Dissertation
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Komplexität von Softwarearchitekturen

- Stile und Strategien -

Carola Lilienthal

Juli 2008

Betreut von:

Prof. Dr. Christiane Floyd, Universität Hamburg

Erstgutachterin: Prof. Dr. Christiane Floyd
Zweitgutachter: Prof. Dr. Claus Lewerentz
Drittgutachter: Dr. Daniel Moldt

Tag der mündlichen Prüfung: 23.01.2008

Danksagung

Prof. Dr. Christiane Floyd danke ich für ihre jahrelange Begleitung auf meinem Weg als Wissenschaftlerin. Sie hat bereits 1995 meine Diplomarbeit betreut und hat mich während meiner Berufstätigkeit in der Wirtschaft immer wieder daran erinnert, dass ich eigentlich noch eine Doktorarbeit schreiben wollte.
Liebe Christiane, meine Dissertation hätte ich ohne Dich nicht schreiben können. Deine Begabung Dich tatsächlich auf den Menschen einzulassen, der vor Dir sitzt und um Rat fragt, ist legendär – ich habe unglaublich davon profitiert. Wenn ich zwar wusste, was ich sagen wollte, aber den richtigen Begriff nicht finden konnte, brauchte ich nur in Dein Büro zu gehen. Ich konnte sicher sein, dass Du Zeit hattest und ich mit dem bestmöglichen Namen für meine Idee wieder herauskommen würde. Dass es doch noch besser ging, erfuhr ich dann oft am nächsten Tag, wenn Du mit dem Satz in mein Zimmer kam: „Ich habe noch ein bisschen darüber nachgedacht und dabei ist mir klar geworden…".
Du wirst mir immer ein Vorbild sein, an dem ich mich ausrichten kann.

Prof. Dr. Claus Lewerentz danke ich dafür, dass er das Zweitgutachten für meine Arbeit übernommen hat. Er hat vor 2 ½ Jahren eine sehr frühe Fassung des 10ten Kapitels gelesen und seine Anmerkungen haben mich mit vielen Ideen für die weitere Arbeit versorgt. Immer wieder hat er mir über Skype wertvolle Kritik zu meinen neu entstandenen Kapiteln gegeben und mich mit Rückfragen zum Nachdenken gebracht. Lieber Claus, vielen Dank für Dein Engagement als Zweitbetreuer.

Dr. Daniel Moldt hat mir mit seinem Feedback im Drittgutachten viele Hinweise zur Verbesserung meiner Arbeit gegeben. Lieber Daniel, vielen Dank für Dein Interesse an softwaretechnischen Themen und Deinen Zuspruch.

Prof. Dr.-Ing. Heinz Züllighoven und Dr. Guido Gryczan kenne ich eben so lange wie Christiane Floyd. Sie haben meinen Weg als Studentin begleitet und ich hatte das Glück später in ihrer Firma der C1 WPS zu arbeiten. Ein Großteil der Erkenntnisse, die ich in dieser Dissertation erarbeitet habe, wäre ohne meine Tätigkeit bei der C1 WPS nicht möglich gewesen. Hier habe ich die Kontakte knüpfen können, die mir zu der großen Anzahl an Fallstudien verholfen haben. Als ich schließlich eine längere Zeit der Ruhe und Abgeschiedenheit brauchte, war es kein Problem, mein Engagement in der Firma zurückzuschrauben. Lieber Heinz, lieber Guido, vielen Dank für Eure Unterstützung in schweren Zeiten und Eurer Verständnis für meinen Wunsch, doch noch eine Doktorarbeit schreiben zu wollen.

Außerdem danke ich den Firmen, bei denen ich Fallstudien durchführen durfte: Techniker Krankenkasse, Hamburg; Deutscher Ring LV KV AG, Hamburg; das NetWeaver-Team bei SAP, Waldorf; d+s solutions, Itzehoe und einige weitere Firmen, die namentlich nicht genannt werden wollen. Ohne die Firma Software-Tomography GmbH wäre die gesamte Arbeit in ihrer Tiefe

nicht möglich gewesen. Bischi und Thomas: vielen, vielen Dank für den Sotographen und Eure Hilfe, wenn ich mal wieder etwas sehr Spezielles im Sotographen auswerten wollte!

Ganz besonders danke ich meinen Kollegen Dr. Wolf-Gideon Bleek, Dr. Axel Schmolitzky, Petra Becker-Pechau, Joachim Sauer und Dr. Hartmut Obendorf. Lieber Wolf-Gideon, ohne Dein beharrliches Nachfragen, ob ich nicht doch noch promovieren wolle, wäre ich sicher nie zurück an die Universität gekommen. Du hast alles gelesen, was ich Dir an Unfertigkeiten gegeben habe, und hast Dich zum Schluss nicht gescheut, fertige Kapitel einzufordern. Herzlichen Dank für all die vielen Stunden Deiner Zeit!
Lieber Axel, Du warst es, der mir vor 3 ½ Jahren mein Thema gegeben hat. Ich kann mich noch sehr gut erinnern, wie Du über Deine Komplexitätsstufen für die Vermittlung objektorientierter Programmiersprachen erzähltest und ich auf einmal wusste: Komplexität von Architekturen, das ist mein Thema! Ohne Dich hätte mein Interesse an Architektur sicher nicht so schnell einen prägnanten Namen bekommen.
Liebe Petra, mit Dir habe ich so viele Wochenenden beim „Dissen" verbracht und wir haben so viele Diskussionen miteinander geführt, dass ich jetzt schon traurig werde, wenn ich daran denke, dass ich „unser" gemeinsames Büro bald verlassen werde! Ich hoffe, wir werden noch eine Weile dort zusammenarbeiten können. Ganz, ganz viel Kraft wünsche ich Dir für die nächste Zeit. Ich werde versuchen, Dir eine ähnlich gute Unterstützung beim Dissertationschreiben zu sein, wie Du es für mich warst.
Lieber Joachim, Du hast ein Zimmer weiter ähnliche Kämpfe mit Deinem Text durchgemacht wie ich. Oft haben wir uns bemitleidet, gemeinsam Cappuccino getrunken und sind kickern gegangen, um uns ein bisschen abzulenken. Ich drücke Dir ganz schwer die Daumen, dass Du bald fertig wirst!
Lieber Hartmut, Du hast gegen Ende mehrere „fertige" Kapitel gelesen und mir besonders bei den Übergängen zwischen den Kapiteln geholfen. Vielen Dank für Dein Feedback trotz Deiner knappen Zeit!

Zu Schluss danke ich meiner Familie, ohne die dieses ganze Unterfangen nicht möglich gewesen wäre. Mein Sohn Frederic hat in den letzten 1 ½ Jahren oft auf mich verzichtet. Den Satz „Das machen wir dann nach Deiner Diss!" ist ihm schon so in Fleisch und Blut übergegangen, dass er, an einem freien Samstag, ziemlich erschreckt zu mir sagte: „Aber Mami, an so einem Tag musst Du doch an Deiner Diss arbeiten!". Lieber Frederic, Du machst mein Leben zu einem wundervollen Abenteuer. Jetzt können wir es genießen!

Liebe Mama, lieber Papa, Ihr habt Euch unermüdlich eingesetzt und Euch um Frederic, meine Einkäufe und mein rebellierendes Auto gekümmert. Ohne Euch wäre ich niemals so weit gekommen, wie ich heute bin. Ihr habt Frederic vom Kindergarten und später von der Schule abgeholt. Habt uns am Wochenende aufgenommen und uns mit Pfannekuchen versorgt. Schließlich habt Ihr auch noch die ganze Arbeit korrigiert. Vielen Dank für einfach Alles!

Zusammenfassung

In dieser Arbeit untersuche ich, wo Komplexität bei der Softwareentwicklung auftritt, und konzentriere mich auf Architekturkomplexität, die in der statischen Struktur von Softwaresystemen, der Softwarearchitektur, enthalten ist. In der Softwaretechnik existiert bereits ein umfassendes Repertoire an Architekturkonzepten und Architekturstilen, um die Komplexität von Softwarearchitekturen zu lindern. Trotzdem scheitern viele Projekte, und Entwicklungsteams beklagen sich darüber, dass ihr Softwaresystem nicht mehr wartbar und weiterentwickelbar ist.

