann.

explizite Kontextmodelle Alle berücksichtigten, verbindlichen Vorgaben und Regeln wurden explizit in Form von Spezifikationen, Kommentaren, Dokumentationen, Verweisen auf Standards, etc. formuliert. Diese Dokumente können dann z.B. zum Verständnis oder zur Prüfung der Einhaltung benutzt werden.

implizite Kontextmodelle Verwendete Vorgaben und Regeln sind implizit in den erstellten Ergebnissen enthalten, aber nicht mehr als solche erkennbar. Diese Informationen stehen nur den Entwicklern als implizites Wissen zur Verfügung. Die Vernachlässigung der zusätzlichen Dokumente gibt den Entwicklern mehr Flexibilität und erspart ihnen die dafür erforderlichen Zusatzarbeiten.

Werden explizite Kontextmodelle für die Entwicklung von Komponenten verwendet, so stellt sich für die Entwickler und evtl. auch Außenstehende die Frage der Verbindlichkeit dieser Dokumente in doppelter Hinsicht:

Verhältnis zwischen Modell und Komponente Das Modell kann entweder eine verbindliche Spezifikation sein, die in jedem Fall eingehalten werden muss oder auch nur ein Entwurf, der als unverbindliche Vorlage für das Endprodukt dient, um damit einen ersten Gesamteindruck zu vermitteln.

Anpassung des Modells Findet die Entwicklung der Komponente in mehreren Phasen statt, so stellt sich die Frage, ob eine Überarbeitung oder Erweiterung des Modells in oder zwischen den Phasen erlaubt ist (dynamisch), oder ob das Kontextmodell während der gesamten Arbeiten unverändert bleibt (statisch).

Diese unterschiedlichen Umgangsformen mit Kontextmodellen haben jeweils ihre Vor- und Nachteile und führen zu verschiedenen Erfordernissen und Möglichkeiten seitens der Beteiligten. Welches nun die richtige Vorgehensweise für eine bestimmte Komponente ist hängt von den konkreten Bedürfnissen der Beteiligten und den vorliegenden Umständen ab.

6.3 Systementwicklung mit Kontextmodellen

6.3.1 Kontextmodelle und Prozessdomänen

Kontextmodelle sollten einen scharf begrenzten Gültigkeitsbereich haben, d.h. es sollte klar erkennbar sein, ob es für einen konkreten Gegenstand in einem bestimmten Kontext (Entwicklungsphase, Zeitpunkt, Softwarepool, etc.) relevant ist oder nicht. Zudem sollten alle Kontextmodelle, die in diesem Kontext für ihn gelten zumindest widerspruchsfrei sein und möglichst wenig redundante Aussagen enthalten, d.h. sie müssen sich zu einer konsistenten Einheit zusammenfügen lassen. In diesem Fall kann man von einem gesamtheitlichen Kontextmodell für ein Objekt sprechen, welches aus einigen Teilmodellen besteht. Solche Teilmodelle sind dann teilweise auch für mehrere unterschiedliche Objekte gültig. So können sich Objekte z.B. ein System-Kontextmodell teilen (vgl. Abs. 6.2.2) oder Modelle für bestimmte Objekttypen vorhanden sein. Auf diese Weise bildet sich ein ganzes Geflecht von Teilmodellen, die zusammen für eine ganze Menge von Objekten ein großes Kontextmodell bilden, das wiederum in sich konsistent sein sollte. Um Probleme und Inkonsistenzen zu vermeiden, sollte der gemeinsame Geltungsbereich all dieser (Teil-)Kontextmodelle klar begrenzt sein. Eine Möglichkeit dafür ist das Konzept der Prozessdomäne, wie es in Abschnitt 4.2.3 geschildert wurde. In diesem Fall gilt ein Modell entweder für die gesamte Prozessdomäne oder gar nicht und alle Kontextmodelle innerhalb dieser Domäne müssen zueinander kompatibel und damit insgesamt wieder konsistent sein.

Werden unterschiedliche Kontextmodelle im Verlauf der Entwicklung verwendet, die sich z.B. im Detaillierungsgrad oder der Perspektive unterscheiden, sollte diese ebenfalls mit einer klaren Trennung der Prozessdomänen verbunden sein, d.h. innerhalb einer Domäne darf stets nur eines dieser Modelle verwendet werden und der Wechsel von einem zum anderen sollte auch mit einem Domänenwechsel verbunden sein, um die Konsistenz sicherzustellen. Dies bedeutet jedoch nicht im Umkehrschluss, dass jede Prozessdomäne ihr eigenes Kontextmodell besitzen

muss. Es ist durchaus möglich, dass eine weitere Trennung aus anderen Gründen stattfindet (z.B. Änderung der Zuständigkeit) und mehrere Domänen das gleiche Kontextmodell verwenden. Ein Kontextmodell ist ein *Teil* des Kontextes eines Entwicklungsprozesses, aber z.B. Werkzeuge gehören ebenso dazu. Man sollte daher nicht den Kontext (des Entwicklungsprozesses) und das verwendete Kontextmodell (des Entwicklungsgegenstands) miteinander verwechseln.

Der Anteil eines gesamtheitlichen Kontextmodells, der nur im impliziten Wissen der entsprechenden Akteure vorhanden ist, kann jedoch nur durch ihr Mitwirken in der entsprechenden Prozessdomäne genutzt werden. Dadurch ist es in vielen Fällen besser, Prozessdomäne und Kontextmodell als eine Einheit zu sehen und den Versuch aufzugeben, ein konsistentes Kontextmodell über Domänen-Grenzen hinweg zu nutzen. Ansonsten muss man sich mit expliziten Modellen begnügen.

Befinden sich nun mehrere Objekte innerhalb einer solchen Prozessdomäne, dann können sie selbst Kontext für andere Objekte sein, d.h. es gibt Relationen zwischen ihnen wie es in Abschnitt 2.2.4 geschildert wurde. In der Softwareentwicklung spielen diese Relationen eine besondere Rolle, da Komponenten immer Teil eines größeren Ganzen sind und oftmals nur im Zusammenspiel mit anderen Objekten ihren Zweck erfüllen. Diese Relationen müssen nicht auf die lokale Prozessdomäne beschränkt sein, sondern können sich eben auch wie die verwendeten Kontextmodelle auf zukünftige Kontexte beziehen. Dadurch verschwimmt die Grenze zwischen Gegenständen und Kontextmodellen und es kann zu einer Inkonsistenz mit anderen Elementen führen, wenn in einer Prozessdomäne ein Objekt bzw. eine Komponente verändert, entfernt, oder hinzugefügt wird. Der Import- und Export-Prozess dient u.a. dafür, solche Relationen für ein hinzugefügtes Element in den lokalen Kontext zu integrieren bzw. bei der Ausgabe aus ihm zu extrahieren. Sind diese Relationen der Infrastruktur einer Prozessdomäne bekannt, können sie entsprechend verwaltet, überwacht und teilweise auch automatisch angepasst werden.

Bei dem Transfer eines Objekts von einem Kontextmodell ins nächste können diese Relationen dann teilweise einfach übertragen werden. Andere fallen jedoch weg bzw. kommen dazu wie es Abb. 6.1 zeigt.

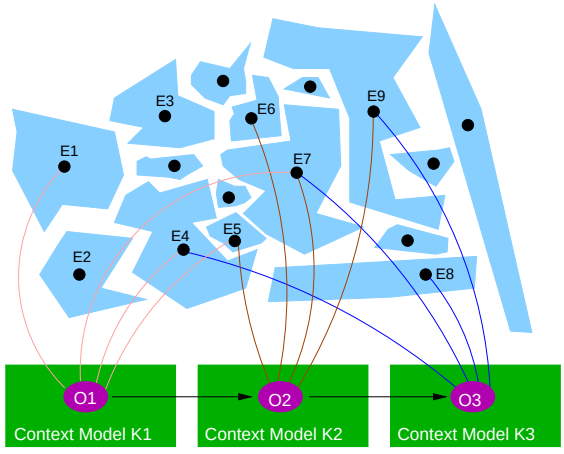

Abbildung 6.1: Kontextwechsel eines Objekts und die Übertragung seiner Relationen

Das Objekt O1 wird aus dem Kontextmodell K1 in K2 transferiert und wird dort zu O2, welches in K3 zu O3 wird. Vereinfacht wird angenommen, dass sich die Elemente E1 bis E9 dabei nicht

verändern. Die farbigen Bögen stellen Relationen des Objekts Ox mit den Elementen Ex in dem jeweiligen Kontextmodell Kx dar. Bei dem Transfer verändern sich diese bis auf die Relation zu E7, die erhalten bleibt. Die Relationen zu E1, E6 und E8 spielen nur in je einem Modell eine Rolle. E5 bleibt beim Wechsel von K1 zu K2 erhalten, E9 zwischen K2 und K3. E4 verschwindet in K2 taucht aber dann in K3 wieder auf.

Um so größer die Menge der beteiligten Objekte, Kontextmodelle und der relevanten Relationen ist, desto schwieriger wird es, die komplexen Zusammenhänge zu überblicken, zu erfassen und die Konsistenz sicherzustellen. Dies ist jedoch für ein korrektes Funktionieren der Softwarekomponenten notwendig. Klar definierte Prozessdomänen, in denen die zugehörigen Kontextmodelle konsistent gehalten werden und zumindest die Verwaltung der Relationen durch die Infrastruktur unterstützt wird, können dazu einen wesentlichen Beitrag leisten. Dies gilt auch für die impliziten Anteile der Modelle, die jedoch nur durch eine entsprechend langfristige Teilnahme der Akteure erreicht werden kann, die dieses implizite Wissen besitzen, verstehen und nutzen.

6.3.2 Kontextproblem von generischen Komponenten

Komponentensysteme wie sie in Kapitel 3 beschrieben wurden und die zugehörigen Softwarepools basieren auf der Idee, ein Softwareelement S in verschiedenen Systemknoten und damit in unterschiedlichen Kontexten zu verwenden. Damit verlieren die Softwareelemente jedoch ihren konkreten Kontext, an dem sich die Entwickler orientieren können und muss durch einen generischen, komponentenspezifischen Kontext Ks ersetzt werden, der die relevante Umgebung aus allen beteiligten Systemknoten enthält (vgl. Abb. 6.2).

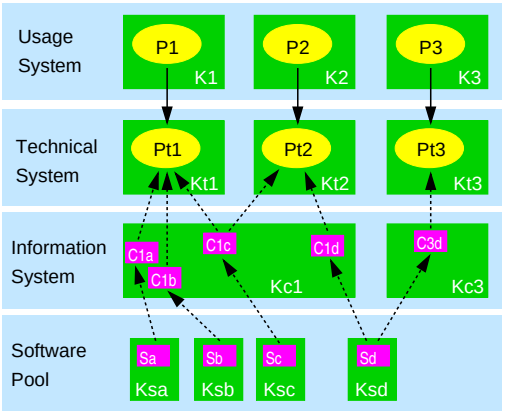

Abbildung 6.2: Systeme mit mehrfach verwendeten Komponenten

Das dem zugrundeliegende Prinzip der Wiederverwendung basiert auf der Nutzung eines Softwareelements in einer Umgebung, für die es ursprünglich nicht vorgesehen war. Auch wenn von Anfang an ein generischer Baustein für viele unterschiedliche Softwaresysteme angestrebt wird, haben die Entwickler trotzdem keine konkrete Umgebung, an der sie sich bei der Erstellung orientieren können. Sie sind dann gezwungen, sich an einem exemplarischen Kontext zu orientieren, der jedoch immer nur eine Näherung sein kann. In beiden Fällen können bei der Verwendung diese Unzulänglichkeiten bis zu einem gewissen Grad durch entsprechende Anpassung oder Kompensation ausgeglichen werden, um die Diskrepanz zwischen Annahmen der

Entwickler und tatsächlicher Umgebung auszugleichen. Es bleibt jedoch nahezu unmöglich eine generische Komponente so zu bauen, dass sie sich optimal in tausende unterschiedliche Umgebungen einfügt. Trotzdem kann man versuchen, einen möglichst guten Kompromiss zwischen den Anforderungen all dieser Kontexte zu finden. In Anbetracht der Schwierigkeiten, die bereits die Anforderungsanalyse einer einzigen konkreten Umgebung macht, wird jedoch klar wie komplex ein solcher Vorgang ist.

Komponentenbasierte Systeme setzen sich zudem aus hunderten Komponenten zusammen, die eine passende Auswahl aus einem Pool von Tausenden darstellen. Durch diesen Vorgang der Systementwicklung entstehen aus einem Baukasten von Tausenden von Komponenten Millionen von unterschiedlichen Softwaresystemen für verschiedenste Aufgaben.

Da einzelne Komponenten auf diese Art und Weise auf Millionen von unterschiedlichen Systemen eingesetzt werden, ist es relativ schwer, ein allgemeingültiges Modell von ihrem Kontext zu erstellen. Andererseits ist es auch nicht möglich, alle unterschiedlichen Umgebungen einzeln zu berücksichtigen. Zur (Weiter-)Entwicklung einer Komponente müsste man eigentlich ganz genau wissen, wie alle möglichen Nutzungsweisen aussehen.

Dies ist die Ursache für das allgemeine Dilemma von Komponenten: einerseits spart man mit ihnen Unmengen redundanten Code, anderseits wird dadurch die Entfremdung von ihrem Kontext immer größer, was dazu führt, dass der Entwickler immer weniger versteht, wie die Nutzer seine Software einsetzen. Meistens ist bei der Erstellung nicht einmal klar, welche Nutzer das Element in welchem Kontext später verwenden. Die üblichen Probleme der Softwareentwicklung kommen noch dazu und potenzieren sich durch die Vielzahl der Kontexte: Der Nutzungszeitraum ist in der Zukunft. Der Kontext ist normalerweise in einem stetigen Wandel begriffen. Kontextbeschreibungen, Software und Hardware sind alle fehlerbehaftet, was gelegentliche Korrekturen notwendig macht.

Selbst der *relevante* Kontext einer Komponente kann folglich nur näherungsweise erfasst werden und ändert sich auch noch ständig. Selbst wenn man es also schafft, „fehlerfreie" Software zu schreiben, ist es trotzdem nur eine Frage der Zeit, bis die ersten Korrekturen notwendig werden. Diese Problematik ist die treibende Kraft hinter einem Großteil der heutigen Softwareentwicklung.

Jedes Komponentensystem muss eine Lösung für diese Problematik finden.

6.3.3 Vereinheitlichung von Kontextmodellen

Das in Abschnitt 6.3.2 geschilderte Kontextproblem von generischen Komponenten hat die Ursache in der großen Anzahl an zu berücksichtigenden Varianten in den unterschiedlichen Kontextbereichen (Nutzungssystem US, technisches System TS, Informationssystem IS und Entwicklungssystem DS).

Ein Lösungsansatz dafür ist die Vereinheitlichung des zugehörigen Kontexts durch eine entsprechende Abstraktion von Details und Varianten, die als unwesentlich angesehen werden. Dies ist in Abb. 6.3 exemplarisch für das US dargestellt.

Aus den drei unterschiedlichen Prozessen P1, P2 und P3 mit den zugehörigen Kontexten K1, K2 und K3 wird durch Vereinheitlichung ein generischer Prozess Pg mit seinem generischen Kontext Kg erstellt, der als ein gültiges Modell für alle drei Prozesse akzeptiert wird. Wenn

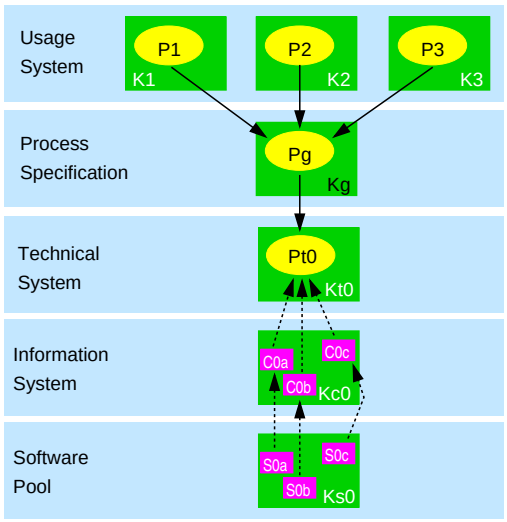

Abbildung 6.3: Verallgemeinerung des technischen Systems durch Einfügen eines generischen Prozessmodells Pg mit seinem generischen Kontextmodell Kg

man ein identisches TS für alle drei Prozesse annimmt, kann man sich so auf die Entwicklung eines einzigen technischen Prozesses Pt0 beschränken, statt für jeden Prozess P einen eigenen Pt zu erarbeiten (wie in Abb. 6.2 dargestellt). Als Konsequenz braucht man dann auch nur ein passendes IS erstellen und auch die entsprechenden Arbeiten am Softwarepool werden dadurch vereinfacht.