Um Architekturkomplexität greifbar zu machen, ziehe ich Grundsätze aus der kognitiven Psychologie heran, die beschreiben, wie Menschen mit komplexen Strukturen umgehen. Dabei wird deutlich, wie objektorientierte Programmiersprachen und Architekturstile diese Grundsätze berücksichtigen und wo Lücken zu erkennen sind.

Auf dieser Grundlage habe ich vierundzwanzig Fallstudien an Softwaresystemen in Industrie und Wissenschaft durchgeführt. Bei zweiundzwanzig Fallstudien konnte ich die Softwaresysteme mit dem Analysewerkzeug Sotograph untersuchen und die Architektur mit dem jeweiligen Entwicklungsteam diskutieren. In einigen Fällen habe ich zusätzlich ein Interview durchgeführt, um weitere Fragen zu klären. Die Ergebnisse aus den Fallstudien zeigen das weite Spektrum der heute in Softwarearchitekturen vorhandenen Komplexität und lassen ihre Ursachen sichtbar werden. Dabei wird deutlich, dass der Architekturstil einen entscheidenden Einfluss auf die Komplexität einer Softwarearchitektur hat.

Parallel zu den Fallstudien habe ich ein Modell für Architekturkomplexität entwickelt. Die drei Faktoren, auf die ich Architekturkomplexität zurückführe, sind: Mustertreue, Modularität und Geordnetheit. Aus diesen drei Faktoren leite ich Kriterien und Fragen ab. Für sechs Fragen definiere ich Maße, die eine quantitative Auswertung erlauben. Anhand des Modells für Architekturkomplexität werden die Ergebnisse aus den Architekturanalysen und aus den Interviews präsentiert und interpretiert.

Um die Ergebnisse konstruktiv nutzbar zu machen, stelle ich drei Stadien der architekturzentrierten Softwareentwicklung vor: Entwerfen, Erhalten und Erneuern von Architektur. Diese Stadien verlangen nach gut abgestimmten Strategien, wie Entwicklungsteams Architekturkomplexität reduzieren können. Schließlich biete ich einen Leitfaden dafür an, wie die Strategien in den drei Stadien der architekturzentrierten Softwareentwicklung geplant und eingesetzt werden sollen, so dass Entwicklungsteams abhängig von ihrer Situation die passende Strategie auswählen können.

Abstract

This work examines where complexity occurs from in software development and concentrates on architectural complexity that resides in the static structure of a software system: its software architecture. Software engineering offers an extensive repertoire of architectural concepts and styles to mitigate complexity in software architecture. However, many projects fail and development teams complain about non-maintainable software.

To substantiate architectural complexity, general principles of cognitive psychology are applied that describe how people deal with complex structures. Object-oriented programming and architectural styles follow these principles to some extent but they still leave gaps.

On this basis twenty-four case studies were conducted in industrial and university contexts. Twenty-two software systems were analysed with the tool Sotograph and their architecture was discussed with the architects. In some cases interviews were carried out to clarify various points. The results show a wide spectrum of complexity in software architecture and unveil their origins. Architectural styles for instance have a crucial effect on architectural complexity.

At the same time a model of architectural complexity has been developed. Architectural complexity is ascribed to three factors: pattern conformity, modularity and ordering. Criteria and questions are derived from these factors. Some questions are substantiated with metrics to perform measurements. Following the model of architectural complexity, the results of the architecture analysis and interviews are presented and interpreted.

To harness these results, three stages in architecture-centric development are presented: designing, preserving and restoring the architecture. These stages demand well adjusted strategies as to how development teams should deal with architectural complexity. Finally a guide connects strategies and stages in such a way that development teams will be able to choose the appropriate strategy for their unique situation.

Schreibweisen

Wenn ich in dieser Arbeit über Menschen in bestimmten Rollen spreche, benutze ich die weibliche und die männliche Form, z. B. „die Entwicklerinnen und Entwickler“ oder „die Architektinnen und Architekten“. In den Fällen, in denen ich von einem Gespräch mit einer bestimmten Person berichte, habe ich jeweils die passende Form gewählt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Softwaresysteme gehören zu den komplexesten Konstrukten, die Menschen erdacht und erbaut haben. Insofern ist es nicht verwunderlich, dass Softwareprojekte scheitern und Altsysteme aus Angst, dass sie ihren Dienst einstellen, nicht mehr angerührt werden. Trotzdem bin ich im Rahmen meines Dissertationsvorhabens und meiner Beratungstätigkeit immer wieder mit Projektteams in Kontakt gekommen, die ihr Softwaresystem unabhängig von seinem Anwendungsgebiet, seiner Größe und seinem Alter für wartbar und erweiterbar hielten.

Die Frage nach den Ursachen für das Gelingen oder Scheitern von Softwareentwicklung lässt sich auf vielen Ebenen beantworten: dem Anwendungsgebiet und der involvierten Organisation, der eingesetzten Technologie, der Qualität des zu wartenden Softwaresystems oder auch der Qualifikation der Mitarbeiterinnen und Mitarbeiter. Für diese Arbeit habe ich den Fokus auf das Thema *Softwarearchitektur* gelegt. Auch in der Softwaretechnik hat dieses Thema in den letzten Jahren mehr und mehr Beachtung gefunden, was unter anderem zum Entstehen verschiedener Konferenzen (z. B. Working IEEE/IFIP Conference on Software Architecture (WICSA) in 1999, International Conference on Quality of Software Architecture (QoSA) in 2005) und der GI-Fachgruppe Architektur im Jahr 2006 geführt hat.

Im Jahr 2000 wurde *Softwarearchitektur* im ANSI/IEEE-Standard-1471 grundlegend definiert. Unter einer Softwarearchitektur versteht man eine oder mehrere Strukturen eines Softwaresystems, die die Elemente, ihre extern sichtbaren Eigenschaften und ihre Beziehungen beinhalten (ANSI/IEEE-Standard-1471 2000). Diese sehr allgemeine Definition erlaubt es, verschiedene Aspekte von Softwaresystemen zu betrachten: die statische Struktur, das dynamische Laufzeitverhalten, die Verteilung der Laufzeitkomponenten auf Hardwarekomponenten oder auch die Einbettung eines oder mehrerer Softwaresysteme in die organisatorischen Prozesse eines Unternehmens.

Im Zentrum meiner Arbeit steht die *statische Struktur* von Softwaresystemen, das heißt, ihr Aufbau aus Klassen, Subsystemen und ihren Beziehungen untereinander. Wie kann diese Struktur gestaltet werden, so dass ein Softwaresystem dauerhaft beherrschbar und damit erweiterbar bleibt? Diese Frage ist deshalb so spannend, weil Softwaresysteme nach verschiedenen *Entwurfsprinzipien* und *Architekturstilen* entwickelt werden. Die Softwarearchitekturen, die dabei entstehen, unterscheiden sich in wesentlichen Punkten; und als Forscherin und Beraterin wurde ich immer wieder mit der Frage konfrontiert: Was ist die bessere Lösung? Gibt es überhaupt eine Lösung, die besser als die bisherige ist?

Um eine Antwort auf diese Frage zu finden, habe ich als zweiten Ausgangspunkt dieser Arbeit *Komplexität* gewählt. Entwicklerinnen und Entwickler sind mit Komplexität konfrontiert, wenn sie ein Softwaresystem entwerfen, warten und weiterentwickeln. Ein Teil dieser Komplexität betrifft die Softwarearchitektur und kann dazu führen, dass das Softwaresystem unbeherrschbar wird.

Um diese Komplexität greifbar zu machen, ziehe ich Grundsätze aus der kognitiven Psychologie heran, die beschreiben, wie Menschen mit komplexen Strukturen umgehen. Auf dieser Basis definiere ich Architekturkomplexität und arbeite ihre verschiedenen Aspekte in einem *Modell für Architekturkomplexität* aus.