Analoge Vorteile hat eine Standardisierung des TS und des IS, d.h. es wäre auch hier wünschenswert, die Vielfalt der möglichen Varianten auf ein Minimum zu reduzieren.

Eine Vereinheitlichung bringt in Bezug auf die Entwicklungsarbeit und den dafür notwendigen Aufwand also deutliche Vorteile. Jedoch sind damit auch zwei wesentliche Nachteile verbunden:

Vereinheitlichungsaufwand Die Standardisierung der Kontexte selbst ist mit Aufwand verbunden, der je nach Situation und Vorgehen extrem hoch sein kann.

Reduzierte Passgenauigkeit Ein generischer Kontext ist immer ein Kompromiss zwischen den unterschiedlichen Varianten und erfordert damit eine entsprechende Kompensation von der betreffenden Umgebung.

Ob die Vor- oder Nachteile überwiegen ist vom Einzelfall abhängig und kann nicht allgemein beantwortet werden. Dabei spielen z.B. folgende Faktoren eine Rolle: die Ähnlichkeit der relevanten Kontexte, der Verhandlungsspielraum bei extremen Abweichungen, die Standardisierungsmethode oder die Anpassungsfähigkeit des jeweiligen Kontextes.

6.3.4 Verkettung von Kontextmodellen

Es ist ein mögliches Vorgehen bei der Softwareentwicklung, mehrere unterschiedliche Kontextmodelle zu verwenden und im Laufe der Arbeiten die Ergebnisse von einem Modell ins andere zu übertragen (vgl. Abb. 6.4).

Dabei findet die Entwicklungsarbeit dann in zwei wesentlichen Bereichen statt:

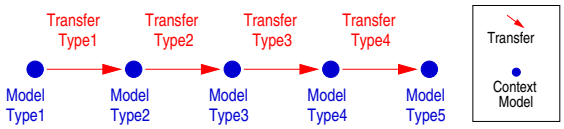

Abbildung 6.4: Softwareentwicklung mit unterschiedlichen Kontextmodellen in jeder Phase

1. Für die jeweilige Transformation (transfer) muss der relevante Kontext erfasst werden. Das Resultat dieser Arbeit ist das entsprechende Kontextmodell (context model).
2. Der Entwicklungsgegenstands S wird dann durch die Transformation von dem vorhergehenden Kontextmodell in das neue übertragen.

Auch der Ursprung der ersten Transformation wird hier als Modell angesehen, da entsprechend den Ausführungen in Abschnitt 6.1 davon ausgegangen wird, dass die bereitgestellten Informationen ebenfalls nur ein Modell der Realität sind und somit nur eine eingeschränkte Sicht auf sie darstellen. Da es hier um die Entwicklung von Software geht, die erst nach ihrer Fertigstellung genutzt werden kann, handelt es sich in jedem Fall bei dem betreffenden Realitätsausschnitt um eine Situation in der Zukunft und kann schon allein deswegen nur modellhaft erfasst werden. Selbst die gegenwärtige Situation des zukünftigen Einsatzortes ist dann nur ein Modell der zukünftigen Situation.

Dabei stellen die Kontextmodelle jeweils eine Sicht auf den relevanten Kontext von S dar, der in der betreffenden Entwicklungsphase relevant ist. S beschreibt dabei jeweils die Vorgänge in dem zugehörigen Systemknoten in Bezug auf das jeweilige Kontextmodell. Das Endergebnis dieser Transformationskette, die angepasste Softwarekomponente, wird dann in dem betreffenden Informationssystem IS abgelegt.

Oftmals werden dabei die jeweils benutzten Kontextmodelle nicht explizit dargestellt oder auch nur charakterisiert. Stattdessen werden die entsprechenden Aspekte des Entwicklungsgegenstands benannt und präzisiert, die dann indirekt eine bestimmte Sichtweise auf den jeweiligen Kontext implizieren. Dies ist ein Grund dafür, warum oftmals die Grenze zwischen dem Kontextmodell und dem Entwicklungsgegenstand verschwimmt. Man kann letztendlich sogar die Software selbst als ein sehr begrenztes Modell des relevanten Kontexts auffassen, das von einem TS interpretiert werden kann.

Betrachtet ein Benutzer das resultierende Verhalten des TS beim realen Einsatz dann als mangelhaft, kann die Ursache dafür entweder in der Software oder der realen Einsatzumgebung zu finden sein. Ist die Software S dafür verantwortlich, besteht ein Widerspruch zwischen den Vorstellungen des Benutzers und dem in S codierten Verhalten. Die Ursache dafür kann nun in unpassenden Kontextmodellen oder einer entsprechend fehlerhaften Transformationen des Entwicklungsgegenstands liegen. Evtl. wurde einfach nur der betreffende Aspekt in keinem der Kontextmodelle und damit auch nicht in S erfasst.

6.3.5 Differenzierung von Kontextmodellen

In Abschnitt 6.3.4 wurde die einfach Verkettung von Kontextmodellen dargestellt, d.h. der Entwicklungsgegenstand wurde stets nur in ein neues Kontextmodell transformiert. Es ist jedoch auch möglich den entsprechenden Gegenstand in mehrere verschiedene Kontextmodelle

zu übertragen, die jeweils unterschiedliche Kontexte repräsentieren. Auf diese Weise können unterschiedliche Varianten eines möglichen Kontexts in verschiedenen Entwicklungszweigen behandelt werden. Die unabhängige Behandlung stellt einen Kompromiss zwischen einer unabhängigen Entwicklung für einzelne Kontexte und der in Abschnitt 6.3.3 dargestellten Vereinheitlichung dar. Auf diese Weise können Details, die sich nicht oder nur schwer miteinander verbinden lassen bzw. ein zu komplexes Kontextmodell nach sich ziehen, unabhängig von einander behandelt werden, ohne die Entwicklung vollständig getrennt durchzuführen.

So kann bei einer Verkettung von Kontextmodellen, die stetig im Detaillierungsgrad zunehmen, zunächst im Rahmen eines gemeinsamen, abstrakten Kontextmodells Arbeiten geleistet werden, die für alle Kontexte zutreffen, und spezifische Unterschiede in einem nachgelagerten Arbeitsschritt mit unterschiedlichen Kontextmodellen behandelt werden.

Auf diese Weise entsteht eine Baumstruktur von Arbeitsbereichen, die den Entwicklungsgegenstand an immer spezifischere Kontextmodelle anpassen, wie es Abbildung 6.5 illustriert.

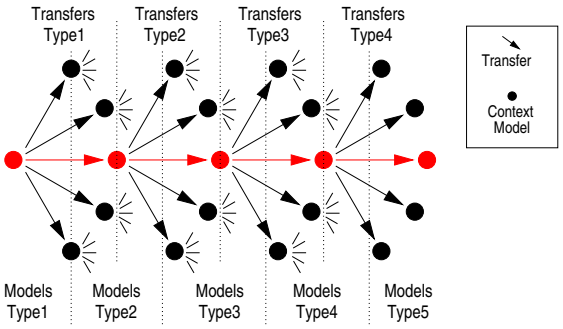

Abbildung 6.5: Baumstruktur von Kontextmodellen

Die rote Kette von Kontextmodellen und ihren Transformationen stellt hier den verfolgten Hauptstrang dar. Die weiteren Verzweigungen von den schwarzen Kontextmodellen wurden durch die kurzen Striche angedeutet. Die Kontextmodelle vom Typ 1 sind hier die allgemeinsten und die von Typ 5 die spezifischsten. Es handelt sich hier um eine schematische Darstellung, in der von jedem Kontextmodell gleich viele Transformationen zu entsprechenden, spezifischeren Kontextmodellen abgehen. Dies muss nicht so sein. Die Anzahl der Verzweigungen ist beliebig.

Der Vorteile einer solchen Zerlegung der Entwicklungsarbeit ist die Reduzierung der Komplexität und des damit verbunden Aufwands für die einzelnen Arbeitsbereiche bei der Erstellung der jeweiligen Kontextmodelle und der Entwicklung der zugehörigen Software. Die Ursache dafür liegt in der kleineren Anzahl der zu berücksichtigenden Kontexte, der Möglichkeit starke Abweichungen unabhängig behandeln zu können und der daraus resultierenden Einfachheit der erstellten Kontextmodelle. Zudem kann auf diese Weise der aufwändige Wissenstransfer wesentlich vermindert werden, da die dezentrale Entwicklungsarbeit es erlaubt, entsprechende Arbeiten kontextnahe durchzuführen.

Die Trennung von Komponenten und Systementwicklung basiert auf diesem Prinzip. Sie wird sowohl in der komponentenbasierten Entwicklung verwendet (vgl. Kapitel 3) wie auch in dem darauf basierenden Ansatz der Software-Produktlinien (vgl. [Clements 07]).

Auch die Model Driven Architecture verwendet diesen Ansatz, indem die Entwicklungsarbeit ebenfalls in mehreren Schritten mit unterschiedlichen Kontextmodellen (CIM, PIM, PSM, etc.)

durchgeführt wird (vgl. [Kleppe 03]).

6.4 Modell der Open-Source-Entwicklung

6.4.1 Prozessbezogene Kontextmodelle

Wie in Abschnitt 3.5 gezeigt, handelt es sich bei Open-Source-Distributionen um Komponentensysteme, die allgemein das in Abschnitt 6.3.2 geschilderte Kontextproblem aufweisen.

Open-Source-Entwicklung findet jedoch während der Wartungsphase des klassischen Lebenszyklus statt. Für diese ausgedehnte Wartungsphase hat sich im Open-Source-Bereich ein eigener Lebenszyklus herausgebildet (vgl. Abs. 4.5.2). Er besteht aus Phasen, die jeweils ein eigenes Kontextmodell haben. Dadurch wird die Erfassung und Berücksichtigung des Kontexts auf viele Prozesse verteilt und eine schrittweise Spezialisierung ermöglicht. Zudem erlaubt dies eine kontextnahe Entwicklung, durch die notwendige Spekulationen auf ein Minimum reduziert werden. Zusätzlich wird durch intensive Rückkopplung (z.B. Änderungswünsche) der relevante Kontext auch in anderen Prozesse übertragen. Der beschriebene Lebenszyklus von Open-Source-Komponenten stellt daher einen Entwicklungsprozess für komponentenbasierte Systeme dar, der das Kontextproblem deutlich reduziert.

Bei näherer Betrachtung der Komponenten, ihrer Kontextmodelle und deren Lebensphasen im Open-Source-Bereich fällt auf, dass sich die Modelle unterschiedlicher Komponenten in den jeweiligen Phasen ähnlich sind, sich von denen anderer Phasen abgrenzen lassen und damit Klassen von Kontextmodellen bilden. So orientieren sich aCPs an einem betrachteten Anwendungsfall (use case), uCPs an einem allgemeingültigen Anwendungsszenario (use scenario), pCPs an einer zugehörigen Systemvorlage (system template), iCPs an dem betreffenden Systemknoten (system node) und cCPs an einem spezifischen Nutzungsprozess (use process).

Die Kontextmodelle sind nicht isoliert zu betrachten, sondern stehen zueinander in Beziehung. So ist das Anwendungsszenario eine Generalisierung aller berücksichtigten Anwendungsfälle, die Systemvorlage eine Generalisierung aller berücksichtigten Systemknoten und dieser eine Generalisierung aller berücksichtigten Nutzungsprozesse.

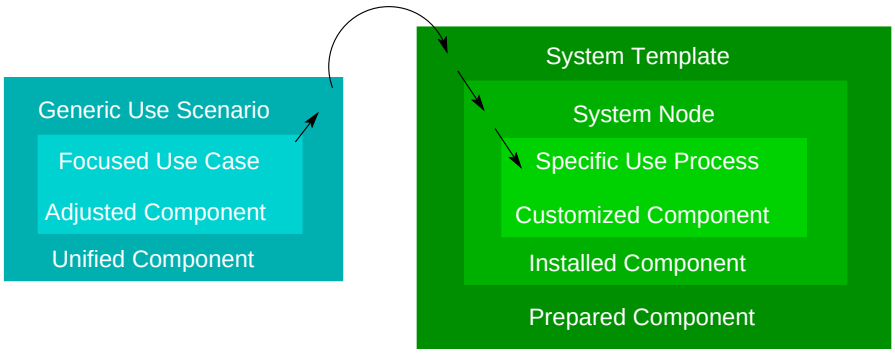

Abbildung 6.6: Weg der Komponenten durch die unterschiedlichen Kontextmodelle

Den ersten Schritt vom Anwendungsfall zum Anwendungsszenario können nur angepasste Komponenten bewältigen, die dem allgemeineren Anwendungsszenario nicht widersprechen.

Auf der anderen Seite findet vom Anwendungsszenario bis zum Nutzungsprozess eine schrittweise Anpassung an den Kontext statt. Die eigentliche Funktionalität wird nicht mehr verändert, sondern nur an zunehmend spezialisierte Kontextmodelle angepasst (vgl. Abb. 6.6), d.h. ihr Abstraktionsgrad nimmt zwischen Anwendungsfall und Anwendungsszenario zunächst zu, erreicht dort seinen höchsten Punkt und nimmt dann bis hin zum Nutzungsprozess wieder stetig ab.

6.4.1.1 Nutzung

Der Ausgangspunkt ist der Nutzungsvorgang selbst, aus dem Änderungsvorschläge hervorgehen, die dann den Lebenszyklus erst anregen. Wird eine Komponente eingesetzt, so befindet sie sich bereits in ihrer konkreten Umgebung, die entsprechenden drei Systeme sind daher der konkrete Kontext und kein Modell. Formuliert man jedoch einen Änderungsvorschlag, muss man Teile dieses Kontexts darstellen, d.h. ein Modell davon erstellen.

US konkrete Domänen, Nutzer und Nutzungsprozess

TS konkretes Hardwaresystem

IS konkretes Informationssystem

DS Modell des Ablaufs einer Wartung/Weiterentwicklung

6.4.1.2 Anwendungsfall

Das Kontextmodell ist ein Anwendungsfall und wird normalerweise in Form eines Änderungswunsches mitgeteilt. Für den beschriebenen Fall ist eine Komponente zwar grundsätzlich geeignet, aber sie könnte besser funktionieren, d.h. an die betrachtete Situation angepasst werden. Bei der Erstellung einer entsprechenden Änderung orientiert sich der Entwickler an diesem beschriebenen Anwendungsfall. Seine exakte Erfassung, Beschreibung und Mitteilung ist wie jede Übermittlung eines Kontexts schwer, größtenteils implizit und stets nur eine Näherung. Dieses Kommunikationsproblem wird kleiner, um so geringer der Unterschied zwischen Nutzungs- und Entwicklungskontext ist. Dies kann erreicht werden, in dem Nutzer und Entwickler die selbe Person sind und für die Entwicklung das selbe System in der selben realen Umgebung verwendet wird. Der Kommunikationsaufwand zur Mitteilung des Kontexts steht dabei oft in keinem Verhältnis zum Änderungsaufwand. Es stehen sich zwei Dinge gegenüber: Das Wissen des Nutzers über den Anwendungsfall und das Wissen des Urhebers über die Komponente. Je nach Umfang der Änderung und Aufwand einer Kontextbeschreibung ist daher eine dezentrale Entwicklung vor Ort für beide Seiten oft wesentlich effizienter. Es handelt sich dabei jedoch um eine Abwägungssache mit vielen Unbekannten.

US fokussierter Anwendungsfall

TS prototypisches Hardwaresystem

IS prototypisches Informationssystem

DS Modell des kommenden Entwicklungszyklus

6.4.1.3 Anwendungsszenario

Im Vereinheitlichungsprozess muss vor allem entschieden werden, welche Änderungen angenommen werden. Um diese Entscheidung ordnungsgemäß treffen zu können, müsste man alle Kontexte genau kennen, in denen die Komponente eingesetzt wird. Da dies nicht möglich ist, kann der Entscheider nur seine bisherige Erfahrung und das daraus resultierende implizite, abstrakte Kontextmodell verwenden. Diese Entscheidungen erfordern wohl im gesamten Lebenszyklus die größte Übersicht und das umfassendste Wissen. Zudem muss hier dafür gesorgt werden, dass die Software konsistent, strukturiert, verständlich, wartbar und nutzbar bleibt. Daher ist es nicht verwunderlich, dass diese Aufgabe den erfahrensten Entwicklern überlassen wird. Da sie die größte Entscheidungskompetenz haben, werden sie oft auch als Projektleiter (project leader) bezeichnet.

US Modell aller berücksichtigten Verwendungsmöglichkeiten

TS Modell aller möglichen Hardwaresysteme

IS Modell aller möglichen Informationssysteme

DS Modell aller möglichen kommenden Entwicklungsschritte

6.4.1.4 Systemvorlage

Distributoren erstellen mit Hilfe einer Vielzahl von Komponenten eine Vorlage, aus der dann ein konkretes System gebaut werden kann. Aus ihrer Zielgruppe und den Zielsystemen leiten sie dabei ein Kontextmodell ab, das ihnen ermöglicht, viele Anpassungen und Vorbereitungen bereits zu erledigen, die ansonsten im Installationsprozess nachgeholt werden müssen. Um so spezieller die Nutzergruppe und das System ist, um so mehr kann in diesem Stadium schon erledigt werden. So wird eine Distribution für den Standardnutzer in der japanischen öffentlichen Verwaltung für genau einen Computertyp ganz anders aussehen als eine für Handys im deutschen Markt.