Mithilfe meines *Modells für Architekturkomplexität* lässt sich beschreiben, wie sich ein guter Softwareentwurf in der Struktur des Softwaresystems auswirken sollte. Viel interessanter ist aber die Frage, ob sich dieser Effekt im implementierten Softwaresystem tatsächlich wiederfinden lässt. Hat der wohldurchdachte Softwareentwurf zu geringer Architekturkomplexität geführt oder stehen die Entwicklerinnen und Entwickler vor denselben Problemen wie bisher, wenn sie das Softwaresystem erweitern sollen. Zum Problem wird Architekturkomplexität für ein Entwicklungsteam nämlich erst im implementierten Softwaresystem und nicht in den vorher auf Papier entworfenen Plänen.

Um Architekturkomplexität in Softwaresystemen zu untersuchen, habe ich *vierundzwanzig Fallstudien* durchgeführt. Dabei habe ich die Architektur der Softwaresysteme mit dem Werkzeug Sotograph analysiert und die Ergebnisse mit den jeweiligen Architektinnen und Architekten diskutiert und bewertet. Anhand des Modells für Architekturkomplexität konnte ich die vierundzwanzig Softwaresysteme sowohl quantitativ, durch Messungen im Programmtext, als auch qualitativ, auf der Basis der Diskussionen und einiger Interviews, vergleichen. Die Ergebnisse aus den Fallstudien zeigen das weite Spektrum der heute in Softwarearchitekturen vorhandenen Komplexität und lassen ihre Ursachen sichtbar werden.

Sind die Ursachen für Architekturkomplexität erkannt, so können die Ergebnisse konstruktiv nutzbar gemacht und Vorgehensweisen zur Reduktion von Architekturkomplexität herausgearbeitet werden. Die Anfang der 1990er Jahre entwickelte Methode Unified Process (UP) hatte sich als ein Merkmal „architekturzentriert“ auf die Fahnen geschrieben (Jacobson et al. 1999). Schon damals ging man unter diesem Begriff davon aus, dass die Softwarearchitektur im gesamten Entwicklungsprozess kontinuierlich weiterentwickelt wird. In dieser Arbeit gehe ich einen Schritt weiter und führe drei *Stadien* für architekturzentrierte Softwareentwicklung ein, die nach unterschiedlichen Vorgehensweisen im Zusammenhang mit Softwarearchitektur verlangen. Ausgehend von diesen Stadien erarbeite ich vier *Strategien*, die Entwicklungsteams anleiten, wie sie Architekturkomplexität beherrschbar machen können. Die Ausarbeitung der Stadien und Strategien runde ich schließlich durch einen *Leitfaden* ab, der es Entwicklungsteams ermöglicht, eine für ihre Situation geeignete Strategie zur Verminderung von Architekturkomplexität zu bestimmen.

1.1. Zielsetzung und Fragestellung

Diese Arbeit hat zwei Ziele:

Auf der einen Seite will ich ein Modell für Architekturkomplexität entwickeln, das sich auf Softwaresysteme als *Produkt* bezieht. Mit diesem Modell können Softwarearchitekturen bewertet und Fragen zu Architekturkomplexität eingeordnet werden.

Auf der anderen Seite will ich architekturzentrierte Softwareentwicklung als *Prozess* beschreiben, der, abhängig von der jeweiligen Situation, den Einsatz von unterschiedlichen Strategien verlangt. Andere Methoden und Strategien zur architekturzentrierten Softwareentwicklung lassen sich auf dieser Basis beurteilen.

Aus dieser Zielsetzung und der Einleitung lassen sich vier Fragen entnehmen, die dieser Arbeit zugrunde liegen:

1. Was ist Architekturkomplexität und wie kann man sie bestimmen?
2. Welche empirischen Erkenntnisse lassen sich über Architekturkomplexität erlangen und was sind ihre Ursachen?
3. Was ist architekturzentrierte Softwareentwicklung?
4. Welche Strategien helfen, Architekturkomplexität zu reduzieren, und nach welchen Kriterien können Entwicklungsteams geeignete Strategien auswählen?

Sowohl das Modell für Architekturkomplexität als auch die architekturzentrierte Sichtweise sind durch die in den Fallstudien vorgenommenen Ergebnisse empirisch abgesichert.

1.2. Aufbau der Arbeit

Im zweiten Kapitel dieser Arbeit untersuche ich, wo Komplexität bei der Softwareentwicklung angesiedelt ist. Drei verschiedene Ebenen der Komplexität und der für Softwarearchitekturen wichtige Anteil von Softwarekomplexität werden herausgearbeitet.

Das dritte Kapitel fasst den aktuellen Stand der Forschung im Bereich Softwarearchitektur zusammen und beschreibt die für diese Arbeit wichtigen Architekturkonzepte und Architekturstile. Das Kapitel schließt mit einer Definition für den Begriff Architekturkomplexität.

Das vierte Kapitel stellt Erkenntnisse aus dem Gebiet der kognitiven Psychologie vor und zeigt, wie Menschen mit komplexen Strukturen umgehen, um sie zu verstehen. Dabei wird deutlich, wie objektorientierte Programmiersprachen und Architekturstile zur Verringerung von Komplexität beitragen und wo Lücken zu erkennen sind.

Im fünften Kapitel beschreibe ich meine Forschungsmethodik und den Ablauf der Fallstudien. Das Vorgehen bei Architekturanalyse und Interviews, das

von mir eingesetzte Analysewerkzeug Sotograph und die Auswertung der Ergebnisse werden erläutert.

Im sechsten Kapitel stelle ich das Modell für Architekturkomplexität vor und argumentiere seinen Aufbau anhand softwaretechnischer Prinzipien. Für Architekturkomplexität werden Faktoren und Kriterien abgeleitet, die schließlich zu Fragestellungen führen, mit denen ich in den Fallstudien gearbeitet habe.

Für sechs Fragestellungen aus dem Modell für Architekturkomplexität definiere ich im siebten Kapitel dreizehn Maße, die eine quantitative Auswertung erlauben. Beispielhaft werden zwei Maße durch eine interne Validierung auf ihre Konformität zur Messtheorie überprüft.

Im achten Kapitel präsentiere ich die Ergebnisse der Untersuchung. Aus den Vermessungen der Softwaresysteme und den in Interviews diskutierten Fragestellungen lässt sich entnehmen, welche Ursachen Architekturkomplexität hat.

Im neunten Kapitel stelle ich architekturzentrierte Softwareentwicklung als Vorgehensweise vor und erweitere sie um drei Stadien, in denen mit unterschiedlicher Architekturkomplexität umgegangen werden muss.

Schließlich werden im zehnten Kapitel Strategien und ein Leitfaden vorgeschlagen. Mit den Strategien fächere ich die Möglichkeiten auf, wie Entwicklungsteams Architekturkomplexität abmildern können. Der Leitfaden ordnet die Strategien den drei Stadien der architekturzentrierten Softwareentwicklung zu, so dass Entwicklungsteams, abhängig von ihrer Situation, die passende Strategie auswählen können.

Das elfte Kapitel liefert eine Zusammenfassung der in dieser Arbeit erzielten Ergebnisse und gibt Anstöße für weitere Forschungsvorhaben.

2. Komplexität in der Softwareentwicklung

Komplexität ist ein schwer zu fassender Begriff, der in vielen wissenschaftlichen Disziplinen untersucht und kontrovers diskutiert wird. Um für diese Arbeit bei der existierenden Vielfalt die begriffliche Grundlage zu legen, gehe ich zu Beginn dieses Kapitels von der Frage aus, wo man sich bei der Softwareentwicklung mit Komplexität auseinander setzen muss. Dabei wird deutlich, dass Komplexität auf verschiedenen Ebenen anzutreffen ist: im Softwaresystem, beim Menschen selbst und in der Interaktion in einem Entwicklungsteam. Auf Basis dieser drei Ebenen ordne ich die in der Literatur vorhandenen Aussagen zu Komplexität ein und beschränke mich auf die für diese Arbeit relevanten Komplexitätsaspekte.

2.1. Ist Software komplex?

In der Literatur wird der Begriff *Softwarekomplexität* vielfach verwendet, es steht aber keine allgemeingültige Definition zur Verfügung. Diese Tatsache mag im ersten Moment erstaunlich sein, einschlägigen Autoren (Henderson-Sellers 1996; Zuse 1990), die dem Thema Softwarekomplexität jeweils ein ganzes Buch gewidmet haben, machen jedoch gleich zu Anfang ihrer Ausführungen klar, dass dieser Begriff nicht definierbar ist. Henderson-Sellers schreibt, dass eine Aussage, wie „complexity can be loosely defined as that characteristic of software that requires effort (resources) to design, understand, or code", das Beste ist, was man als Definition für Softwarekomplexität erreichen kann (Henderson-Sellers 1996, S. 50). Dieser Satz stellt in den Raum, dass Software die Eigenschaft (characteristic) hat, komplex zu sein, und daraus für Menschen verschiedene Schwierigkeiten entstehen. Was aber bedeutet das Wort „komplex"?

Die sprachliche Wurzel des Wortes „komplex" kann uns einen Hinweis auf seine Bedeutung geben. Komplex besteht aus zwei Wortteilen „kom" und „plex". „kom" lässt sich übersetzen als „mit" oder „zusammen"; der Wortteil „plex" hat seine Wurzel im lateinischen Verb plectere = flechten und geht ursprünglich auf das altgriechische Partizip plektós (πλεκτός) zurück (vgl. Kluge 2002). Komplex lässt sich daher übersetzen als „zusammen geflochten" bzw. „verflochten" oder „verwoben". Dieser Wortsinn weist auf ein Gebilde hin, das aus einer Vielzahl von Verbindungen zwischen verschiedenen Elementen aufgebaut ist. Der Gegensatz von Komplexität – Einfachheit – leitet sich in der englischen Übersetzung (simplicity) von dem selben Wortstamm „plex" her, allerdings mit dem entscheidenden Zusatz „sim" - einfach, so dass simplicity für „einmal geflochten" steht (Gell-Mann 1994, S. 66).

2.1.1. Komplexität von Softwaresystemen

Betrachtet man Softwaresysteme ausgehend von dieser Wortbedeutung, so kann man ihnen Komplexität als eine Eigenschaft zuschreiben. Sie bestehen sowohl an ihrer Oberfläche als auch in ihrem Inneren aus einer Vielzahl

einzelner Elemente und mannigfaltiger Beziehungen zwischen diesen Elementen (Buhr 1976, S. 642; Simon 1994, S. 145; Strube et al. 1996, S. 323).

Einige Autoren gehen so weit zu sagen, dass Softwaresysteme komplexer sind als irgendein anderes von Menschen erzeugtes Konstrukt, weil zwei Softwaresysteme nie gleich sind (Booch 2004, s. 10ff; Brooks 1986, S. 1070f; Perry und Wolf 1992, S. 41). Beim Häuserbau oder beim Bau von Computerhardware kann durch Replikation einer relativ kleinen Anzahl von vorgefertigten Elementen ein großes „Bauwerk" errichtet werden. Die Struktur von Softwaresystemen setzt sich dagegen aus beliebigen, von den Entwicklerinnen und Entwicklern neu geschaffenen oder immer wieder angepassten Elementen zusammen, so dass Größe durch das Hinzufügen von neuen Designelementen erreicht wird. Diese Flexibilität führt dazu, dass für die unterschiedlichsten Problemfelder Software gebaut werden kann, hat aber zur Folge, dass in einem Softwaresystem zur Laufzeit eine große Anzahl an Zuständen möglich ist, die schwer zu beherrschen sind.

Unabhängig von dieser Diskussion bleibt festzuhalten, dass die erste Ebene, auf der bei der Softwareentwicklung Komplexität vorkommt, das Softwaresystem selbst ist. Diese Betrachtungsweise deckt sich mit der Art und Weise, wie in der Forschung im Bereich Softwaretechnik der Begriff *Softwarekomplexität* verwendet wird, nämlich als Eigenschaft eines Softwaresystems (vgl. Booch 2004; Darcy 2001; Fenton und Pfleegler 1997; Zimmermann und Nagappan 2006; Zuse 1990). Was sich im Detail hinter Softwarekomplexität verbirgt, wird im weiteren Verlauf dieses Kapitels geklärt (s. Abschnitt 2.4). Vorher führe ich noch zwei weitere Ebenen ein, die in der Literatur im Zusammenhang mit Softwarekomplexität diskutiert, aber oft nicht getrennt betrachtet werden.

2.1.2. Komplexität für Entwicklerinnen und Entwickler

Der am Anfang zitierte Satz von Henderson-Sellers enthält neben der Aussage, dass Software die Eigenschaft hat, komplex zu sein, einen weiteren wichtigen Hinweis: Menschen müssen sich anstrengen, um Software zu entwerfen, zu verstehen und zu implementieren. Henderson-Sellers geht sogar so weit, als besseren Begriff für Softwarekomplexität „cognitve complexity" vorzuschlagen. Zuse zeigt in seinem umfassenden Buch zu Softwaremaßen in dieselbe Richtung, wenn er sagt: „The term software complexity is a misnomer. The true meaning of the term software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs" (Zuse 1990, S. 5). Die beiden Autoren verweisen mit ihren Aussagen auf die zweite Ebene von Komplexität, mit der Entwicklerinnen und Entwickler bei der Erledigung von Aufgaben umgehen müssen (s. Abbildung 2-1).

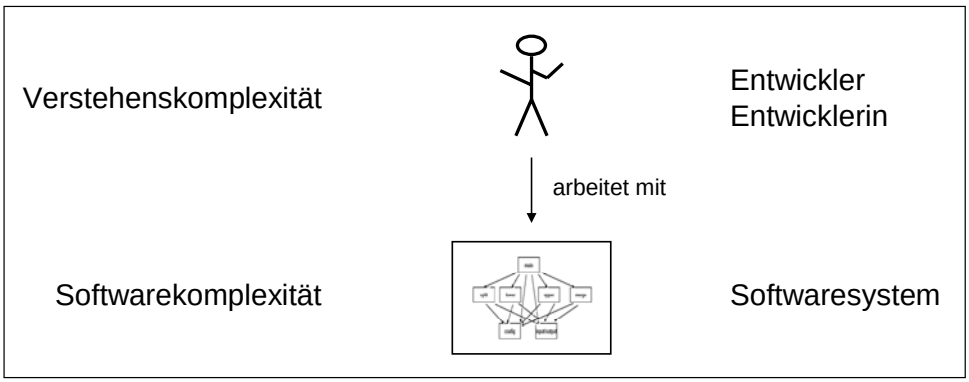

Abbildung 2-1: Zwei Komplexitätsebenen

Zwar können die Beteiligten sich im Regelfall auf Aussagen einigen, wie „eine Versicherungssoftware ist komplexer als ein Schachprogramm", oder „dieses in Assembler geschriebene Programm ist komplexer als jenes in Java implementierte". Das individuelle Empfinden für Komplexität hängt aber von einer Reihe anderer Faktoren ab, z. B. von der persönlichen Erfahrung mit dem Softwaresystem und mit der zu erledigenden Aufgabe. Soll eine Entwicklerin oder ein Entwickler lediglich zwei Felder in der Oberfläche eines Softwaresystems vertauschen, so wird ihr oder ihm diese Aufgabe bei entsprechender Vorbildung in GUI-Programmierung weniger komplex erscheinen als der Austausch der kompletten Datenbankanbindung, wodurch die Basis des gesamten Softwaresystems verändert wird. Diese Ebene wird im Folgenden als *Verstehenskomplexität* bezeichnet und im Abschnitt 2.3 genauer beleuchtet.

2.1.3. Komplexität im Entwicklungsteam

Schließlich lässt sich eine dritte Ebene identifizieren: die Entwicklerinnen und Entwickler, die in einem Team zusammenarbeiten (s. Abbildung 2-2). Auf dieser Ebene ist nicht allein die verflochtene Struktur oder die zu erledigende Aufgabe entscheidend, sondern die Teammitglieder und ihre Interaktion (Booch 2004, S. 10; Brooks 1986, S. 17).