US Modell aller passenden Domänen, Benutzer und Prozesse

TS Modell aller passenden Hardwaresysteme

IS Modell aller generierbaren Informationssysteme für diese Vorlage

DS Modell aller vorgesehenen Entwicklungsschritte der Vorlage

6.4.1.5 Systemknoten

Bei der Installation auf einem konkreten Hardwaresystem sind bereits viele Dinge festgelegt. So ist klar welche Hardware verwendet wird, welche anderen Softwarekomponenten sich auf dem System befinden und auch das Nutzungssystem ist bereits durch den Standort und die Zugriffsmöglichkeiten weitgehend eingeschränkt.

US konkrete Domäne, Modell für betreute Nutzer und Prozesse

TS konkretes Hardwaresystem

IS konkretes Informationssystem ohne Nutzerbereiche

DS Modell aller verbleibenden Entwicklungsschritte

6.4.1.6 Nutzungsprozess

Bei der Nutzeranpassung werden die letzten Teile des Kontextmodells durch entsprechende Angaben zum Benutzer und seine Vorlieben konkretisiert und die Komponente auf eine konkrete Nutzung durch den Nutzer vorbereitet.

US konkrete Domäne/Nutzer und Modell des vorgesehenen Prozesses

TS konkretes Hardwaresystem

IS konkretes Informationssystem

DS Modell der Meta-Vorgänge um den Nutzungsprozess

Man kann den Open-Source-Lebenszyklus folglich in fünf Bereiche gliedern, die jeweils eine Klasse von Prozessen und Kontextmodellen umfasst. Es handelt sich dabei um die Anwendung des Prinzips der Verkettung von Kontextmodellen wie es in Abschnitt 6.3.4 vorgestellt wurde. Diese Zerlegung funktionieren wie eine Art Puffer zwischen Komponenten- und Systembereich. Einerseits können viele Anpassungen lokal begrenzt vorgenommen werden, andererseits durch eine entsprechende Einordnung auch redundante Arbeiten vermieden werden.

Eine wesentliche Aufgabe der beschriebenen Prozesse ist dabei die Transformation der Komponenten von einem Kontextmodell ins andere.

6.4.2 Verwendete Erkenntnismethoden

Wie in Abschnitt 6.1.1 geschildert wurde, benötigt man in der Softwareentwicklung umfangreiches Wissen, um ein entsprechendes Vorhaben zufriedenstellend für alle Beteiligten durchzuführen. Dieses Wissen ist jedoch i.d.R. am Beginn der Arbeiten nicht vorhanden und benötigt in den meisten Fällen zusätzlich eine permanenten Aktualisierung. Dafür ist eine umfangreiche Erkenntnisarbeit notwendig. In Abschnitt 6.1 wurden dafür drei unterschiedliche Erkenntnismethoden vorgestellt, die im Open-Source-Bereich miteinander kombiniert werden.

“Ein System das verwendet wird, wird verändert werden” lautet Lehmanns erstes Gesetz [Endres 03, S. 163]. Brooks schreibt ebenfalls “Die einzige Konstante ist die Veränderung selbst” [Brooks 95, S. 117]. Letztendlich muss man daher in der Softwareentwicklung auf lange Sicht davon ausgehen, dass gefundene Erkenntnisse bezüglich des relevanten Kontexts revidiert werden müssen. Dies kann man nur verhindern, wenn man die betreffenden Bereiche derart unter Kontrolle hat, dass man die natürliche Veränderung verhindern kann. Dies ist z.B. bei dem eigenen Computersystem in begrenzter Form der Fall.

Es stellt sich daher die Frage wie man dieser konstanten Veränderung begegnet. Bezüglich der Erkenntnisformen bedeutet dies, den Anspruch aufzugeben, einen Kontext endgültig erfasst zu haben. Konsequenter Weise muss man dann entsprechende Kontextmodelle und die zugehörige Software stets als provisorisch betrachten.

Der Open-Source-Lebenszyklus trägt diesem Umstand Rechnung, in dem Entwicklung und Wartung miteinander verschmolzen wurden (vgl. Abschnitt 4.5.1) und die eigentliche Entwicklung auf diese Weise während der Nutzung nie wirklich endet. Eine Open-Source-Komponente wird sozusagen so lange weiterentwickelt wie sie genutzt wird. Da dies auch für das zugrundeliegende Kontextmodell gilt, stellt dies eine evolutionäre Erkenntnismethode für den Bereich der globalen, zentralen Komponentenentwicklung dar.

Betrachtet man nun jedoch die Herangehensweise eines Administrators, der für seinen lokalen Systemknoten eine Komponente modifiziert und sie an die spezifischen Bedürfnisse anpasst, so kann man die dafür verwendete Erkenntnismethode i.d.R. als direkt einstufen.

Will ein mittelständisches Unternehmen seine kompletten EDV Arbeitsplätze auf GNU/Linux umstellen und dafür einen entsprechenden Dienstleister beauftragen, so muss dieser einerseits eine möglichst präzise Vorstellung von den realen Umständen und Bedürfnissen der jeweiligen Arbeitsplätze erhalten und anderseits einen konkreten wirtschaftlichen Rahmen für die zu erbringenden Leistungen ausarbeiten. Für ein solches Szenario ist zumindest zum Teil eine analytische Erkenntnismethode notwendig, selbst wenn man die Geschäftsverbindung anschließend in einen dauerhaften Wartungsvertrag aufrecht erhält.

Allgemein kann man festhalten, dass für ein konkretes Problem mit einem klaren zeitlichen, wirtschaftlichen und inhaltlichen Rahmen entweder eine direkte oder eine analytische Erkenntnismethode geeignet ist. Eine evolutionäre Herangehensweise ist dafür nicht geeignet, da diese niemals zu einem abschließenden Resultat kommen (kann), sondern das Erreichte als hypothetisches Zwischen-Ergebnisse unendlich in Frage stellt. Dies ist für den Alltag und die Lösung von konkreten Problemen nicht praktikabel.

Anders sieht es bei der Entwicklung von allgemeinen Komponenten mit dem in Abschnitt 6.3.2 beschriebenen Kontextproblem aus. Dabei handelt es sich nicht um ein *konkretes* Problem, sondern um ein *hypothetisches*, welches erst durch die Verwendung in einem realen Kontext einen konkreten, überprüfbaren Rahmen erhält, in dem es dann wiederum eine konkrete Lösung darstellen kann. Hier ist die evolutionäre Erkenntnismethode nicht nur sinnvoll, sondern notwendig, wenn man eine Komponente im Sinne von Kapitel 3 als eine *potentielle* Lösung für *potentielle* Probleme betrachtet. Grob kann man sagen, dass der Bereich der Systementwicklung i.d.R. eine konkrete Zielsetzung verfolgt und dort daher typischerweise die direkte oder analytische Erkenntnismethode verwendet wird. Im Gegensatz dazu ist in der Komponentenentwicklung eine evolutionäre Methode empfehlenswert.

Jedes spezifische Entwicklungsvorhaben entscheidet sich für eine bestimmte Auffassung ihres Kontexts, die sich in dem verwendeten Kontextmodell ausdrückt. Gleichgültig, ob dies explizit dargestellt oder nur implizit verwendet wird. Zudem entscheiden sich die Entwickler jedoch auch bewusst oder unbewusst für eine Erkenntnismethode, in dem sie sich darauf festlegen, wie weit sie ihre Auffassung des relevanten Kontexts zur Disposition stellen und dazu bereit sind, ihr Modell auf neue Bereiche auszudehnen und es dementsprechend anzupassen.

Die in Abschnitt 6.4.1 charakterisierten Kontextmodelle sind relativ zueinander unterschiedlich abstrakt. Der Anwendungsfall und Nutzungsprozess haben den niedrigsten und das Anwendungsszenario den höchsten Abstraktionsgrad. Wie abstrakt nun die Auffassung eines konkreten Entwicklungsvorhaben ist, kann man so allgemein nicht sagen, da es ihre freie Entscheidung ist. So kann man ein “Open-Source-Projekt” auf einer Hosting Plattform starten und ausschließlich die eigenen lokalen Bedürfnisse bei der Entwicklung berücksichtigen oder auch eine persönliche Distribution pflegen, die nur für einen Systemknoten verwendet wird. Dies sind zwar Extreme, die hier gar nicht berücksichtigt werden sollen, veranschaulicht jedoch die Problematik, den jeweiligen Klassen von Kontextmodellen im Open-Source-Bereiche eine bestimmte Erkenntnismethode zuzuordnen. Trotzdem kann man ganz allgemein sagen, dass typischerweise bei Anwendungsszenario und Systemvorlage eher eine evolutionären Methode verwendet wird und bei den anderen Kontextmodellen je nach Umständen eher eine direkte oder analytische.

6.4.3 Die Verkettung der Kontextmodellen

In Abschnitt 6.4.1 wurde die Verkettung von Kontextmodellen im Open-Source-Lebenszyklus veranschaulicht. Vergleicht man diese Verkettung mit den klassischen Vorgehensweisen, dann lassen sich die in Abbildung 6.7 dargestellten Zusammenhänge erkennen.

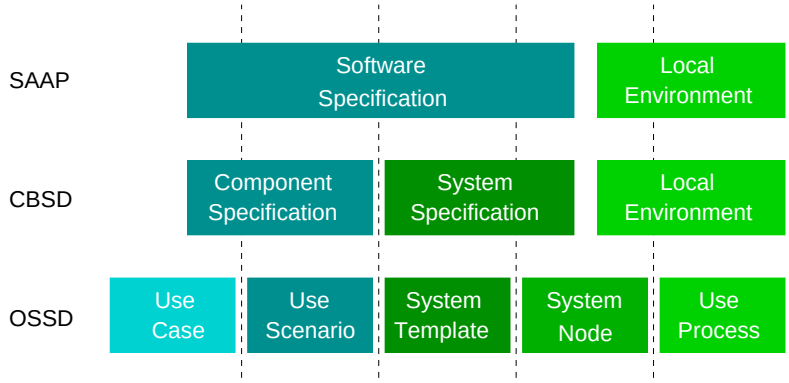

Abbildung 6.7: Zerlegung der Kontextmodelle der unterschiedlichen Vorgehensweisen im Vergleich

Das typische Vorgehen im Bereich “Software as a product” (SAAP) ist das Erstellen einer Spezifikation im Rahmen der Anforderungsanalyse vor Beginn der eigentlichen Softwareentwicklung. Sie stellt den relevanten Kontext der Software vollständig in Bezug auf ihre Nutzung dar. Dies ist notwendig, da sie oftmals auch als Grundlage für entsprechende Verträge dient. Sie bleibt dann über die gesamte Entwicklungszeit gültig. Erst in der realen Einsatzumgebung wird durch den Nutzer im Rahmen der Konfiguration oder der Nutzung ein anderes Kontextmodell verwendet.

In der allgemeinen komponentenbasierten Systementwicklung (CBSD), wie sie in Abschnitt 3.1.3 erläutert wurde kann man bereits drei Bereiche identifizieren: Komponenten, Systeme und ebenfalls die Einsatzumgebung. In der Open-Source-Entwicklung (OSSD) wurde diese Unterteilung nun weiter in die geschilderten fünf Bereiche zerlegt.

Bisher wurden jedoch nur einfache Transformationen von genau einem Kontextmodell in genau ein anderes betrachtet. Im Open-Source-Lebenszyklus kommt es jedoch beim Vereinheitlichungsprozess zu einer Zusammenführung von Ergebnissen aus mehreren unterschiedlichen Kontextmodellen und die entsprechenden Ergebnisse werden anschließend von mehreren unterschiedlichen Vorbereitungsprozessen verwendet, was faktisch einer Differenzierung in mehrere unterschiedliche Kontextmodelle entspricht wie es in Abschnitt 6.3.5 beschrieben wurde. Nach dem Vereinheitlichungs- und dem Installationsprozess finden weitere derartige Differenzierungen statt (vgl. Abb. 6.8).

Bei der Zusammenführung werden aus bereits vorhandenen Ergebnissen, die für einen spezifischen Kontext entwickelt wurden, Teile extrahiert, die in ein allgemeineres Kontextmodell passen. Diese Transformation dient dazu geeignete Änderungen aus einem spezifischen in ein allgemeineres Kontextmodell zu propagieren. Z.B. bei einer vorhergehenden Differenzierung besteht ein gewisser Bedarf dafür. Auf diese Weise können Probleme zunächst in einem einfachen, spezifischen Kontext gelöst werden und später bei entsprechender Eignung verallgemeinert werden. In einem solchen Fall wird die Entwicklungsrichtung gewissermaßen umgedreht und die zugehörige Änderung aus dem spezifischen in das allgemeinere Kontextmodell zu übertragen.

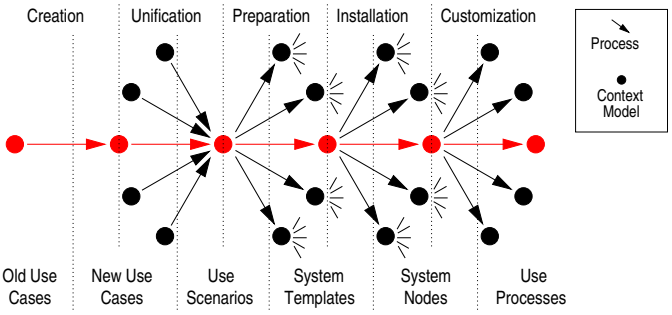

Abbildung 6.8: Verzweigungen der Kontextmodelle im Open-Source-Lebenszyklus

Das Zusammenführen und Differenzieren der Kontextmodelle haben eine Erleichterung der Entwicklungsarbeit zur Folge. Die Zusammenführung hilft redundante Entwicklungsarbeit zu sparen. Die Differenzierung reduziert die Komplexität des relevanten Kontexts in einer Entwicklungsphase was einerseits die Erfassung und anderseits die Softwareentwicklung vereinfacht.

6.4.4 Implizite Kontextmodelle

Wie in Abschnitt 6.2.3 erörtert gibt es unterschiedliche Formen des Umgangs mit Kontextmodellen. Im Open-Source-Bereich gibt es nur selten explizite Spezifikationen bzw. Kontextmodelle zu einer Software. Der Quelltext ist i.d.R. das einzige verbindliche Dokument und oftmals auch der einzige konsistente Konsens in Bezug auf den Kontext. Es ist sehr wohl möglich, dass jeder Akteur in einem Projekt eine unterschiedliche Sicht auf die Umgebung einer Software hat und damit letztliche auch von einem anderen Kontextmodell ausgeht. Dies spielt jedoch so lange für alle anderen keine Rolle, so lange es keinen wahrnehmbaren Effekt auf die Software bzw. ihre Entwicklung hat. Dass solche unterschiedlichen Standpunkte auch zu Konflikten führen, zeigen die diversen Mailinglisten (z.B. Linux-Kernel-Mailingliste). Das Erstellen einer detaillierten Spezifikation im Vorfeld ist jedoch ungleich schwieriger als ein solches Klären von spezifischen Sachfragen. Die Heterogenität der Perspektiven ist normalerweise auch kein Nachteil, sondern spiegelt nur die Realität mit ihrer Vielfalt wider. Viele Designentscheidungen stellen eine schwierige Konsensfindung dar, bei der viele unterschiedliche Aspekte der Software direkt oder indirekt beeinflusst werden. Welchem Detail welche Priorität eingeräumt wird, ist oftmals eine Frage der individuellen Interessenlage. In diesem Sinne kann man jeden Akteur innerhalb eines Entwicklungsvorhabens auch als Interessenvertreter sehen.

Die Verschmelzung von Code und Spezifikation ist nicht nur auf den Open-Source-Bereich beschränkt. Auch die Model Driven Architecture versucht diese beiden Bereiche zu verbinden, jedoch anstatt auf der technischen Code-Ebene auf einer höheren Modellebene, die von den technischen Feinheiten abstrahiert.

Open-Source-Projekte bevorzugen i.d.R. konkrete Code-Änderungen (Patches) deutlich gegenüber abstrakten Überlegungen und Vorschlägen wie die Software eigentlich sein sollte. Als Hinweise sind sie zwar gerne gesehen, aber endlose Diskussionen werden manchmal auch einfach mit der Forderung nach einer vorzeigbaren Lösung beendet (z.B. “Show me the code!”). Denn konkrete Änderungen können klar entschieden werden. Im Gegensatz dazu können abstrakte Vorstellungen leicht missverstanden werden und benötigen zum Verständnis oftmals einen höhe-

ren Kommunikationsaufwand. Es ist daher auch üblich, dass die Änderungen von den Akteuren durchgeführt werden, die ein entsprechendes Interesse daran haben. Auf diese Weise spart man sich den aufwendigen Wissenstransfer.

Änderungswünsche (Patches, Fehlermeldungen und Anregungen) können in diesem Sinne auch verstanden werden als die Bitte um Berücksichtigung eines neuen Kontexts, der zu den bisherigen Vorstellungen der entsprechenden Entwickler oder des Projektleiters nicht passt. Diese Entscheidung ist dann eine Abwägung zwischen Allgemeingültigkeit, Passgenauigkeit und dem damit jeweils verbunden Aufwand.

7 Selbstorganisation in der Open-Source-Entwicklung

Dieses Kapitel hat zwei verschiedene Funktionen innerhalb dieser Arbeit. Einerseits soll es die Anwendbarkeit und damit die Nützlichkeit des erstellten Modells zeigen, das aus den Ergebnisse der vorhergehenden vier Perspektiven-Kapiteln besteht. Anderseits soll hier die Rolle der Selbstorganisation in der Open-Source-Entwicklung ausführlicher untersucht werden, nachdem es sich im Laufe der Arbeit als eine ihrer wesentlichen Grundlagen herausgestellt hat.

Für die Erfüllung dieser beiden Aufgaben musste jedoch zunächst geklärt werden, was Selbstorganisation eigentlich ist, ob und wie weit es einen normalen, alltäglichen Vorgang darstellt, wann und wie sie in Erscheinung tritt und in welcher Form sie dann erfasst werden kann.

7.1 Grundlagen der Selbstorganisation

7.1.1 Komplexe, adaptive Systeme als Ursprung von Selbstorganisation

Axelrod/Cohen führen im Vorwort ihres Buch zu diesem Thema den Begriff komplexes adaptives System (CAS) folgendermaßen ein:

> In a world where many players are all adapting to each other and where the emerging future is extremely hard to predict, what actions should you take?
>
> We call such worlds Complex Adaptive Systems. In Complex Adaptive Systems there are often many participants, perhaps even many kinds of participants. They interact in intricate ways that continually reshape their collective future. New ways of doing things - even new kinds of participants - may arise, and old ways - or old participants - may vanish. Such systems challenge understanding as well as prediction.[Axelrod 99, S. xi]

In derartigen Systemen ist es also für die Akteure problematisch zielorientiert zu handeln, weil aufgrund der vielfältigen Interaktionen zwischen den Beteiligten das Verständnis und die Vorhersage der Zusammenhänge und Abläufe schwierig oder unmöglich ist.

Die Definition von John H. Holland bezieht das Verhältnis zwischen dem Gesamtsystem und den darin agierenden Agenten mit ein:

> A Complex Adaptive System (CAS) is a dynamic network of many agents (which may represent cells, species, individuals, firms, nations) acting in parallel, constantly acting and reacting to what the other agents are doing. The control of a CAS tends to be highly dispersed and decentralized. If there is to be any coherent behavior in the system, it has to arise from competition and cooperation among the agents

> themselves. The overall behavior of the system is the result of a huge number of decisions made every moment by many individual agents. [Waldrop 92]

Unter einem CAS wird also ein Netzwerk von vielen Agenten verstanden, die zwar bis zu einem gewissen Grad autonom handeln, aber permanent miteinander interagieren. Das Verhalten dieses Gesamtsystems ist dann die Resultierende aus dem Verhalten der Individuen.

Zum Verständnis dieses Konzepts, muss man zunächst erkennen, dass es sich dabei um eine Sichtweise handelt und nicht eine verifizierbare Aussagen über die Realität, auch wenn es wie jede Betrachtungsweise implizite Annahmen über geeignete Betrachtungsgegenstände enthält [Axelrod 99, S. xvi]:

> The Complex Adaptive System approach is a way of looking at the world. It provides a set of concepts, a set of questions, a set of design issues. By itself, it is not a falsifiable theory.

Diese Sichtweise kann theoretisch für jedes beliebiges System verwendet werden, in dem viele Akteure interagieren, gleichgültig ob es sich dabei um Zellen, Insekten, Menschen oder eine andere Gruppe von klar abgegrenzten Agenten handelt. Derartige Netzwerke sind allgegenwärtig und Teil unseres Alltags wie es Barabasi in seinem Buch zu komplexen Netzwerken beschreibt [Barabasi 03].

Damit stellt sich die Frage, wann man ein System mit Hilfe dieser Sichtweise betrachten sollte. Dies steht in enger Verbindung mit dem Komplexitätsbegriff, für den es bisher keine allgemein anerkannte Definition gibt [Axelrod 99, S. 16]. Laut Edmonds gibt es zwar einen gewissen Bezug zu den folgenden Größen, aber sie darauf zu reduzieren, sieht er als eine recht schwache Definition an [Edmonds 99]: Umfang (Anzahl der betrachteten Elemente), Unwissenheit (Unkenntnis der Details), minimale Beschreibungsgröße (notwendige Informationsmenge für eine Repräsentation), Vielfalt (Verschiedenartigkeit der Elemente) und Ordnungsgrad (Umfang der enthaltenen Regeln). Axelrod/Cohen sehen das wesentliche Merkmal in einer starken Interaktion zwischen den Elementen eines Systems [Axelrod 99, S. 7].

Die Welt ist komplex und dies lässt sich auch durch die Betrachtungsweise nicht ändern [Axelrod 99, S. 2]:

> The book does not provide a magic lens that will suddenly make the cloudy future clear. The complexity of the world is real. We do not know how to make it disappear. What the book does provide is a framework, a way of thinking through a complex setting that takes advantage of complexity to generate new questions and new possibilities for action. It suggests ways of "harnessing complexity."

Man extrahiert aus der stets komplexen Realität durch eine solche Vorgehensweise wenige Aspekte und befasst sich dann im Grunde nur noch mit der resultierenden Repräsentation. Edmonds hält die "reale" Komplexität eines Gegenstands in diesem Zusammenhang für irrelevant, da wir uns nie mit dem Gesamtsystem auseinander setzen, sondern uns stets nur mit Teilaspekte befassen. Er bezieht daher den Komplexitätsbegriff auf die Repräsentation eines Systems statt auf den betrachteten Gegenstand und definiert Komplexität wie folgt [Edmonds 99]:

> That property of a language expression which makes it difficult to formulate its overall behaviour, even when given almost complete information about its atomic components and their inter-relations.

Edmonds bezeichnet folglich ein System als komplex, wenn man sein relevantes Gesamtverhalten nur schwer in der jeweiligen Darstellungsform formalisieren kann. Daraus folgt, dass die Komplexität von der Eignung der Sprache abhängt, mit dem der jeweiligen Sachverhalt beschrieben wird. Aus diesem Blickwinkel betrachtet, kann das von Axelrod/Cohen vorgestellt Framework als der Versuch einer Sprache aufgefasst werden, um CAS möglichst einfach zu beschreiben. Es handelt sich damit um ein Meta-Modell für die Modellerstellung von passenden Realitätsausschnitten, die schwer formalisierbare Aspekte enthalten. In ihrem Fazit fassen sie ihr Framework mit Hilfe ihrer Grundbegriffe (hervorgehoben) zusammen [Axelrod 99, S. 154]:

> *Agents*, of a *variety* of *types*, use their *strategies*, in patterned *interaction*, with each other and with *artifacts*. *Performance measures* on the resulting events drive the *selection* of agents and/or strategies through processes of error-prone *copying* and *recombination*, thus changing the frequencies of the types within the *system*.

Diese Sichtweise ist hilfreich, wenn es eine Verständnislücke zwischen den Vorgängen auf unterschiedlichen Granularitätsebenen gibt, d.h. das Gesamtsystem ein Ergebnis oder Verhalten aufweist, das aufgrund der Betrachtung der einzelnen Agenten darin und ihres Verhaltens zunächst nicht zu erklären ist. Derartige Phänomene werden auch als emergent bezeichnet (vgl. [Fromm 05, Fromm 04, Heylighen 89, Johnson 01, Stephan 99]).

Die Ursache für Emergenz ist in vielen Fällen eine Selbstorganisation der Agenten innerhalb eines solchen Systems. Mit Selbstorganisation ist hier das typischerweise ungeplante Entstehen einer Ordnung aus der Interaktion der Agenten gemeint, die zu einer Wechselwirkungen zwischen ihnen und zu daraus resultierenden, emergenten Ergebnissen führt (siehe Abschnitt 7.1.2).

CAS weisen zudem wie der Name schon sagt die Eigenschaft auf, sich an ihre Umgebung anzupassen. Das COSI-Projekt hebt diesen Aspekt in seiner Definition von CAS hervor:

> Macroscopic collections of simple (and typically non-linearly) interacting units that are endowed with the ability to evolve and adapt to a changing environment. [COSI 03, complex adaptive system]

Dieser Anpassungsvorgang taucht auch in dem oben zitierten Fazit von Axelrod/Cohen auf. Die Umwelt wird hier indirekt durch die Bewertungskriterien (performance measures) abgebildet. Jedoch muss es sich dabei nicht um explizite erkennbare Kriterien handeln. Es kann sich auch um eine komplexe Umwelt handeln, deren vollständige Erfassung nicht möglich ist, die aber dem jeweiligen System als Orientierung dient.

Laut Heylighen kann man Anpassung verstehen als das Erreichen eines positiven Verhältnisses des Systems zu seiner Umgebung, so dass es erhalten bleibt oder wächst. Er sieht in der Anpassung an die Umwelt eine Voraussetzung für Selbstorganisation, die bei allen selbstorganisierenden Systemen stattfindet. Er bezeichnet Systeme als adaptiv, wenn sie sich auch an verändernde Umweltfaktoren anpassen können [Heylighen 01, S. 15].

Eine erfolgreiche Anpassung ist immer mit einer gewissen Erkenntnis über die Umwelt verbunden. Oftmals stellt dieser Adaptionsvorgang eines CAS an seine Umgebung selbst einen emergenten Vorgang dar, dessen exakter Ablauf nur schwer nachzuvollziehen ist. Bestimmte komplexe Systeme wie Organismen speichern und nutzen diese Information in Form von internen Modellen der Umwelt und verbessern damit das Verhältnis zu ihr. Dies geschieht teilweise

durch ein gesondertes Subsystem wie das Erbgut oder das Gehirn. Aber auch diese Modelle selbst müssen sich vorher irgendwie zu einer passenden Entsprechung der Umwelt entwickelt haben [Heylighen 01, S. 16].

Im Bereich der biologischen Organismen findet dies (teilweise) über die Evolution des Erbgutes statt, das als derartiges Modelle verstanden werden kann. Über die Erkenntnis in der biologischen Evolution schreibt Riedl:

> Unsere Betrachtung des stammesgeschichtlichen Werdens der Organismen erfolgt unter dem Gesichtspunkt, dass jeder erfolgreiche Schritt der Anpassung einem Zuwachs an Information über jenes Milieu entspricht, das für sie von Bedeutung ist. Wir beschreiben die Evolution als einen erkenntnisgewinnenden Prozeß. Dabei wird 'Erkenntnis' nicht als philosophischer Fachausdruck verstanden, sondern in dem allgemeinen Sinne, als die lebenden Systeme durch ihre allmähliches Entsprechen dieser Welt Gesetzlichkeiten extrahieren; wie etwa unser Auge die Gesetze der Optik wiedergebildet hat. [Riedl 81, S. 7]

In diesem Fall spiegelt sich die Güte des Umgebungsmodells in einer entsprechend erfolgreichen/-losen Population der zugehörigen Organismen wieder. Dieses "Volk" kann nun als ein CAS höherer Ordnung verstanden werden, dessen Anpassungsstrategie eine entsprechende Fortpflanzung umfasst.

Die drei beschriebenen Merkmale (Emergenz, Selbstorganisation und Anpassung) stellen die wesentlichen Kennzeichen eines komplexen adaptiven Systems dar (vgl. [Ottino 04]).

7.1.2 Selbstorganisation als Ordnungsprinzip in sozialen Systemen

Die Idee einer Selbstorganisation von komplexen Systemen wurde bereits durch Charles Darwin mit der Formulierung der Evolutionstheorie für biologische Systeme aufgebracht [Darwin 59]. Dieses Prinzip wird heute in vielen wissenschaftlichen Disziplinen für die jeweiligen fachspezifischen Systeme untersucht (Soziologie, Physik, Wissenschaftstheorie, Wirtschaftswissenschaften, etc.). "Der gemeinsame Nenner, auf den sich alle diese Überlegungen bringen lassen, ist die Frage nach der Entstehung von Ordnung" [Göbel 98, S. 17]. Ordnung ist die Voraussetzung für unser Denken und unsere Erkenntnisfähigkeit oder anders ausgedrückt: "Eine Welt ohne Ordnung wäre weder erkennbar noch denkbar" [Riedl 90, S. 25]. "Da wir ohne Ordnung nicht denken können, besteht die erste Hürde in der Schwierigkeit, zwischen realer und gedachter Ordnung zu unterscheiden" [Riedl 90, S. 26]. "Nach der radikal konstruktivistischen Erkenntnistheorie ist sogar fraglich, ob es reale Ordnung überhaupt gibt, oder ob wir sie erfinden, weil sie für uns denknotwendig ist [...] Zugleich lehrt die Erfahrung, dass sich die bewusste Herstellung von Ordnung im sozialen Bereich als umso schwieriger erweist, je größer und vernetzter die zu ordnenden Systeme werden." [Göbel 98, S. 18]. Das Eigenverhalten solcher dynamischer Systeme steigt mit ihrer Komplexität an und das System entzieht sich damit zunehmend einer aufgezwungen Konstruktion, die dadurch zu einer Illusion (Deskription) bzw. einem nicht umsetzbaren Wunsch (Preskription) wird (vgl. [von Hayek 80, S. 50] u. [Göbel 98, S. 18]). "Zwischen der Gefährdung und dem offensichtlichen Vorhandensein von Ordnung bildet sich ein Spannungsfeld" in dem "die Furcht vor dem Chaos angesichts der schwierigen Gestaltbarkeit von komplexen, dynamischen Systemen unversehens umschlagen [könnte] in eine optimistische

Gelassenheit, weil sich Ordnung 'selbstorganisierend', etwa durch evolutionäre Prozesse ergibt." [Göbel 98, S. 18].

Anderseits ist die Selbstorganisation mit einer höheren gestalterischen Kompetenz verbunden, da sie den Akteuren ein hohes Eigensteuerungspotential zugesteht. Zudem gestalten soziale Systeme (z.B. ein Unternehmen) ihre direkte Umgebung zu einem wesentlichen Teil selbst, womit vermeintliche Sachzwänge oft Produkte des eigenen Systems sind und damit letztlich veränderbar. Betrachtet man ein soziales System aus der Selbstorganisationsperspektive, so werden bestimmte Gestaltungsmöglichkeiten eröffnet und andere reduziert. Damit sich solche Systeme jedoch selbst organisieren und nicht nur eine Fremdorganisation stören können, muss zunächst ein geeigneter Rahmen existieren, der auch durch gezieltes Eingreifen erzeugt werden kann (vgl. [Göbel 98, S. 19-20]). Man muss sich also zunächst entscheiden, ob und wie weit man ein System sich selbst organisieren lässt, wobei man in den meisten Fällen dabei aufgrund der Komplexität eben kein vorhersehbares Ergebnis anstreben oder anordnen kann, sondern nur dem System die Freiheit lassen kann seine eigene Ordnung auszubilden. Selbstorganisation strebt einer systemimmanenten Ordnung entgegen, die sich mit der Veränderung des Systems selbst mitentwickelt. Bei einem dynamischen System, das sich permanent verändert, ist auch die darin enthaltene Ordnung in ständigem Fluss.

Die aus der Selbstorganisation entstehende spontane Ordnung (im Gegensatz zur künstlichen, konstruierten Ordnung) in sozialen Systemen, wie sie von von Hayek beschrieben wird, sieht Göbel durch drei wesentliche Merkmale gekennzeichnet [Göbel 98, S. 54][von Hayek 80, S. 58ff]:

- "Sie ist das Ergebnis menschlichen Handelns, aber nicht menschlichen Entwurfs. Sie ist nicht planmäßig geschaffen und unbeabsichtigt.
- Sie entsteht aus den Handlungen vieler Individuen, die nach bestimmten abstrakten Regeln handeln. Diese Regeln müssen den Individuen nicht bewusst sein.
- Die Umstände auf die Individuen reagieren, können verschieden sein und auch zu verschiedenen faktischen Handlungen führen. Solange, sie dabei bestimmten abstrakten Regeln folgen, bildet sich daraus spontan eine Gesamtordnung."