In der soziologischen Systemtheorie wurde von Niklas Luhmann für die Interaktion zwischen Menschen der Begriff *soziales System* geprägt (Luhmann 1987, S. 30). Soziale Systeme haben keine zentrale Instanz, die das gesamte System koordiniert, sondern die Elemente besitzen Eigendynamik, so dass sich die Struktur des Systems durch Selbstorganisation im Laufe der Zeit neu gestaltet und das Verhalten des Systems sich nur aus dem Verhalten der einzelnen Elemente erklärt (Luhmann 1987, S.15f; Waldrop 1992, S. 294ff).

Um diese Komplexität zu beschreiben und sie zu der Komplexität anderer lebendiger Systeme in Beziehung zu setzen, wurde in den letzten Jahrzehnten der Versuch unternommen, eine interdisziplinäre Systemtheorie komplexer Systeme in Physik, Biologie, Technik und den Sozialwissenschaften zu erarbeiten (vgl. Coveney und Highfield 1995; Gell-Mann 1994; Kornwachs 2004; Luhmann 1987).

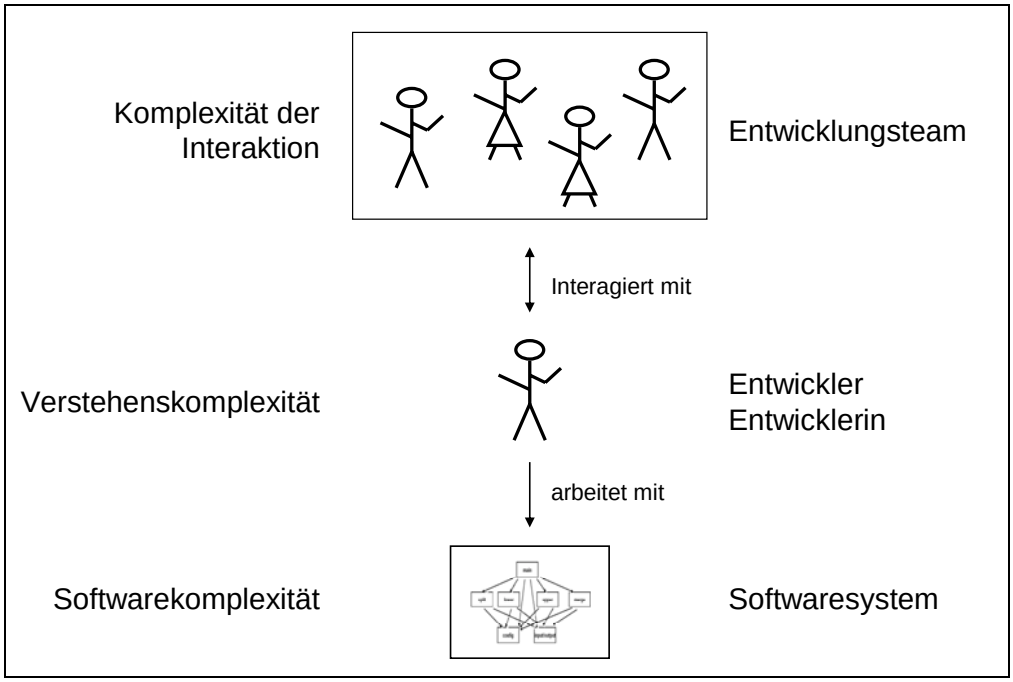

Abbildung 2-2: Drei Komplexitätsebenen

Im Folgenden führe ich das von Luhmann geprägte Verständnis zur Komplexitätsreduktion in sozialen Systemen ein (2.2), untersuche anschließend Verstehenskomplexität (2.3) und bereite so den Weg für die Diskussion der in einem Softwaresystem enthaltenen Komplexität (2.4). Die Darstellung der Theorie sozialer Systeme werde ich vereinfacht vornehmen, da der sehr allgemeine Ansatz der sozialen Systeme und seine vielfältigen Möglichkeiten weit über das inhaltliche Interesse dieser Arbeit hinausgehen.

2.2. Komplexität in sozialen Systemen

Luhmann unterscheidet in seinen Arbeiten grundsätzlich zwischen Maschinensystemen und lebenden Systemen. Maschinensysteme folgen einem von außen gesetzten Zweck. Lebende Systeme organisieren sich selbst und genügen keiner äußeren Zwecksetzung. Weiterhin teilt Luhmann lebende Systeme in Organismen, soziale Systeme und psychische Systeme ein (Luhmann 1987, S. 18). Eine exakte Definition dieser vier verschiedenen Systemarten gibt Luhmann nicht, lässt aber die folgende Bedeutung anklingen: Maschinensysteme sind technische Systeme, die aus Maschinen, Geräten, Computern etc. bestehen. Organismen dagegen sind biologische Systeme, z.B. ein Tier, eine Pflanze oder der menschliche Körper. Der menschliche Körper dient als Träger für den Bewusstseinszusammenhang eines Menschen, der von Luhmann als psychisches System betrachtet wird. Soziale Systeme wiederum bestehen aus Interaktionen zwischen Menschen und stützen sich auf psychischen Systemen ab. (Luhmann 1987, S. 16). Die im letzten Abschnitt identifizierten Ebenen: Softwarekomplexität, Verstehenskomplexität und Komplexität der Interaktion spiegeln die drei für Softwareentwicklung relevanten Systemarten von Luhmann wieder: Maschinensysteme, psychische

und soziale Systeme. Organismen werden im Folgenden nicht weiter betrachtet.

2.2.1. Soziale Systeme und ihre Umwelt

Im Rahmen seiner Theorie sozialer Systeme beschäftigt sich Luhmann mit der Frage, wie ein soziales System sich selbst als Einheit erhält. Denn soziale Systeme haben in Gegensatz zu Maschinen oder physischen Systemen keine sichtbaren Grenzen. Um die Grenzen eines sozialen Systems zu bestimmen, verwendet Luhmann den Begriff der *Umwelt*. Die Umwelt darf nach Luhmann nicht als eine Art Restkategorie missverstanden werden (Luhmann 1987, S. 242f). In der Umwelt eines Systems existieren weitere Systeme, die das System selbst wiederum als ihre Umwelt betrachten. Alles, was vorkommt, ist immer zugleich zugehörig zu einem System (oder zu mehreren Systemen) und zugehörig zur Umwelt anderer Systeme (Luhmann 1987, S. 243).

Wird Softwareentwicklung aus dem Blickwinkel der Theorie sozialer Systeme betrachtet, so lässt sich die Interaktion in einem Entwicklungsteam als soziales System interpretieren. Ein soziales System *Entwicklungsteam* ist in seiner Umwelt mit einer Reihe von anderen sozialen Systemen konfrontiert: den Anwendern des Softwaresystems, der Organisation, die die Anwenderinnen und Anwender beschäftigt und die Software einsetzt, sowie dem Management der beteiligten Unternehmen (Anwendungsorganisation und ggf. externer IT-Dienstleister).

Soziale Systeme entfalten und erhalten sich dadurch, dass sie eine Differenz zu ihrer Umwelt aufbauen, und sie benutzen ihre Grenzen zur Regulierung dieser Differenz. Diesen grundlegenden Mechanismus der Selbsterhaltung übernahm Luhmann in den 80er Jahren unter dem Begriff *Autopoiesis* von Maturana und Varela. Für soziale Systeme legte Luhmann fest, dass sie sich selbst fortwährend erneuern und reproduzieren und damit ihre Autopoiesis aufrechterhalten. Luhmann sagt dazu (Luhmann 1986, S. 269): „Ein soziales System kommt zustande, wenn immer ein autopoietischer Kommunikationszusammenhang entsteht und sich durch Einschränkung der geeigneten Kommunikation gegen eine Umwelt abgrenzt.“ Die Wahrnehmung der Umwelt durch ein soziales System ist laut Luhmann immer selektiv. Ein System kann seine spezifische Wahrnehmungsweise der Umwelt nicht ändern, ohne seine spezifische Identität zu verlieren. Das Ende eines sozialen Systems wird dadurch bestimmt, dass seine Autopoiesis und damit seine Differenz zur Umwelt zusammenbricht (Luhmann 1987, S. 62).

Diese Abgrenzung und die damit verbundene selektive Wahrnehmung lassen sich auf Organisationen übertragen. Entwicklungsteam, Anwendergruppe und Management erhalten sich jeweils als soziale Systeme, indem sie sich voneinander abgrenzen und das jeweils andere System als Umwelt und nicht als Teil des eigenen Systems betrachten. Um diese Kommunikationsbarriere zu überwinden, wird in anwendungsorientierten Vorgehensmodellen in der Softwareentwicklung versucht, die Anwenderinnen und Anwender in ein Projektteam zu integrieren, so dass sie für dieses Projekt Teil des sozialen

Systems Softwareentwicklung werden. Gelingt dieses Unterfangen, so grenzt sich das neue soziale System Projektteam wiederum von seiner Umwelt ab.

Umwelt und System sind für Luhmann eng verbunden, weil sie Wechselwirkungen aufeinander haben (Luhmann 1987, S. 47). Wird ein System geändert, so ändert sich auch die Umwelt anderer Systeme und wächst an einer Stelle die Komplexität, so vergrößert sich die Komplexität der Umwelt für alle anderen Systeme (Luhmann 1987, S. 243). Auch diesen Zusammenhang kann man bei der Softwareentwicklung wieder finden. Verändert sich eine Organisation, indem sie beispielsweise neue Firmen hinzukauft und in die eigene Organisation integriert, so wächst die Komplexität im sozialen System *Anwendungsorganisation*. Das *Entwicklungsteam* als soziales System in der Umwelt der Anwendungsorganisation hat mit hoher Komplexität zu kämpfen. Das Entwicklungsteam muss nun mit verschiedenen Anwendergruppen zusammenarbeiten, die unterschiedliche zu vereinheitlichende Softwaresysteme einsetzen.

2.2.2. Komplexitätsreduktion in sozialen Systemen

Als entscheidenden Ansatzpunkt für die Aufrechterhaltung der Autopoiesis untersucht Luhmann den Umgang des sozialen Systems mit der in ihm selbst und in seiner Umwelt vorhandenen Komplexität (Luhmann 1987, S. 47f). Kann das soziale System die vorhandene Komplexität beherrschen und reduzieren, so ist dies nach Luhmann – neben der Abgrenzung – ein weiterer entscheidender Schritt zur Erhaltung der Autopoiesis.

Die Bewältigung von Komplexität ist dabei keine einmalige Aufgabe, sondern ein soziales System entwickelt sich im ständigen Umgang mit der inneren und äußeren Komplexität kontinuierlich weiter. Kontinuierliche Komplexitätsreduktion führt nicht dazu, dass Komplexität vermieden wird, sondern sie hebt die vorhandene Komplexität lediglich auf und eröffnet so den Zugang zu neuer Komplexität auf einer höheren Ebene (Luhmann 1987, S. 12). Fast definierend sagt Luhmann: „Von Reduktion der Komplexität sollte man dagegen in engerem Sinne immer dann sprechen, wenn das Relationsgefüge eines komplexen Zusammenhanges durch einen zweiten Zusammenhang mit weniger Relationen rekonstruiert wird. Nur Komplexität kann Komplexität reduzieren.“ (Luhmann 1987, S. 49). Luhmann übernimmt hier den Begriff *requisite variety* (notwendige Vielfalt) von W. Ross Ashby, um auszudrücken, dass ein System, das einem anderen System gegenübersteht, um es zu verstehen, selbst keine geringere innere Vielfalt aufweisen darf. Ein System benötigt *requisite variety*, um ein adäquates Beobachten der Umwelt und eine angemessene Verarbeitung der so gewonnenen Informationen zu sichern. Ist das nicht der Fall, so muss das System durch Selektion der Komplexität der Umwelt begegnen (Luhmann 1987, S. 47f)

Entsprechende Mechanismen findet man bei der Arbeit mit Softwaresystemen. Für die Anwenderinnen und Anwender in einer Organisation ist es von entscheidender Bedeutung, dass sie bei ihren Arbeitsaufgaben durch ein zu den Aufgaben passendes Softwaresystem unterstützt werden, das ihnen

uneingeschränkt und ausreichend schnell zur Verfügung steht. Die Anwenderinnen und Anwender können die Komplexität ihrer Arbeit und die damit verbundene Interaktion mit allen Beteiligten nur bewältigen, wenn das Softwaresystem Komplexität reduziert und damit neue Ebenen der Komplexität möglich werden. Ohne das Softwaresystem würden Anwender die ihnen zugedachten Aufgaben nicht bewältigen können, da die Arbeit ohne Softwaresystem beträchtlich länger dauern würde oder auch ganz unmöglich wäre.

2.2.3. Komplexitätsreduktion und Strategien

Der für diese Arbeit interessante Aspekt in Luhmanns Theorie der Komplexitätsreduktion sind Luhmanns Ausführungen zu der Art, wie ein soziales System die Komplexitätsreduktion seiner Umwelt gegenüber vornimmt.

Ein soziales System kann die Komplexität seiner Umwelt nur reduzieren, indem es eine Kapazitätsreserve für den Umgang mit der Komplexität aufbaut. Ist das soziale System vollständig mit Interaktion ausgelastet, so kann es Komplexität nicht reduzieren und wird auf Dauer seine Autopoiesis nicht aufrechterhalten können. Die Kapazitätsreserve wird nach Luhmann verwendet, um eine Planung für den Umgang mit der Komplexität durchzuführen. Als Teil dieser Planung muss ein Modell der Umwelt des sozialen Systems gebildet werden. Das benötigte Modell ist eine vereinfachte Version der Komplexität der Umwelt, die in das System integriert wird (Luhmann 1987, S. 636). Diese vereinfachte Zweitausgabe der Komplexität der Umwelt wird erst durch Planung sichtbar gemacht.

Neben dem Modell der Umwelt benötigt das soziale System als weiteren Teil der Planung ein Handlungsprogramm (Luhmann 1987, S. 432). Handlungsprogramme sind Prozesse, deren einzelne Schritte festgelegt sind. Neben Handlungsprogrammen mit festgelegten Schritten führt Luhmann anpassbare Handlungsprogramme ein, die im Laufe ihrer Ausführung an geänderte Zusammenhänge angepasst werden können. Diese anpassbaren Handlungsprogramme nennt Luhmann *Strategien* (Luhmann 1987, S. 432). Handlungsprogramme und Strategien sind ein bewusstes Vorgehen, das das soziale System im Rahmen seiner Kapazitätsreserve geplant hat.

Luhmanns Verwendung entspricht der Wortbedeutung von Strategie: Das Wort *Strategie* stammen aus dem Griechischen und bedeutete dort ursprünglich Heeresführung (griechisch, Στρατηγική, στρατός = Heer, ἀγω = führen). Ein Stratege war im antiken Griechenland ein gewählter Heerführer. Bei den römischen Geschichtsschreibern wurde der Terminus *Strategia* verwendet, um ein Territorium zu bezeichnen, das sich unter der Kontrolle eines „Strategus" befand. Ihre heutige Bedeutung bekam die Strategie durch den französischen Militärtheoretiker Graf Guibert, der im Jahr 1779 das Wort *Stratégique* verwendete, um die neuartigen Feldzüge Friedrich des Großen zu beschreiben. Heute wird der Begriff nicht nur im militärisch Bereich verwendet, sondern man versteht unter einer Strategie ein längerfristig ausgerichtetes

planvolles Anstreben einer vorteilhaften Lage oder eines Ziels (Oetinger et al. 2005, S. 37f).