Es handelt sich dabei um eine "zweistufige spontane Ordnungsbildung". Einerseits entsteht eine Ordnung durch ein in gewissen abstrakten Aspekten ähnliches (regelhaftes) Handeln von Individuen. Anderseits können auch die Regeln, denen sie folgen, spontan entstanden sein, wobei diese Regelbildung sich nicht plötzlich ereignet, sondern sich auch über einen langen Zeitraum erstrecken kann (z.B. Generationen). Diese Regeln müssen jedoch nicht spontan entstehen, sondern können auch vorgegeben werden. So können sie einerseits aus bereits bekannten Erfahrungen extrahiert und übertragen werden oder sogar künstlich entworfen werden (vgl. [Göbel 98, S. 54]).

Die daraus resultierende Ordnung bleibt jedoch unberechenbar. Kleine Veränderungen der Umstände, der Akteure oder des Regelwerks können bereits zu völlig anderen Ergebnissen führen. Haben sich jedoch Regeln vielfach bewährt, so kann man mit einer gewissen Zuversicht darauf spekulieren, dass sie in einer ähnlichen Situation wieder ähnliche Ergebnisse hervorbringen.

All diese Überlegungen beschränken sich jedoch zunächst nur auf die Ordnung, die Struktur eines sozialen Systems. Die Ordnung hat aber eine große Bedeutung, weil sie festlegt, wie sich ein System als Ganzes verhält. Um so größer und komplexer ein System wird, um so wichtiger

wird seine Ordnung und um so mehr weicht die Bedeutung der einzelnen Akteure innerhalb des Gesamtsystems zurück. Übertragen auf ein Softwareprojekt hat die Ordnung dann einen entscheidenden Einfluss darauf wie die Softwareentwicklung abläuft, was entwickelt wird, wie das Ergebnis beschaffen ist, welche Kosten entstehen und wieviel Zeit es in Anspruch nimmt.

Zudem sind einzelne soziale Systeme (z.B. ein Softwareprojekt) immer auch Teil oder Interaktionspartner von anderen sozialen Systemen, die direkt oder indirekt über Erfolg bzw. Misserfolg des Systems mitbestimmen.

7.1.3 Stigmergie als emergente Koordinationsform

Im Abschnitt 7.1.2 wurde die Bedeutung der Ordnung in einem sozialen System erläutert und wie sie bei der Selbstorganisation durch ein regelhaftes Verhalten von Akteuren entsteht. Dabei wurde das Milieu in dem sie agieren weitgehend ausgeklammert. Dies spielt jedoch für die Softwareentwicklung eine wichtige Rolle, da es primär um ein Artefakt geht, was Teil der Umgebung ist: Software.

Die heute wesentliche Theorie zu der Rolle der Umgebung und ihrer Veränderung in der Selbstorganisation geht auf die Arbeit des Biologen Pierre-Paul Grassé zurück [Grassé 59]. Bei seiner Forschungsarbeit zur Entstehung von Termiten-Hügeln entwickelte er eine Theorie, die das Koordinationsparadoxon zwischen der individuellen und gemeinschaftlichen Beobachtungsebene auflösen konnte: das Verhalten der einzelnen Termiten und das Verhalten eines ganzen Termitenvolkes. Denn obwohl in der Handlungen der einzelnen Termiten zunächst keine Koordination zu erkennen war, agierte das gesamte Volk wie ein Organismus. Er erkannte, dass die einzelnen Insekten ihre Arbeit über die Veränderung der gemeinsamen Umwelt koordinierten, in dem sie unabhängig von einander den gegenwärtig Zustand der Umwelt als Hinweis verwenden, was als nächstes zu tun ist. Mit ihren Handlungen verändern sie wiederum die Umwelt, so dass die nächste Termite entsprechend der neuen Situation handeln kann. Grassé nannte diesen Effekt “Stigmergie”. Die Insekten folgen dabei regelhaft einem Verhaltensmuster. Die daraus entstehende Ordnung lässt sie gemeinsam agieren und daraus diesen Hügel entstehen. Dabei wird aus der Ordnung des sozialen Systems eine Ordnung in dem erstellten Artefakt (Hügel). Umgekehrt wird die Umgebung genutzt, um das soziale System zu ordnen.

Dieser Ansatz wurde in den letzten Jahrzehnten in einer Vielzahl von unterschiedlichen Forschungsbereichen aufgegriffen und dabei entsprechende Phänomene auch in anderen Bereichen identifiziert. Es hat sich dabei gezeigt, dass der zugrundeliegende Mechanismus nicht auf Insekten beschränkt ist, sondern ein allgemeines Muster darstellt, das oftmals bei der Interaktion von vielen Agenten zu beobachten ist.

> The notion of stigmergy, its relation with the environment, interaction through artifacts, and the many sorts of structures and behaviours that emerge from stigmergic coordination in complex social bodies: all are strictly inter-related issues that have been the subjects of investigation in a multiplicity of heterogeneous research areas. [Ricci 06, Abschnitt 2]

Dieser Ansatz lässt sich auch auf soziale Systeme mit menschlichen Akteuren übertragen und stellt einen grundlegenden Koordinationsmechanismus in Gesellschaften und Organisationen dar.

> [A] number of relevant works in the field of cognitive sciences put in evidence how stigmergy – as the social mechanism of coordination based on interaction through local modifications to a shared environment – is a fundamental mechanism of coordination in the context of human societies and organisations. [Ricci 06, Abschnitt 1]

Ricci et al. haben dafür ein allgemeines Modell entwickelt, das sie als "Cognitive Stigmergy" bezeichnen und führen folgende Grundbegriffe für die Beschreibung der Vorgänge ein: Agent (agent), Artefakt (artifact), Arbeitsbereich (workspace) und Anmerkungen (annotation).

Zunächst teilen sie mögliche Systemelemente in zwei Gruppen: Agenten und Artefakte. Dabei bezeichnen Agenten jene Entitäten, die sich an ihren eigenen Zielen/Aufgaben orientieren bzw. von ihrer Umsetzung angetrieben werden und eigenständig agieren. Während Artefakte jene Entitäten bezeichnen, die nicht selbstständig handeln und keine eigenen Ziele verfolgen. Sie können zwei unterschiedliche Rollen übernehmen. Einerseits können sie das Zielobjekt der Arbeit von Agenten sein (domain level) und andererseits auch ein Hilfsmittel, das sie einsetzen, um ihre Ziele zu erreichen (tool level), in dem sie eine Funktionalität oder auch einen Dienst verwenden, den das betreffende Artefakt bereitstellt. Artefakte können räumlich sehr ausgedehnt bzw. verteilt sein und auf diese Weise für die Interaktion über weite Strecken genutzt werden (z.B. Telefon oder Internet). Agenten *kommunizieren mit* Agenten, aber Agenten *nutzen* Artefakte. [Ricci 06, Abschnitt 3.1][Ricci 05, Susi 01]

Arbeitsbereiche stellen die lokale Umgebung eines Agenten dar. Mit lokal ist der Teil der Umgebung gemeint, der durch seine Handlungen direkt beeinflusst werden kann bzw. den er wahrnehmen kann. Das Prinzip der Lokalität ist bei Termiten auf die physikalische Umgebung bezogen. In der kognitiven Stigmergie kann ein Agent jedoch durch verteilte Artefakte weit über seinen physikalischen Wirkungskreis hinaus interagieren. Derartige Arbeitsbereiche können sich überschneiden bzw. verschachtelt sein und sich auf diese Weise Artefakte und Agenten teilen. [Ricci 06, Abschnitt 3.2]

In einem Stigmergie-System werden die Auswirkungen der Handlungen eines Agenten auf seine Umwelt als Zeichen verstanden, die nach ihrer Entstehung unabhängig vom dem Urheber weiter bestehen und von anderen Agenten wahrgenommen werden können. Auf diese Weise findet eine indirekte Interaktion über die betreffenden Artefakte statt. Im Gegensatz zur einfachen Stigmergie umfassen die Zeichen bei der kognitiven Stigmergie symbolische Inhalte, die Informationen beinhalten, die hier als Anmerkungen bezeichnet werden. [Ricci 06, Abschnitt 3.3]

In der kognitive Stigmergie werden Artefakte primär dazu genutzt, bei den Agenten ein Bewusstsein für die Aktivitäten anderer Agenten zu schaffen, die wiederum dazu dienen können, die eigenen Vorhaben voran zu bringen. Dieses Bewusstsein ist ein Schlüsselaspekt zur Unterstützung von emergenten Koordinationsformen, bei denen kein vorab definierter Plan die Abhängigkeiten und Wechselwirkungen zwischen den Aktivitäten (mit den zugehörigen Agenten und Artefakte) und ihre Regelung festlegt, sondern dies im Verlauf und durch die Aktivitäten selbst entsteht. [Ricci 06, Abschnitt 4]

7.2 Übertragung auf den Open-Source-Bereich

Folgt man der Darwinschen Evolutionstheorie findet Selbstorganisation auf der Erde statt, seit dem sie existiert, d.h. lange bevor es überhaupt Menschen gab. Sie ist aber auch heute allge-

genwärtig. Wann auch immer mehrere Elemente eines Systems interagieren, die keiner vorgegebenen Ordnung folgen, ergibt sich als resultierende ihrer Handlungen eine eigene Ordnung. Eine Ordnung entsteht in jedem Fall. Entweder wird sie vorgegeben, dann handelt es sich um Fremdorganisation, oder sie ergibt sich aus der Handlung selbst, dann kann man dies als Selbstorganisation bezeichnen, die folglich aufgefasst werden kann als die Abwesenheit von Fremdorganisation.

Man begegnet immer wieder dem Vorurteil, dass die Vorgabe einer Ordnung für das entstehen von sinnvollen Ergebnissen notwendig ist, was impliziert, dass Selbstorganisation nicht funktioniert oder zumindest stets schlechtere Ergebnisse hervorbringt, als eine Fremdorganisation. Diese These steht im Widerspruch zur Organisation eines Großteils des menschlichen Lebens. Selbstorganisation ist die Grundlage von vielen bewährten Systemen (z.B. Demokratie, Marktwirtschaft, Wissenschaft, Börse, Konkurrenz). Auch die persönliche Freiheit und die Selbstbestimmung des eigenen Lebens sind nicht denkbar ohne Selbstorganisation. Große Teile der Vorgänge in der Natur wie z.B. die Organisation eines Insektenstaats beruhen ebenfalls auf diesem Prinzip.

Es handelt sich also bei der Selbstorganisation um ein altbewährtes Prinzip, das sich in vielen Bereichen etabliert hat und sich auch oftmals in der Konkurrenz mit dem Versuch einer Fremdorganisation durchsetzen konnte. Allerdings ist auch der Umkehrschluss falsch, dass sie immer die richtige Wahl ist. Ansonsten bräuchten wir keine Gesetze, Standards, Konventionen, etc. Zwar haben sich sicher viele dieser Regeln in einer Selbstorganisation herausgebildet, aber heute treten sie normalerweise als vorgegebene Ordnung in Erscheinung, die jedoch oftmals durch selbstorganisierende Systeme laufend aktualisiert werden.

Beide Prinzipien haben ihre Berechtigung. Man sollte weder das eine noch das andere grundsätzlich ausschließen, sondern für eine konkrete Situation den richtigen Kompromiss zwischen vorgegebener Ordnung und Selbstorganisation finden.

Wenn man nun ein bestimmtes Ziel anstrebt und den Erfolg sicherstellen will, stellt die Vorwegnahme des Ergebnis durch die Formulierung des Ziels eine Vorgabe dar. Legt man auch noch genau durch einen Plan fest, wie dieses Ziel erreicht werden soll, wird eine noch größere Ordnung vorgegeben. Man kann nun den Weg zu diesem Ziel als eine Entwicklung von dem gegenwärtigen Ist-Zustand zu dem erwünschten Soll-Zustand ansehen, der in einem Entwicklungssystem stattfindet, was alle Elemente und Vorgänge umfasst, die für diese Entwicklung von Bedeutung sind. Der Detaillierungsgrad der Vorgaben für die Beschaffenheit des Ergebnis, den Verlauf der Entwicklung durch einen Plan und die Zusammensetzung des Entwicklungssystems legt dann den Spielraum fest, der für eine Selbstorganisation bleibt.

Handelt es sich bei dem angestrebten Ergebnis um Software oder einen Systemknoten und passt das zugehörige Entwicklungssystem zu den Darstellungen von Kapitel 2 bis 6, so basieren entsprechende Vorgaben auf Annahmen bezüglich des zugehörigen Kontexts wie es in Kapitel 6 beschrieben wurde.

Jegliche Inkonsistenz zwischen diesen Vorgaben und dem realen Kontext bzw. Verlauf führt dabei zu Problemen, die letztendlich nur durch eine Änderung der realen Umstände und/oder der entsprechenden Vorgaben behoben werden können. Entsprechende Vorgaben nutzen daher nur, soweit sie sich im Einklang mit den realen Umständen befinden.

Während eine Selbstorganisation nur eine systemimmanente Ordnung zum Vorschein bringt, versucht eine Fremdorganisation einem System eine bestimmte Ordnung “aufzuzwingen”, was

nicht immer gelingen muss. Demzufolge kann eine Fremdorganisation im Gegensatz zu einer Selbstorganisation fehlschlagen. Eine Selbstorganisation hat kein Ziel, das es verfehlen könnte und kann damit auch nicht scheitern.

Die Anforderungen einer Fremdorganisation in der Softwareentwicklung sind damit ungleich höher, da sie bei der Erstellung der Vorgaben ein umfangreiches Wissen zu dem entsprechenden System, seinem Kontext und seiner Zukunft voraussetzt. Zudem müssen entsprechende Mechanismen etabliert und erhalten werden, um die Vorgänge in dem jeweiligen System entsprechend den Vorgaben zu lenken. Bei der Softwareentwicklung wird das Wissen durch ein entsprechende Anforderungsanalyse erworben und eine Steuerung des Entwicklungssystems findet durch das Management statt wie dies in Abschnitt 5.1 beschrieben wurde.

7.2.1 Handlungsfreiheit als Mangel an vorgegebener Ordnung

In dem vorgestellten Modell der Open-Source-Entwicklung wurde nun gezeigt, dass in Bezug auf jede der untersuchten Perspektiven den einzelnen Agenten mehr Handlungsfreiheit zugebilligt wird, indem ihnen mehr Handlungsmöglichkeiten zur Verfügung stehen. Dies schafft einerseits mehr Raum für Selbstorganisation, reduziert anderseits die Fremdorganisation und senkt damit den Bedarf an zentralisiertem Wissen für die Erstellung von Vorgaben und und den Aufwand für die Durchsetzung einer vorgegebenen Ordnung.

7.2.1.1 Komponenten-Perspektive

Die Distributionen stellen Komponentensysteme im Open-Source-Bereich dar. Im proprietären Bereich ist die Nutzung von Komponentensystemen i.d.R. auf eine Black-Box-Verwendung beschränkt. Im Gegensatz dazu verpflichten die Open-Source-Lizenzen die Distributionen alle vier genannten Verwendungsarten (Black-, Glass-, Grey- und White-Box) zu ermöglichen. Dies gibt den entsprechenden Systementwicklern, umfangreiche Möglichkeiten entsprechende Komponenten beliebig zu verändern.

Zudem haben sie auch durch die öffentliche Entwicklung in den Open-Source-Projekten, die als Grundlage der Distributionspakete dienen, und teilweise auch in den Distributionen selbst die Möglichkeit an der Entwicklung selbst mitzuwirken. Auf diese Weise können sie für eine Berücksichtigung ihrer Interessen in späteren, globalen Versionen des Komponentensystems sorgen und lokale Sonderlösungen in der Zukunft vermeiden. Dieses Vorgehen kann unter Umständen die effizientere Variante darstellen.

7.2.1.2 Prozess-Perspektive

Durch die Verschiebung eines Großteils der Entwicklung in den Wartungsbereich, die Etablierung eines entsprechend effizienten Wartungsmechanismus und die daraus resultierende Einbindung der Benutzer in den Entwicklungsverlauf, entsteht die Möglichkeit zu einer frühen Prüfung und Korrektur von Ergebnissen.

Die im Open-Source-Bereich übliche kleine Schrittgröße, das iterative Vorgehen und die daraus entstehende Parallelisierung der Entwicklungsaktivitäten schafft gemeinsam die Möglichkeit der

Kurskorrektur in jedem neuen Zyklus. Auf diese Weise können zwischenzeitlich erworbenes Wissen und Änderungen im Kontext im weiteren Entwicklungsverlauf genutzt werden, die Planung muss nur kurze Zeiträume abdecken und Vorgänge und Abläufe können flexibler gestaltet werden.

7.2.1.3 Agenten-Perspektive

Die Agenten-Perspektive stellt hier einen Sonderfall dar, da bereits in dem Grundlagenkapitel dazu (s. Kap. 5) zwei Formen der Organisation unterschieden wurden: projektbasierte und kooperative Entwicklung. Dabei repräsentiert letzteres eine Methode der Zusammenarbeit, die primär auf Selbstorganisation basiert. Klassische Projekte hingegen sind primär dafür gedacht, vorhandene Vorgaben in einer effizienten Form mit Hilfe von Plänen und Management zu erfüllen, was eine Form der Fremdorganisation darstellt.