Für diese Arbeit festzuhalten bleibt, dass soziale Systeme Komplexität sowohl innerhalb ihres Systems als auch gegenüber der Umwelt reduzieren müssen, um ihr Überleben zu sichern. Komplexitätsreduktion muss dabei kontinuierlich erfolgen, damit immer wieder neue Komplexität zugelassen werden kann. Um Komplexitätsreduktion zu bewerkstelligen, setzen soziale Systeme *Strategien* ein, die ein Modell der Umwelt und eine Planung einzelner Schritte beinhalten. Welche Strategien Entwicklungsteams einsetzen, um die Komplexität ihres Softwaresystems zu reduzieren, werden die nächsten Kapitel zeigen.

2.3. Verstehenskomplexität

Unterhalb der Ebene der sozialen Systeme und der dort vorhandenen Komplexität in der Interaktion befinden sich zwei weitere Ebenen der Komplexität: die Verstehens- und die Softwarekomplexität.

Verstehenskomplexität entsteht beim Menschen, der eine Aufgabe mit bzw. an einem Softwaresystem ausführt (Frese 1987, S. 320ff). Der Mensch empfindet individuell Komplexität, die sich auf seine Aufgabe und das Softwaresystem beziehen kann. Das Zusammenspiel aus Aufgabe, Benutzerin bzw. Benutzer und Computer wird in der Arbeitspsychologie als ein Dreieck dargestellt und als ABC-Modell bezeichnet (Frese und Brodbeck 1989, S. 101; Maaß 1994, S. 6f; Oberquelle 1991, S. 9ff). Das ABC-Modell geht davon aus, dass Benutzerinnen und Benutzer einen Computer verwenden, um eine bestimmte Aufgabe zu erledigen und dabei unterschiedlichen Einflüssen ausgesetzt sind. In der Software-Ergonomie wurden auf Basis des ABC-Modells Kriterien zur Bewertung und Gestaltung ergonomischer Benutzungsschnittstellen erarbeitet (Maaß 1994, S. 6f).

Dieses Modell eignet sich nicht nur für die Diskussion von ergonomischen Benutzungsschnittstellen, sondern auch, um das Verhältnis von Verstehenskomplexität und Softwarekomplexität klarer herauszuarbeiten. Zu diesem Zweck übertrage ich das ABC-Modell aus dem allgemeinen Kontext der Arbeit von Benutzerinnen und Benutzern auf die Arbeit von Entwicklerinnen und Entwicklern: Für sie entsteht Verstehenskomplexität in Bezug auf ein Softwaresystem, wenn sie die Aufgabe bekommen, es (weiter) zu entwickeln. Die sie dabei beeinflussenden Faktoren sind in Abbildung 2-3 dargestellt.

Geht man von diesem Modell aus, so erfahren die Entwicklerin bzw. der Entwickler mehr oder weniger Verstehenskomplexität, abhängig von den Faktoren Fachkompetenz, Erschließbarkeit und Softwarekomplexität.

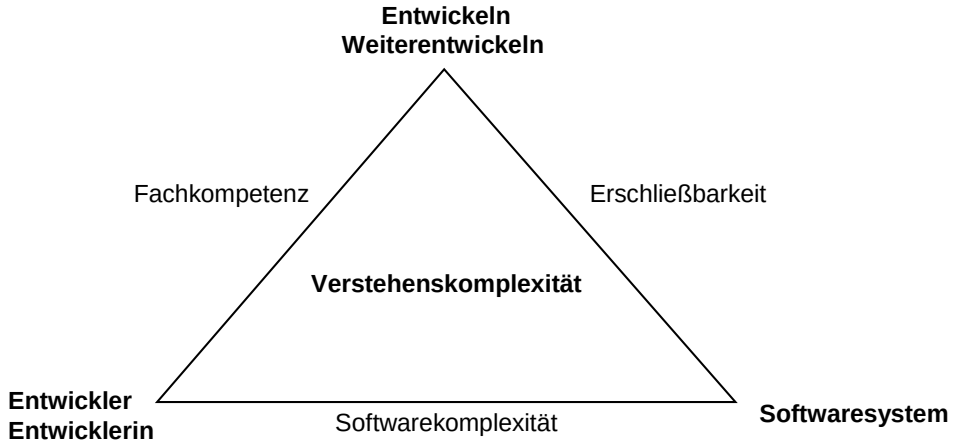

Abbildung 2-3: Verstehenskomplexität bei der Softwareentwicklung

Verstehenskomplexität wird zum einen beeinflusst von der *Fachkompetenz* der Entwicklerinnen und Entwickler für die Aufgabe (Weiter-)Entwicklung. Fachkompetenz beinhaltet Kenntnisse über softwaretechnische Grundprinzipien, die verwendete Programmiersprache, die Entwicklungsumgebung und die verwendete Infrastruktur. Wird eine Java-Entwicklerin oder ein Java-Entwickler beispielsweise mit der Aufgabe konfrontiert, ein Cobol-Programm auf einem Großrechner zu erweitern, so ist die Verstehenskomplexität hoch. Dieser Aspekt von Verstehenskomplexität wird von Henderson-Sellers mit dem Begriff *programmer characteristics* bezeichnet und er weist darauf hin, dass dieser Aspekt schwer zu untersuchen ist (Henderson-Sellers 1996, S. 51).

(Weiter-)Entwicklung kann außerdem nur dann gelingen, wenn das Softwaresystem *erschließbar* ist. Ein Softwaresystem ist dann leicht erschließbar, wenn sein Programmtext, die Dokumentation, eine passende Arbeitsumgebung und die benötigten Werkzeuge für die Entwicklerin und den Entwickler verfügbar sind. Ist der Programmtext verloren gegangen und Weiterentwicklung nur durch einen Eingriff in den Maschinencode möglich, erhöht sich die Verstehenskomplexität beträchtlich. Ein gut dokumentiertes Softwaresystem hingegen, dessen Programmtext mit allen benötigten externen Komponenten in der entsprechenden Umgebung bereitsteht, ist leichter erschließbar und somit leichter verständlich.

Die Fachkompetenz der Entwicklerinnen und Entwickler sowie die Erschließbarkeit des Softwaresystems sind nicht Thema dieser Arbeit. Der Fokus liegt vielmehr auf der Komplexität des Softwaresystems selbst, also der unteren Achse in Abbildung 2-3. Was ich in dieser Arbeit unter Softwarekomplexität verstehe, erläutere ich im nächsten Abschnitt.

2.4. Softwarekomplexität

Die Komplexität des Softwaresystems speist sich aus zwei verschiedenen Quellen: dem Anwendungsproblem, für das das Softwaresystem gebaut wurde, und der Lösung aus Programmtext, Datenbank usw. In der Literatur

lassen sich verschiedene Begriffe für diese beiden Dimensionen von Softwarekomplexität finden. Ich folge hier der von Ebert eingeführten Unterscheidung in *problem-inhärente* und *lösungs-abhängige Komplexität* (Ebert 1995, S. 174). Problem-inhärente Komplexität wird von anderen Autoren als Problemkomplexität (Henderson-Sellers 1996, S. 58) oder auch als funktionale Komplexität bezeichnet (Card und Glass 1990, S. 43f; Ebert 1995, S. 166). Dem Begriff lösungs-abhängige Komplexität lassen sich eine Reihe von Begriffen aus der Literatur zuordnen, wie strukturelle Komplexität, algorithmische oder prozedurale Komplexität (Fenton und Pfleegler 1997, S. 245, 279ff; Henderson-Sellers 1996, S. 51; Zuse 1998, S. 41).

2.4.1. Problem-inhärente Komplexität

Die *problem-inhärente Komplexität* umfasst die Schwierigkeiten, die Entwicklerinnen und Entwicklern aus Gegenstandsbereich und Einsatzkontext des Softwaresystems entstehen. Unter dem Gegenstandsbereich verstehe ich in Anlehnung an Floyd/Züllighoven den Ausschnitt der Realität, der im Softwaresystem modelliert und ggf. durch das Softwaresystem verändert wird. Der Einsatzkontext meint die Prozesse, in denen das Programm angewendet wird. Einsatzkontexte können sein die Steuerung von technischen Prozessen oder die Unterstützung von Arbeitsprozessen (Floyd und Züllighoven 2002, S. 764).