In dem vorgestellten Open-Source-Modell wurde dabei die Organisationsform in Open-Source-Projekten und anderen Prozessdomänen des Open-Source-Bereichs der kooperativen Entwicklung zugeordnet, die den agierenden Agenten ein hohes Maß an Autonomie zugesteht.

7.2.1.4 Kontext-Perspektive

Die Verzweigung des Entwicklungsverlaufs und der zugehörigen Kontextmodelle ermöglicht die Erstellung von verschiedenen Softwarevarianten für unterschiedliche Umgebungen. Dies wird durch die Möglichkeit der Prozessdomänen unterstützt, jeweils ihr eigenes Kontextmodell zu erstellen und nicht einem durchgehend gültigem Modell verpflichtet zu sein. Auf diese Weise besteht mehr Freiheit für die jeweiligen Entwickler, lokale Besonderheiten zu berücksichtigen und Details offen zu lassen, die in dem jeweiligen Stadium noch nicht endgültig entschieden werden können.

Die Nutzung von mehreren Kontextmodellen in Verbindung mit dem Verzicht auf explizite Dokumentation der Modelle, ermöglicht das hinausschieben von Entscheidungen zu dem Zeitpunkt, wenn sie zu realen Konsequenzen führen und damit ein Konsens notwendig wird. Auf diese Weise wird die Handlungsfreiheit nicht durch voreilige Entscheidungen eingeengt, die später dann vielleicht doch wieder verworfen werden.

7.2.1.5 Zusammenfassung

Der Open-Source-Ansatz zeichnet sich in allen vier Perspektiven im Vergleich zur klassischen Softwareentwicklung durch eine Reduzierung der vorgegeben Ordnung aus. Dieser “Mangel” wirkt sich auf die Akteure durch eine erhöhte Handlungsfreiheit aus. Wie bereits erläutert wird dieses strukturelle “Vakuum” durch Selbstorganisation gefüllt, in der die Ordnung als Resultierende der Einzelabläufe entsteht.

7.2.2 Strukturen der Selbstorganisation in dem Open-Source-Modell

Der CAS-Ansatz ist eine Betrachtungsform, die auf viele Bereiche in der Open-Source-Gemeinde angewendet werden kann. Bereits Raymond zeigte in seinem Aufsatz “The Cathedral

and the Bazaar" die Parallelen zwischen Open-Source-und adaptiven Systemen auf, ohne sich dabei jedoch direkt auf die Theorie der CAS zu beziehen:

> But more powerful analogies to adaptive systems in biology and economics also irresistibly suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. [Raymond 99, S. 64]

Der CAS-Ansatz wurde später auch in unterschiedlichen Betrachtungen direkt verwendet (vgl. [Axelrod 99, S. 52ff.], [Muffatto 03], [van Aardt 04]). Es wurden dabei *unterschiedliche* Systeme als CAS identifiziert und es ist anzunehmen, dass es noch viele weitere im Open-Source-Bereich gibt. Daher sollte man nicht nach *dem* System suchen, sondern davon ausgehen, dass es dort eine Vielzahl von Systemen gibt, die sinnvollerweise als CAS aufgefasst werden können. So macht es z.B. Sinn größere Projekte wie den Linux-Kernel, KDE, Apache oder Mozilla als ein CAS zu sehen, aber auch eine entsprechende Betrachtung von Distributionen oder großen Rechnerpools mit dem zugehörigen technischen Personal ist sinnvoll. Bezieht man eine ausreichend große Benutzergruppe ein, kann man auch um fast jede OSS-Komponente ein passendes System identifizieren. Anderseits kann man auch die Open-Source-Gemeinde als Ganzes auf diese Weise betrachten.

Es geht also nicht (mehr) um die Frage, ob es sich bei der Open-Source-Gemeinde um eines oder mehre CAS handelt bzw. ob sie als solche aufgefasst werden kann/können, sondern um die Identifikation, explizite Darstellung und Analyse entsprechender Systeme, d.h. es sind entsprechende Modelle gesucht.

Ähnlich sieht es bei der Anwendung des Stigmergie-Prinzips aus. Das indirekte Interagieren von Agenten über Artefakte ist für Termiten eine erstaunliche Erkenntnis, für die Interaktion von Menschen ist diese Aussage allein jedoch trivial. Auch die Feststellung, dass nicht jeder Akteur, alle Auswirkungen seines Handelns überschaut oder die Zukunft nicht vollständig vorhersehbar ist, ist keine neue Erkenntnis. Da Open-Source-Software von Menschen entwickelt wird, muss man daher davon ausgehen, dass es dort einige Abläufe gibt, die dem Stigmergie-Prinzip folgen. Die Aussage, dass in der Open-Source-Entwicklung Stigmergie stattfindet ist daher ebenfalls trivial.

Ähnlich wie bei den CAS ist jedoch die konkrete Identifikation, explizite Darstellung und Analyse entsprechender Abläufe sowie die Erklärung von emergenten Phänomen damit außerordentlich schwierig. Man ist also auch hier auf der Suche nach passenden Modellen. Robles et al. haben dazu einen ersten Beitrag geleistet, in dem sie ein Modell erstellt haben, wie die vielen Entwickler ihre Zeit und Energie auf die vielen vorhanden Open-Source-Projekte verteilen [Robles 05][Robles 06, S. 191], das auf dem ursprünglichen Stigmergie-Prinzip von Insekten basiert. Elliott stellt ein allgemeines Framework für die Zusammenarbeit von Massen mit Hilfe von Stigmergie vor. Er geht dabei auch auf das Open-Source-Phänomen ein und greift die Arbeit von Robles auf [Elliott 07]. Heylighen wiederum zieht das Stigmergie Prinzip heran, um zu erklären wie Open-Access-Entwicklung (zu der man auch die Open-Source-Entwicklung zählt) effizient koordiniert werden kann [Heylighen 07].

Das vorgestellte Open-Source-Modell kann nun genutzt werden, um diese Überlegungen weiter zu führen, in dem ein passendes CAS und entsprechende Stigmergie-Mechanismen darin

identifiziert werden.

Dafür wird der ganze Open-Source-Bereich als ein CAS angesehen. Das globale Open-Source-Softwaresystem stellt dabei den zentralen Gegenstand dar, der letztlich das Entwicklungssystem abgrenzt wie der Termitenhügel gewissermaßen das Stigmergie-System der Termiten abgrenzt. Die Open-Source-Gemeinde hat es in den letzten zwanzig Jahren geschafft, ein Software-Infrastruktur zu bauen, die von vielen Millionen Menschen in der einen oder anderen Form genutzt wird: über den Zugriff auf Inhalte von entsprechenden Webservern, Nutzung von zugehörigen Diensten oder auch als Software auf dem eigenen Computer. Sie setzt sich aus tausenden von Komponenten zusammen, die auf Millionen von verschiedenen Systemknoten mit unterschiedlichem Kontext installiert wurde. Theoretisch kann man diese Systemknoten zwar als unabhängige Elemente ansehen, aber die Vernetzung, Interaktion und die wechselseitigen Abhängigkeiten der konstituierenden Softwarekomponenten legen es nahe, diese gesamte Infrastruktur als ein einziges globales Softwaresystem anzusehen wie dies z.B. für das vollständige Internet und den damit verbunden Geräten bereits durch den Ansatz vom “Globalen Gehirn” getan wird (vgl. [Heylighen 05]). Dieses globale Softwaresystem ist das Ergebnis von vielen kleinen unabhängigen Vorgängen und seine Entstehung kann somit als emergent angesehen werden.

Es lassen sich dann aus dem vorgestellten Modell der Open-Source-Entwicklung folgende Abläufe identifizieren, die von der Gesamtheit aller Akteure gelöst werden und als emergente Phänomene angesehen werden können:

Kontext-Erfassung Damit dieses globale Softwaresystem effizient und den Bedürfnissen der Nutzer entsprechend mit seiner Umgebung interagieren kann, ist es notwendig den relevanten Kontext zu erfassen und das Softwaresystem daran anzupassen. Dies umfasst für jede Komponente alle in Abschnitt 6.2 betrachteten Kontext-Bereiche für jeden Systemknoten, auf dem sie genutzt wird. Dieser globale Kontext besitzt eine enorme Vielfalt und einen außerordentlichen Umfang. Seine vollständige Erfassung in Form eines großen, detaillierten Kontextmodells ist daher nicht möglich. Relevante Aspekte werden daher direkt in die Software hinein kodiert und auf diese Weise direkt berücksichtigt. Aufgrund der mehrstufigen und verzweigten Entwicklung geschieht dies in vielen kleinen Prozessdomänen, die dann in der Summe diesen globalen Kontext erfassen.

Qualitätssicherung Jedoch ist für eine optimale Interaktion zwischen dem Softwaresystem und der Umwelt nicht nur die Erfassung des Kontext notwendig, sondern auch die Umsetzung von entsprechenden Erkenntnissen in Form von Erstellung, Änderung, Erneuerung und Ausmusterung von entsprechenden Softwareelementen. Bei derartigen Anpassungen können sich auch wieder neue Probleme ergeben bzw. die Lösungen können mangelhaft sein. Oftmals ist es schwierig die Konsequenzen einer Änderung in einem derartig komplexen System abzuschätzen und es ist daher eine schwierige Entscheidung, ob eine Änderung für das Gesamtsystem eine Verbesserung oder eine Verschlechterung darstellt. Es muss daher in der Open-Source-Gemeinde einen Mechanismus geben, der zumindest in der Tendenz für eine Verbesserung des globalen Systems sorgt.

Koordination Da es sich bei der Open-Source-Entwicklung um eine wartungszentrierte Methode handelt, laufen Nutzung und Überarbeitung gewissermaßen parallel ab. Zudem wird in vielen Tausend bzw. Millionen Prozessdomänen gleichzeitig an dem globalen Softwaresystem gearbeitet. Alle diese Aktivitäten aufeinander abzustimmen ist mit einer zentralen Form der Koordination nach Brooks Gesetz nicht möglich, da dabei der Koordinations-

aufwand exponentiell zum Umfang der Aktivitäten steigt [Brooks 95]. Stattdessen muss die Koordination dezentral zwischen den Prozessdomänen stattfinden.

Die Open-Source-Gemeinde musste u.a. diese drei Aufgaben bewältigen, um das heute real existierende globale Softwaresystem zu erstellen. Da es keine zentrale Steuerungsinstanz gibt und eine solche wohl auch nicht in der Lage wäre diese Probleme zu lösen, müssen die Agenten diese mit Hilfe von Selbstorganisation bewältigen.

Bei einem Abgleich der kognitiven Stigmergie mit dem vorgestellten Open-Source-Modell ist die Agenten-Perspektive von besonderer Bedeutung. Insgesamt lassen sich die vier wesentlichen Elemente des Stigmergie-Modells wie folgt zuordnen:

Arbeitsbereiche Das in Abschnitt 4.2.3 eingeführt Konzept der Prozessdomänen hat sich im Verlauf dieser Arbeit als wesentliche Struktureinheit des Open-Source-Bereichs herausgestellt. Dabei bilden Open-Source-Projekte nur einen Sonderfall dieser Organisationsform (vgl. Abs. 5.4.1). Letztlich finden alle Prozesse des Open-Source-Lebenszyklus in einer solchen Domäne statt. Sie bilden den Wahrnehmungs- und Wirkungshorizont eines Agenten. Dabei muss jedoch bedacht werden, dass ein Akteur in mehreren solcher Prozessdomänen aktiv sein kann, wobei er dabei jeweils eine andere Rolle einnimmt. Innerhalb dieser Arbeitsbereiche findet nun i.d.R. eine kooperative Entwicklung statt (vgl. Abs. 5.4.4).

Agenten Hierbei handelt es sich um Akteure in ihren jeweiligen Rollen innerhalb eines Arbeitsbereichs. Sie können dabei die drei identifizierten Funktionen Anregen, Gestalten und Lenken übernehmen (vgl. Abs. 4.1.2 und 5.4.4).

Artefakte Die wesentlichen Artefakten in einer Prozessdomäne sind Softwarekomponenten und Aufgaben (vgl. Abs. 5.4.4). Die verschiedenen Ausprägungen von Komponenten stellen das zentrale Artefakt innerhalb dieser Arbeitsbereiche (Prozessdomänen) dar, da sie einerseits für die Agenten als Werkzeuge für das Erreichen ihrer Ziele dienen (tool level) und andererseits auch Gegenstand ihrer Arbeit sind (domain level).

Anmerkungen Sie stellen Meta-Informationen dar, die anderen Agenten helfen sich für zukünftige Betätigungsfelder zu entscheiden. Anmerkungen sind vor allem in den Aufgabe-Artefakten enthalten (vgl. Abs. 5.4.3). Jedoch enthält auch der gegenwärtige Zustand einer Komponente Anmerkungen. Dies können explizit bereitgestellte Informationen sein (z.B. Einträge in Logs von Code-Verwaltungssystemen) oder auch implizit der Code selbst.

Die Grundbegriffe aus der Agenten-Perspektive lassen sich also den wesentlichen Elementen der kognitiven Stigmergie zuordnen. Die in der zugehörigen Arbeitsweise innerhalb eines Prozessdomäne dargestellte grundlegende Koordination findet primär über die Artefakte statt (vgl. Abs. 5.4 und 5.2.3), die allerdings durch zusätzliche Mittel wie direkte Kommunikation unterstützt wird. Dies bezieht sich jedoch ausschließlich auf die Vorgänge innerhalb einer Prozessdomäne. Die Koordination zwischen den Prozessdomänen ist ungleich komplexer. So sind die Arbeitsbereiche untereinander vernetzt und Ergebnisse von unterschiedlichen Prozessdomänen treffen in anderen Prozessdomänen zusammen. So werden in einer Vereinheitlichungsdomäne (Open-Source-Projekt) Patches zusammengeführt, in einer Distribution Releases kombiniert und harmonisiert und bei der Nutzung interagieren über die Vernetzung der Systemknoten Komponenten unterschiedliche Distributionen miteinander. Die dabei entstehenden Erfahrungen werden dann über die Domänengrenzen hinaus genutzt. Dies geschieht oftmals über Artefakte (z.B.

Patches oder Fehlerberichte). Die Prozessdomänen überschneiden sich dabei an vielen Stellen und teilen sich dadurch Agenten und Artefakte.

Es gibt oftmals mehrere alternative Lösungsmöglichkeiten für eine Aufgabe, was eine Art Überproduktion darstellt. Da Komponenten in der einen Prozessdomäne weiterentwickelt und in einer anderen genutzt werden, kann man von einer gewissen Unabhängigkeit zwischen Produktion und Nutzung sprechen. Diese Trennung in Verbindung mit der Überproduktion führt zu einer entkoppelten Selektion im Rahmen der Nutzung. Dabei handelt es sich um einen Evolutionsprozess, der ein weiteres fundamentales Prinzip der Selbstorganisation darstellt. Diese Evolution sorgt z.B. dafür, dass Prozessdomänen verkümmern. Anderseits können sich dann leicht neue Prozessdomänen bilden, wenn es einen hohen Bedarf für eine entsprechende Komponente gibt.

In jedem Fall konzentrieren sich die Agenten jeweils auf ihre lokalen Ziele, um diese jedoch erreichen zu können, müssen sie oftmals Arbeiten leisten, die einen globalen Effekt haben und in letzter Konsequenz zu den emergenten Ergebnissen führen.

Zusammenfassend kann man sagen, dass die Open-Source-Entwicklung in vielen Bereichen auf Selbstorganisation basiert und viele Vorgänge gar nicht anders zu erklären sind. Es wird jedoch wohl noch einige Forschungsarbeit erfordern, bis man die zugrundeliegenden Abläufe zufriedenstellend erfasst hat.

8 Ergebnis

8.1 Paradigmenwechsel

O'Reilly vertritt in seinem Aufsatz "The Open Source Paradigm Shift" die These, dass Open-Source einen Paradigmenwechsel in der Informationstechnologie im Sinne von Kuhn darstellt [Feller 05, S. 461-481][Kuhn 93]. Er sieht dabei in Open-Source den Ausdruck von drei langfristigen Trends: Wandel der Software zu austauschbaren Komponenten, internetbasierte Zusammenarbeit und Anpassbarkeit von Software (softwarebasierte Dienste).

Die vorliegende Arbeit stützt die These, dass dem Open-Source-Phänomen ein Paradigmenwechsel im Sinne von Kuhn zugrunde liegt. Es stellt sich jedoch die Frage ob die Trennlinie wirklich zwischen proprietärer und Open-Source-Software verläuft. Aufgrund von den gefundenen Ergebnissen in dieser Arbeit lassen sich zwei Paradigmen in der Softwareentwicklung von einander abgrenzen, die verschiedene Merkmale aufweisen. Diese wurden in Tabelle 8.1 gegenüber gestellt.

Merkmal	Abschnitt	Produkt	Dienst
primärer Fokus der Entwicklung	3.1.3	Komponente	System
Organisationsform	5.3	Projekt	kooperative Entwicklung
Auffassung des Kontexts	7.1.1	deterministisch	komplex
Umfangreichste Erkenntnismethode	6.1.4	analytisch	evolutionär
Entwicklungsprinzip	4.4.4	Perfektion	Evolution
Gültigkeitsbereich der Kontextmodelle	6.3	global	lokal

Tabelle 8.1: Gegenüberstellung der Paradigmen

Bei dem Gültigkeitsbereich der Kontextmodelle ist mit "global" ein einheitliches Kontextmodell für alle beteiligten Prozessdomänen gemeint. Findet ein Wechsel der Kontextmodelle statt (Verkettung bzw. Verzweigung), werden sie als "lokal" bezeichnet.

Man könnte die Liste dieser Merkmale noch beliebig um zusätzliche Elemente erweitern. Entscheidend ist jedoch dabei die Auffassung vom Kontext. Bei der Auffassung des Kontexts ist mit "deterministisch" gemeint, dass entsprechende Systeme (vgl. Abs. 6.2.1) und ihre Vorgänge als erfassbar, berechenbar und planbar angesehen werden. Dies ist dann der Fall, wenn die darin enthaltene Ordnung für den Betrachtungszeitraum ausreichend einfach, erkennbar und stabil ist, so dass man ihn mit den zur Verfügung stehenden Mitteln in einem formalen Modell erfassen kann ohne dabei wesentliche Aspekte vernachlässigen zu müssen. Nimmt man an, dass dies nicht möglich ist, wird der Kontext als "komplex" aufgefasst.

Das Produkt-Paradigma benötigt einen deterministischen Kontext und stellt damit eine Forderung an den relevanten Kontext. Wie in Abschnitt 7.1.1 erläutert, ist die Realität grundsätzlich komplex. Daher ist diese Bedingung nicht darauf bezogen, sondern auf den als relevant angesehenen Ausschnitt davon und damit letztlich auf dessen Modell. Es ist also die Frage, ob man die Gültigkeit dieses Modells sicherstellen kann. Dies schließt auch die Gewährleistung der Abdeckung aller relevanten Aspekte mit ein.

Da das Dienst-Paradigma von vornherein davon ausgeht, dass der Kontext nicht kontrollierbar ist, gilt hier nicht diese Einschränkung. Allerdings zieht dies auf der anderen Seiten einen höheren Aufwand nach sich und die (angenommene) Unberechenbarkeit des Kontexts wird in die Entwicklung übertragen. Daher ist seine Anwendung zwar stets möglich, aber nur bei einem komplexen Kontext sinnvoll.

Die Erfahrung zeigt, dass Software heute größtenteils in diesem Sinne einen komplexen Kontext hat, seine Reduzierung auf eine beherrschbares Maß (z.B. durch Abstraktion) inzwischen keine zufriedenstellenden Ergebnisse mehr liefert und die Sicherstellung der Kontextmodelle inzwischen erhebliche Belastungen und Nachteile mit sich bringt. Aus diesen Gründen kann man einen allgemeinen Trend in der Informationstechnologie zu dem Dienst-Paradigma beobachten.

Folgt man dem Dienst-Paradigma bedeutet dies in vielerlei Hinsicht die Aufhebung von Ordnungszwängen. Dies zieht nicht notwendigerweise die Auflösung der zugehörigen Ordnungsstrukturen nach sich, da sie auch aus anderen Gründen aufrecht erhalten werden können (z.B. gesetzliche Bestimmungen, unternehmerische Vorgaben, etc.). Allerdings besteht so die Möglichkeit auf sie zu verzichten bzw. sie durch neue zu ersetzen.

In Abschnitt 7.2.1 wurde mit Hilfe des dargestellten Modells der Open-Source-Entwicklung gezeigt, dass die Open-Source-Gemeinde genau dies gemacht hat. Sie haben Ordnungsstrukturen abgelegt, die in der klassischen Softwareentwicklung primär wegen des Produkt-Paradigmas üblich sind, und mit Hilfe von Selbstorganisation wurden neue gebildet. Obwohl diese neuen Strukturen sich auch weiterhin durch die Selbstorganisation in einem permanenten Wandel befinden, haben sich in der Open-Source-Entwicklung in den letzten zwanzig Jahren gewisse Strukturen (vorübergehend) stabilisiert.

Die vorliegende Arbeit hat versucht diese relativ stabilen Ordnungsstrukturen zu identifizieren, zu belegen und darzustellen.

8.2 Extrahierte Grundlagen aus der klassischen Softwaretechnik

In den Abschnitten zur theoretischen Fundierung von allen Perspektiven konnten ausreichend Strukturen und Konzepte aus der klassischen Softwaretechnik ermittelt werden, die auf den Open-Source-Bereich übertragbar sind. Es hat sich jedoch auch gezeigt, dass es wesentliche Unterschiede gibt, die zu einem großen Teil auf die beiden unterschiedlichen Paradigmen zurückzuführen sind, die in dem jeweiligen Bereich vorherrschen. So musste bei der Übertragung hauptsächlich in Bezug auf Produktorientierung, Spezifikation, Planung und Management abstrahiert werden. Dies wurde durch einen Rückzug auf fundamentale Theorien bewerkstelligt. So wurden aus Spezifikationen Kontextmodelle, aus Projekten die kooperative Entwicklung, aus detaillierten Vorgehensmodellen der Einheitsprozess und aus der komponentenbasierten Entwicklung das allgemeine Konzept der Komponentensysteme abgeleitet. Diese Basiskonzepte dienten als Ausgangspunkt für die verwendeten Perspektiven.

Die Identifikation geeigneter Perspektiven und ihrer Grundlagen stellte zunächst ein Problem dar, das durch das allgemeine Modell der Entwicklung von Computersystemen in Kapitel 2 gelöst wurde. Aufgrund der hohen Abstraktion ist es sowohl für die klassische Softwaretechnik wie auch den Open-Source-Bereich gültig. Mit Hilfe dieses Modells konnten dann jeweils ähnliche Strukturen in beiden Bereichen ermittelt werden, aus denen sich passende Basiskonzepte und die zugehörige Perspektiven entwickeln ließen.

Dabei zeigte sich zudem, dass Softwareentwicklung letztendlich immer ein Arbeiten auf mehreren Ebenen gleichzeitig ist. Diese Ebenen zeichnen sich jeweils durch den Gegenstand aus, der durch eine Veränderung betroffen ist: Objekt, Komponente, Informationssystem, Systemknoten, Nutzungssystem und Softwarepool. Das eigentliche Ziel ist dabei stets die Veränderung der Abläufe im Nutzungssystem. In diesem Zusammenhang wird der Systemknoten nur als eine vollständige funktionale Einheit wahrgenommen. Im Open-Source-Bereich ist dabei im Vergleich zur klassischen Softwaretechnik eine Verschiebung weg von der Komponente hin zum System zu beobachten, d.h. es wird stärker auf der Systemebene (mit Komponenten) entwickelt, als auf der Komponentenebene die Internas verändert. Dadurch werden die Vorgänge zwischen den Komponenten und den zugehörigen Open-Source-Projekten wichtiger. Entsprechend wurde auch die Betrachtung der Entwicklung auf diesen Bereich ausgedehnt.

8.3 Modell der Open-Source-Entwicklung

Die gefundenen Perspektiven und die zugehörigen Grundlagen wurden dann für die Erstellung des Modells der Open-Source-Entwicklung genutzt. Dabei konnten einige wesentliche Merkmale mit Hilfe dieser Sichtweisen identifiziert werden:

Komponenten-Perspektive Die typischen Distributionen von GNU/Linux können als Komponentensysteme aufgefasst werden. Die Kombination von verschiedenen Nutzungsarten statt der üblichen Beschränkung auf die Black-Box-Verwendung ist hierbei eine charakteristische Besonderheit, aus der wesentliche Strukturen und Vorgänge der Open-Source-Entwicklung resultieren.

Prozess-Perspektive In der klassischen Softwaretechnik wird zwischen der Entwicklungsarbeit bei der initialen Erstellung und Wartung unterschieden, die durch die (erste) Auslieferung der Software von einander getrennt sind. Überträgt man diese Einteilung auf den Open-Source-Bereich findet die typische Entwicklung ausschließlich in der Wartung statt. Diese Entwicklungsarbeit zeichnet sich aus durch eine besondere Aufgabenteilung. Dabei findet durch kleine Schrittgrößen eine Parallelisierung der Entwicklungsaktivitäten statt. Daraus entsteht ein abweichender Open-Source-Lebenszyklus, der als Wartungszyklus gesehen werden kann.

Agenten-Perspektive Obwohl der Name es suggeriert, handelt es sich bei Open-Source-Projekten nicht um Projekte im klassischen Sinn. Sie stellen viel mehr die Umgebung dar, in denen einfache Entwicklungsprozesse stattfinden, und sind eine Sonderform der Prozessdomänen. Diese Prozessdomänen stellen die zentrale Struktureinheit in der Open-Source-Entwicklung dar und werden als kooperative Entwicklung organisiert.

Kontextmodell-Perspektive Aufgrund von der fortlaufenden Überarbeitung von Ergebnissen handelt es sich bei der Arbeit innerhalb der Prozessdomänen der Open-Source-Entwicklung um eine evolutionäre Erkenntnismethode. Die verwendeten Kontextmodelle

bleiben dabei implizit und ihre Gültigkeit wird als auf die jeweilige Prozessdomäne beschränkt angesehen. Dadurch entsteht eine Verzweigung der Kontextmodelle, die eine differenzierte Behandlung der lokalen Umstände und eine kontextnahe Entwicklung erlaubt. Auf diese Weise entfallen große Teile des aufwendigen Wissenstransfers.

8.4 Verwendung des erstellten Modells

Der ursprüngliche Ausgangspunkt für die hier dargestellte Forschung war die Untersuchung von unterstützenden Infrastrukturen für die Open-Source-Entwicklung [Evers 00]. Dabei wurden die Unklarheiten erkannt, die mit der Open-Source-Entwicklung verbunden sind, und der Bedarf eines entsprechenden Modells offensichtlich. Die Arbeit an diesem Modell führte zu grundsätzlichen Fragen, deren Beantwortung ein allgemeines Modell nahe legten, und damit zur Loslösung von den Infrastrukturen führte. Trotzdem wurde u.a. dieser Anwendungsfall des Modells laufend zur Überprüfung der Ergebnisse herangezogen. Im Rahmen dieser exemplarischen Anwendung entstanden bereits drei Diplomarbeiten, die auf einer früheren Version des gefundenen Modells basieren [Suhr 07, Tolzmann 07, Klink 07]. Sie stellen u.a. einen Beleg für die Anwendbarkeit des gefundenen Modells dar.

Die Anwendung des Modells ist jedoch nicht auf die zugehörigen Infrastrukturen beschränkt. Es kann in vielerlei Form zum Verständnis, zur Orientierung und als Grundlage für weitere Arbeiten dienen. So wurde es z.B. in Kapitel 7 für die Betrachtung der Rolle der Selbstorganisation in der Open-Source-Entwicklung genutzt.

Der wesentliche Nutzen dabei ist die mögliche Loslösung von dem nicht erfassbaren Open-Source-Phänomen, das stets nur subjektiv wahrgenommen werden kann. Stattdessen kann man sich auf das vorliegende Modell der Open-Source-Entwicklung beschränken und damit arbeiten. Dass die hier dargestellten Strukturen ein passendes Modell für den jeweiligen Nutzungskontext sind, kann jeweils begründet und belegt werden, aber es bleibt letztendlich ein subjektives Urteil.

Literaturverzeichnis

[Abran 04] Alain Abran, Pierre Bourque, Robert Dupuis, James W. Moore & Leonard L. Tripp (Hrsg.). *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2004.

[Allen 98] Paul Allen & Stuart Frost. *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*. Cambridge University Press, New York, NY, USA, 1998.

[Arief 04] B. Arief & C. Gacek. *The Many Meanings of Open Source*. Software, IEEE, Band 21, Nr. 1, January 2004.

[Axelrod 97] Robert Axelrod. *The Complexity of Cooperation*. Princeton University Press, August 1997.

[Axelrod 99] Robert Axelrod & Michael D. Cohen. *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. Free Press, New York, 1999.

[Axelrod 06] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, New York, revised Auflage, 2006.

[Balzert 08] Helmut Balzert. *Lehrbuch der Softwaretechnik: Softwaremanagement*. Spektrum Akademischer Verlag, Berlin, 2. Auflage, 2008.

[Barabasi 03] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life*. Plume Books, New York, April 2003.

[Bezroukov 99] Nikolai Bezroukov. *Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)*. First Monday, Band 4, Nr. 10, October 1999.

[Biggerstaff 89] T. J. Biggerstaff & C. Richter. *Reusability Framework, Assessment, and Directions*. Seiten 1–17, 1989.

[Brooks 87] Frederick P. Brooks. *No Silver Bullet - Essence and Accidents of Software Engineering.* IEEE Computer, Band 20, Nr. 4, Seiten 10–19, 1987.

[Brooks 95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, Boston, USA, 20th anniversary Auflage, August 1995.

[Brown 02] Alan W. Brown & Grady Booch. *Reusing Open Source Software and Practices: The Impact of Open-Source on Commercial Vendors*. In Cristina Gacek (Hrsg.), *Software Reuse: Methods, Techniques, and Tools. 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002. Proceedings (Lecture Notes in Computer Science Vol. 2319)*, Seiten 123–136, 2002.

[Brügge 04] Bernd Brügge, Dietmar Harhoff, Arnold Picot, Oliver Creighton, Marina Fiedler & Joachim Henkel. *Open-Source-Software – Eine ökonomische und technische Analyse*. Springer-Verlag, Berlin, 2004.

[Buschermöhle 06] Ralf Buschermöhle, Heike Eekhoff & Bernhard Josko. *SUCCESS: Erfolgs- und Misserfolgsfaktoren bei der Durchführung von Hard- und Software-Entwicklungsprojekten in Deutschland*. BIS Verlag, Oldenburg, 2006.

[ChaosReport 03] *CHAOS Chronicles v3.0*. Forschungsbericht, The Standish Group, 2003.

[Checkland 90] Peter Checkland & Jim Scholes. *Soft Systems Methodology in Action*. John Wiley & Sons, New York, 1990.

[Clements 07] Paul Clements & Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2007.

[COSI 03] *COSI Website: Research Training Network; Glossary*, 2003. Abgerufen von http://www.irit.fr/COSI/glossary/ am 02.04.2008.

[Darwin 59] Charles Darwin. *On the Origin of Species*. John Murray, London, 1859.

[Debian 07] *Debian Project Website*, 2007. Abgerufen von http://www.debian.org am 22.04.2007.

[DiBona 99] Chris DiBona, Sam Ockman & Mark Stone (Hrsg.). *Open Sources - Voices from the Open Source Revolution*. O'Reilly & Associates Inc., Sebastopol, 1999.

[Dietze 04] Stefan Dietze. *Modell und Optimierungsansatz für Open Source Softwareentwicklungsprozesse*. Ph.D. dissertation, University of Potsdam, March 2004.

[Dolstra 06] Eelco Dolstra. *The Purely Functional Software Deployment Model*. Ph.D. dissertation, University of Utrecht, 2006.

[Edmonds 99] Bruce Edmonds. *What is Complexity? - The Philosophy of Complexity Per Se with Application to Some Examples in Evolution*. In Francis Heylighen, Johan Bollen & Alexander Riegler (Hrsg.), *The Evolution of Complexity*, Dordrecht, 1999. Kluwer Academic Publishers.

[Elliott 07] Mark Elliott. *Stigmergic Collaboration: A Theoretical Framework for Mass Collaboration*. Ph.D. dissertation, The University of Melbourne, 2007.

[Endres 03] Albert Endres & Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Addison Wesley, May 2003.

[Estublier 99] Jacky Estublier (Hrsg.). *System Configuration Management, 9th International Symposium, SCM-9, Toulouse, France, September 5-7, 1999, Proceedings*, Band 1675 von *Lecture Notes in Computer Science*. Springer, 1999.

[Evers 00] Steffen Evers. *Development Environments for Open Source Software*. diploma thesis, Technical University of Berlin, 2000.

[Fedora 08] *Fedora Project Website*, 2008. Abgerufen von http://www.fedoraproject.org am 22.02.2008.

[Feistner 06] Edith Feistner & Alfred Holl. *Mono-Perspective Views of Multi-Perspectivity: Information Systems Modeling and 'The Blind Men and the Elephant'*. Nr. 87: Information systems, 2006.

[Feller 02] Joseph Feller & Brian Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley Longman, Amsterdam, 2002.

[Feller 05] Joseph Feller, Brian Fitzgerald, Scott A. Hissam & Karim R. Lakhani (Hrsg.). *Perspectives on Free and Open Source Software*, London, July 2005. The MIT Press Ltd.

[Flood 93] Robert Flood & Ewart Carson. *Dealing with Complexity: An Introduction to the Theory and Application of Systems Science*. Plenum Press, New York, 2. Auflage, 1993.

[Fritschi 02] Hans Fritschi. *A Component Framework to Construct Active Database Management Systems*. Ph.D. dissertation, University of Zurich, December 2002.

[Fromm 04] Jochen Fromm. *The Emergence of Complexity*. Kassel University Press, 2004.

[Fromm 05] Jochen Fromm. *Types and Forms of Emergence*. Forschungsbericht, Jun 2005.

[Gnodtke 03] Alexander Gnodtke. *Model of a Platform for Open Source Software Development in South Africa*. diploma thesis, Technical University of Berlin, 2003.

[Grassé 59] Pierre-Paul Grassé. *La reconstruction du nid et les coordinations inter-individuelles chez Bellicositermes natalensis et Cubitermes sp. La théorie de la stigmergie: essai d'interprétation du comportement des termites constructeurs*. Insectes Sociaux, Band 6, Seiten 41–81, 1959.

[Göbel 98] Elisabeth Göbel. *Theorie und Gestaltung der Selbstorganisation*. Betriebswirtschaftliche Forschungsergebnisse. Duncker und Humblot, Berlin, 1998.

[Haug 01] Michael Haug, Eric W. Olsen, Gonzalo Cuevas & Santiago Rementeria (Hrsg.). *Managing the Change: Software Configuration and Change Management*. Springer-Verlag, London, UK, 2001.

[Heineman 01] George T. Heineman & William T. Councill (Hrsg.). *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Heylighen 89] Francis Heylighen. *Self-Organization, Emergence and the Architecture of Complexity*. In *Proceedings of the 1st European Conference on System Science, (AFCET, Paris)*, Seiten 23–32, 1989.

[Heylighen 01] Francis Heylighen. *The Science of Self-Organization and Adaptivity*. In L. D. Kiel (Hrsg.), *The Encyclopedia of Life Support Systems(EOLSS):*

Knowledge Management, Organizational Intelligence and Learning, and Complexity, Oxford, 2001. Eolss Publishers.

[Heylighen 05] Francis Heylighen. *Conceptions of a Global Brain: An Historical Review*. In *Technological Forecasting and Social Change*, 2005.

[Heylighen 07] Francis Heylighen. *Warum ist Open-Access-Entwicklung so erfolgreich? Stigmergische Organisation und die Ökonomie der Information*. In Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.), *Open Source Jahrbuch 2007 – Zwischen freier Software und Gesellschaftsmodell*. Lehmanns Media, Berlin, 2007.

[Holl 99] Alfred Holl. *Empirische Wirtschaftsinformatik und evolutionäre Erkenntnistheorie*. In Jörg Becker, Wolfgang König & Reinhard Schütte (Hrsg.), *Wirtschaftsinformatik und Wissenschaftstheorie. Bestandsaufnahme und Perspektiven*, Seiten 163–207. Gabler Verlag, Wiesbaden, 1999. Translation: Information Systems as Empirical Science and Evolutionary Epistemology.

[Holl 02a] Alfred Holl. *Nutzen und Tücken von Analogieschlüssen in der Verbalmorphologie: Rückläufige Ähnlichkeit als tertium comparationis in ausgewählten romanischen und germanischen Sprachen*. In S. Heinemann, G. Bernhard & D. Kattenbusch (Hrsg.), *Roma et Romania. Festschrift für Gerhard Ernst zum 65. Geburtstag*, Seiten 151–167. Niemeyer, Tübingen, 2002.

[Holl 02b] Alfred Holl & Thomas Krach. *Ubiquitäre IT - ubiquitärer naiver Realismus?* In Bernd Britzelmaier (Hrsg.), *Der Mensch im Netz - Ubiquitous Computing: 4. Liechtensteinisches Wirtschaftsinformatik-Symposium an der Fachhochschule Liechtenstein*, Seiten 53–69. Teubner, Stuttgart, 2002.

[IEEE1058 98] *IEEE Standard for Software Project Management Plans*. Nummer IEEE Std 1058-1998. 1998.

[IEEE1074 06] *IEEE Standard for Developing a Software Project Life Cycle Process*. Nummer IEEE Std 1074-2006. 2006.

[IEEE12207.0 96] *Standard Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology - Software Life Cycle Processes*. Nummer IEEE Std 12207.0-1996. 1996.

[IEEE12207.1 97] *Standard Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology - Software Life Cycle Processes - Life Cycle Data*. Nummer IEEE Std 12207.1-1997. 1997.

[IEEE12207.2 97] *Standard Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology - Software Life Cycle Processes - Implementation Considerations*. Nummer IEEE Std 12207.2-1997. 1997.

[IEEE1517 99] *IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes*. Nummer IEEE Std 1517-1999. 1999.

[IEEE15288 04] *Adoption of ISO/IEC 15288:2002 Systems Engineering - System Life Cycle Processes*. Nummer IEEE Std 15288-2004. 2004.

[IEEE610.12 90] *IEEE Standard Glossary of Software Engineering Terminology*. Nummer IEEE Std 610.12-1990. 1990.

[IEEE828 05] *IEEE Standard for Software Configuration Management Plans*. Nummer IEEE Std 828-2005. 2005.

[ISO10746-1 98] *Information technology - Open Distributed Processing - Reference Model: Overview*. Nummer ISO/IEC 10746-1:1998(E). 1998.

[ISO10746-2 96] *Information technology - Open Distributed Processing - Reference Model: Foundations*. Nummer ISO/IEC 10746-2:1996(E). 1996.

[ISO12207-0A1 02] *Information Technology - Software Life Cycle Processes Amendement 1*. Nummer ISO/IEC 12207.0:1995/Amd.1:2002. 2002.

[ISO12207-0A2 04] *Information Technology - Software Life Cycle Processes Amendement 2*. Nummer ISO/IEC 12207.0:1995/Amd.2:2004. 2004.

[ISO14764 06] *Software Engineering - Software Life Cycle Processes - Maintenance*. Nummer ISO/IEC 14764:2006(E). 2006.

[Johnson 01] Steven Johnson. *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. Penguin Books, London, 2001.

[Johnson 03] J. N. Johnson & P. F. Dubois. *Issue Tracking*. Computing in Science & Engineering [see also IEEE Computational Science and Engineering], Band 5, Nr. 6, Seiten 71–77, 2003.

[Jones 98] Capers T. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[Karlsson 95] Even-André Karlsson (Hrsg.). *Software Reuse: A Holistic Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Kitchenham 99] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen & Hongji Yang. *Towards an Ontology of Software Maintenance*. Journal of Software Maintenance, Band 11, Nr. 6, Seiten 365–389, 1999.

[Kleppe 03] Anneke G. Kleppe, Jos Warmer & Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Klink 07] Martin Klink. *Werkzeug zur Informationsverwaltung in der dezentralen Systementwicklung*. Diplomarbeit, Technische Universität Berlin, 2007.

[Koch 04] Stefan Koch. *Free/Open Source Software Development*. Idea Group Publishing, Hershey, PA, USA, 2004.

[Koru 04] Gunes A. Koru & Jeff Tian. *Defect Handling in Medium and Large Open Source Projects*. IEEE Softw., Band 21, Nr. 4, Seiten 54–61, 2004.

[Koskine 04] Jussi Koskine. *Software Maintenance Costs*. Forschungsbericht, Information Technology Research Institute, University of Jyväskylä, Finnland, 2004.

[Krueger 92] Charles W. Krueger. *Software Reuse*. ACM Comput. Surv., Band 24, Nr. 2, Seiten 131–183, 1992.

[Kuhn 93] Thomas Kuhn. *Die Struktur wissenschaftlicher Revolutionen*. Suhrkamp-Taschenbuch Wissenschaft ; 25. Verlag Suhrkamp, Frankfurt am Main, 12. Auflage, 1993.

[Lutterbeck 04] Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.). *Open Source Jahrbuch 2004: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2004. Lehmanns Media – LOB.de.

[Lutterbeck 05] Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.). *Open Source Jahrbuch 2005: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2005. Lehmanns Media – LOB.de.

[Lutterbeck 06] Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.). *Open Source Jahrbuch 2006: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2006. Lehmanns Media – LOB.de.

[Lutterbeck 07] Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.). *Open Source Jahrbuch 2007: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2007. Lehmanns Media – LOB.de.

[Lutterbeck 08] Bernd Lutterbeck, Matthias Bärwolff & Robert A. Gehring (Hrsg.). *Open Source Jahrbuch 2008: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, March 2008. Lehmanns Media – LOB.de.

[Magnusson 98] Boris Magnusson (Hrsg.). *System Configuration Management, ECOOP'98 SCM-8 Symposium, Brussels, Belgium, July 20-21, 1998, Proceedings*, Band 1439 von *Lecture Notes in Computer Science*. Springer, 1998.

[Mahr 97] Bernd Mahr. *Gegenstand und Kontext - Eine Theorie der Auffassung*. In Bernd Mahr, Klaus Eyferth, Roland Posner & Fritz Wysotzki (Hrsg.), *Prinzipien der Kontextualisierung - KIT Report 141*. Technische Universität Berlin, 1997.

[Mahr 03] Bernd Mahr. *Modellieren. Beobachtungen und Gedanken zur Geschichte des Modellbegriffs*. In Sybille Krämer & Horst Bredekamp (Hrsg.), *Bild-Schrift-Zahl*, Seiten 59–86, München, 2003. Wilhelm Fink Verlag.

[Mahr 04] Bernd Mahr. *Das Wissen im Modell*. KIT Report 150, Technische Universität Berlin, 2004.

[Mahr 06] Bernd Mahr. *Ein Modell der Auffassung*. KIT Report 151, Technische Universität Berlin, 2006.

[Mahr 08] Bernd Mahr. *Ein Modell des Modellseins - Ein Beitrag zur Klärung des Modellbegriffs*. In Ulrich Dirks & Eberhard Knobloch (Hrsg.), *Modelle*, Seiten 178–218, Frankfurt, 2008. Peter Lang Verlag.

[Malone 94] Thomas W. Malone & Kevin Crowston. *The Interdisciplinary Study of Coordination*. ACM Computing Surveys, Band 26, Nr. 1, Seiten 87–119, 1994.

[McIlroy 68] M. D. McIlroy. *Mass Produced Software Componentents*. In Peter Naur & Brian Randell (Hrsg.), *Software Engineering - Report on a conference sponsored by the NATO science commitee*, Seiten 138–151, 1968.

[Meredith 00] Jack R. Meredith & Samuael J. Mantel. *Project Management: A Managerial Approach*. John Wile & Sons, Inc., New York, 4. Auflage, 2000.

[Muffatto 03] Moreno Muffatto & Matteo Faldani. *Open Source as a Complex Adaptive System*. In *Emergence: Complexity and Organization*, Band 5(3), Seiten 83–100, 2003.

[OSI 07] *Open Source Initiative Website*, 2007. Abgerufen von http://www.opensource.org am 22.04.2007.

[Ottino 04] J.M. Ottino. *Engineering Complex Systems*. Nature, Band 427, Seite 399, January 2004.

[Overhage 06] Sven Overhage. *Vereinheitlichte Spezifikation von Komponenten : Grundlagen, UnSCom Spezifikationsrahmen und Anwendung*. Ph.D. dissertation, University of Augsburg, June 2006.

[Patzak 04] Gerold Patzak & Günter Rattay. *Projektmanagement*. Linde Verlag, Wien, 4. Auflage, 2004.

[Pavlicek 00] Russell C. Pavlicek. *Embracing Insanity: Open Source Software Development*. Sams, Indianapolis, IN, 2000.

[PMBOK 00] *A Guide to the Project Management Body of Knowledge*. Project Management Institute, Newtown Square, 2000.

[Rajlich 00] Vaclav T. Rajlich & Keith H. Bennett. *A Staged Model for the Software Life Cycle*. Computer, Band 33, Nr. 7, Seiten 66–71, 2000.

[Raymond 99] Eric S. Raymond. *The Cathedral & the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates Inc., Sebastopol, 1999.

[Ricci 05] Alessandro Ricci, Mirko Viroli & Andrea Omicini. *Programming MAS with Artifacts*. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix & Amal El Fallah-Seghrouchni (Hrsg.), *PROMAS*, Band 3862 von *Lecture Notes in Computer Science*, Seiten 206–221. Springer, 2005.

[Ricci 06] Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli & Enrico Oliva. *Cognitive Stigmergy: Towards a Framework Based on Agents and Artifacts*. In *Environments for Multi-Agent Systems III*, Seiten 124–140, 2006.

[Riedl 81] Rupert Riedl. *Biologie der Erkenntnis - Die stammesgeschichtliche Grundlage der Vernunft*. Verlag Paul Parey, Berlin, 3. Auflage, 1981.

[Riedl 90] Rupert Riedl. *Die Ordnung des Lebendigen - Systembedingungen der Evolution*. Piper Verlag, München, 1990.

[Robles 05] G. Robles, J. J. Merelo & J. M. Gonzalez-Barahona. *Self-Organized Development in Libre Software: A Model Based on the Stigmergy Concept*. In *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling*, St.Louis, Missouri, USA, May 2005.

[Robles 06] Gregorio Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. Ph.D. dissertation, Universidad Rey Juan Carlos, Madrid, 2006.

[Rupp 07] Chris Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. Carl Hanser Verlag, Wien, 4. Auflage, 2007.

[Sametinger 97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, New York, NY, USA, 1997.

[Schneidewind 99] Norman Schneidewind, Barbara Kitchenham, Frank Niessink, Janice Singer, Anneliese von Mayrhauser & Hongji Yang. *Resolved: Software Maintenance is Nothing More than Another Form of Development*. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, Seite 63, Washington, DC, USA, 1999. IEEE Computer Society.

[SCM 05] *Proceedings of the 12th International Workshop on Software Configuration Management, SCM 2005, Lisbon, Portugal, September 5-6, 2005*. ACM, 2005.

[Sindre 95] Guttorm Sindre, Reidar Conradi & Even-André Karlsson. *The REBOOT Approach to Software Reuse*. J. Syst. Softw., Band 30, Nr. 3, Seiten 201–212, 1995.

[Staehle 99] Wolfgang H. Staehle. *Management. Eine verhaltenswissenschaftliche Perspektive*. Vahlen, München, 8. Auflage, 1999.

[Stephan 99] Achim Stephan. *Emergenz: Von der Unvorhersehbarkeit zur Selbstorganisation*. Dresden University Press, Dresden, 1999.

[Suhr 07] Gregor Suhr. *Werkzeug zur Aufgabenverwaltung in der dezentralen Systementwicklung*. Diplomarbeit, Technische Universität Berlin, 2007.

[Susi 01] Tarja Susi & Tom Ziemke. *Social Cognition, Artefacts, and Stigmergy: A Comparative Analysis of Theoretical Frameworks for the Understanding of Artefact-mediated Collaborative Activity*. Cognitive Systems Research, Band 2, Nr. 4, Seiten 273–290, December 2001.

[Tolzmann 07] Marius Tolzmann. *Hosting Plattform für die dezentrale Systementwicklung*. Diplomarbeit, Technische Universität Berlin, 2007.

[van Aardt 04] Albert van Aardt. *Open Source Software Development as a Complex Adaptive System: Survival of the Fittest?* In Samuel Mann & Tony Clear (Hrsg.), *Conference Proceedings of the 17th Annual NACCQ*, 2004.

[Vogel 05] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, U. Mehlig, T. Neumann, M. Voelter, & U. Zdun. *Software-Architektur: Grundlagen, Konzepte, Praxis*. Spektrum Akademischer Verlag, Berlin, October 2005.

[Vollmer 90] Gerhard Vollmer. *Evolutionäre Erkenntnistheorie*. Hirzel, Stuttgart, 1990.

[von Hayek 80] Friedrich A. von Hayek. *Recht, Gesetzgebung und Freiheit: Regeln und Ordnung*. Verlag Moderne Industrie, München, 1980.

[Waldrop 92] Mitchell M. Waldrop. *Complexity: The Emerging Science at the Edge of Order and Chaos*. Simon & Schuster, 1992.

[Weber 04] Steven Weber. *The Success of Open Source*. Harvard University Press, April 2004.

[Westfechtel 03] Bernhard Westfechtel & André van der Hoek (Hrsg.). *Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, May 14-15, 2001 and Portland, OR, USA, May 9-10, 2003. Selected Papers*, Band 2649 von *Lecture Notes in Computer Science*. Springer, 2003.

[Wilkins 95] John S. Wilkins. *Evolutionary Models of Scientific Theory Change*. Master's thesis, Department of Philosophy, Monash University, 3 Peel Grove, Mount Martha 3934, Australia, 1995.

[Woolridge 01] Michael Woolridge & Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.