Je nachdem, welcher Gegenstandsbereich und welcher Einsatzkontext gewählt wird, kann die problem-inhärente Komplexität unterschiedliche Ursachen haben: Für eine Versicherungssoftware müssen beispielsweise eine Vielzahl von Versicherungsprodukten abstrakt und generisch modelliert werden, so dass alle Produktvarianten abbildbar bleiben und gleichzeitig die Struktur des Kernkonzepts erweiterbar bleibt. Die Komplexität entsteht durch die Vielfalt der Produkte. Hingegen ist ein Einzelplatzsystem für einen Buchhalter im Bereich der Berechnungs- und Buchungsregeln algorithmisch komplex. Oder aber ein Softwaresystem wird verwendet, um den Transport von Fahrzeugen zwischen Autovermietungsstationen zu organisieren, so dass alle beteiligten Personen wissen, wo sich die Fahrzeuge befinden und in welchem Zustand sie sind. Hier liegt ein beträchtlicher Anteil an problem-inhärenter Komplexität in der Vielzahl der Zustände und Bearbeitungsschritte, die durch die Kooperation zwischen zentralem Fahrzeug-Disponenten, Stationsmitarbeiter und Spediteur möglich sind.

Entwicklerinnen und Entwickler stehen in der Regel vor der Schwierigkeit, dass sich problem-inhärente Komplexität nicht vermeiden oder reduzieren lässt, sondern nur mit möglichst guten Analysemethoden durchdrungen werden kann (Card und Glass 1990, S. 46; Henderson-Sellers 1996, S. 51). In dieser Arbeit wird problem-inhärente Komplexität und die Möglichkeit, sie zu bestimmen, nicht weiter untersucht, sondern es wird die Architektur von Softwaresystemen und damit die lösungs-abhängige Komplexität betrachtet.

2.4.2. Lösungs-abhängige Komplexität

Neben der problem-inhärenten Komplexität steht die *lösungs-abhängige Komplexität*. Hat ein Entwicklungsteam durch eine Analyse einen Einblick in das Problem bekommen, so muss es eine Lösung entwerfen und implementieren. Dabei setzen die Teammitglieder softwaretechnische Prinzipien aus ihrem Fachwissen ein, um zu einer funktionierenden Lösung zu kommen. Bei diesem Prozess entsteht lösungs-abhängige Komplexität, die sich in Entwurfsdokumenten und Programmtexten niederschlägt.

Henderson-Sellers versucht in seinem Buch zu Komplexitätsmessungen, lösungs-abhängige Komplexität in Entwurfskomplexität und Programmkomplexität zu unterteilen (Henderson-Sellers 1996, S. 52). Dieser Unterscheidung folge ich hier nicht, sondern betrachte Softwareentwicklung im Sinne der Design-Sicht bei Floyd insgesamt als Entwurf, der sich in der Lösung niederschlägt und lösungs-abhängige Komplexität aus Entwurf und Implementierung besitzt (Floyd 1992a, S. 93f). Ähnlich argumentieren Card/Glass und Ebert, die von Designkomplexität sprechen (Card und Glass 1990, S. 43f; Ebert 1995, S. 166).

Die in der Lösung enthaltene Komplexität geht über die problem-inhärente Komplexität hinaus, wird aber auch von ihr beeinflusst (Henderson-Sellers 1996, S. 51). Als Beispiel, wie problem-inhärente und lösungs-abhängige Komplexität zusammenhängen, soll hier das Rollenmuster aus der objektorientierten Programmierung und sein Einsatz im Gegenstandsbereich Finanzgeschäfte dienen (Züllighoven 2005, S. 315f). Wird ein Softwaresystem entwickelt, das verschiedene Sparten im Finanzgeschäft abdeckt, so muss das Kernkonzept „Kunde“ in allen Sparten zur Verfügung stehen, dort aber um jeweils spezifische Zusatzinformationen angereichert werden. Diese Anforderung ist im Gegenstandsbereich enthalten und weist für Entwicklerinnen und Entwickler einiges an problem-inhärenter Komplexität auf. Das Kernkonzept „Kunde mit spartenspezifischen Rollen“ ist in der Regel schwerer zu verstehen als ein Kernkonzept, das in allen Sparten mit denselben Eigenschaften benötigt wird. Haben die Entwicklerinnen und Entwickler das Problem verstanden, so müssen sie eine Lösung entwerfen und implementieren, die diese Anforderung umsetzt. Zu diesem Zweck bietet sich das Rollenmuster an.

Das Rollenmuster macht es möglich, dass das Kernkonzept Kunde mit Schnittstellen und Attributen für die verschiedenen Sparten versehen wird, die nicht in den Systemteilen der jeweils anderen Sparten sichtbar sind. Die lösungs-abhängige Komplexität ist angemessen für die problem-inhärente Komplexität, weil die Anforderung, verschiedene Rollen zu haben, in objektorientierten Softwaresystemen nur auf diese Weise umgesetzt werden kann (vgl. Bäumer 1998). Die Lösung hat unabhängig vom Problem ihre eigene lösungs-abhängige Komplexität, denn beim Rollenmuster werden auf mehreren Stufen Vererbungshierarchien mit Benutzungsbeziehungen kombiniert eingesetzt, was im Vergleich zur einfachen Vererbungshierarchie eine komplexere Struktur entstehen lässt.

Die in Abschnitt 2.3 getroffene Unterscheidung zwischen Verstehens- und Softwarekomplexität lässt sich an diesem Beispiel ebenfalls verdeutlichen: Wird das Rollenmuster eingesetzt, entsteht Softwarekomplexität, bestehend aus problem-inhärenter und lösungs-abhängiger Komplexität. Für die Entwicklerinnen und Entwickler kommt aber, abhängig von ihrem Fachwissen über das Rollenmuster und ihrer Möglichkeit, sich das Softwaresystem zu erschließen, Verstehenskomplexität in unterschiedlichem Ausmaß hinzu.

Dadurch, dass für ein Problem eine angemessene Lösung gefunden werden muss, besteht zwischen problem-inhärenter und lösungs-abhängiger Komplexität für Softwareprojekte oft eine kritische Wirkungskette. Die Ergebnisse einiger Untersuchungen machen deutlich, dass ein Anstieg der problem-inhärenten Komplexität zu einem mehrfachen Anstieg in der lösungs-abhängigen Komplexität führt (Glass 2002, S. 19ff; Woodfield 1979, S. 76). Dieser Zusammenhang ist die Ursache dafür, dass Vorhersagen über die Kosten oder die Dauer einer Softwareentwicklung häufig zu gering ausfallen, weil die eigentliche Komplexität des Problems zu Beginn eines Projekts nicht erfasst und so die Komplexität der Lösung um ein Vielfaches unterschätzt wurde (Booch 2004, S. 3; McBride 2007, S. 76f). Hier setzen eine Reihe von iterativen Methoden an, die versuchen, am Ende jeder Iteration auf Basis eines Prototyps nur die als nächstes zu implementierende Funktionalität zu schätzen und so die problem-inhärente Komplexität und die daraus resultierende Komplexität der Lösung im Prozess immer wieder neu zu betrachten (vgl. Beck 2005; Floyd et al. 1989; Jacobson et al. 1999; Lippert et al. 2002; Züllighoven 2005).

Neben der Problematik, dass problem-inhärente Komplexität schwer zu bestimmen ist, können der Entwurf und die Implementierung eines Problems je nach Erfahrung und Methodenfestigkeit der Entwicklerinnen und Entwickler unterschiedlich komplex ausfallen (Fenton und Pfleegler 1997, S. 245; Henderson-Sellers 1996, S. 51). Im Idealfall würde man sich wünschen, dass die Lösung eine für das Problem angemessene Komplexität aufweist. In diesem Fall kann man davon sprechen, dass es sich um eine gute Lösung handelt. Weist die Lösung mehr Komplexität als das eigentliche Problem auf, so ist die Lösung nicht gut gelungen und ein entsprechendes Redesign ist erforderlich. Dieser Unterschied zwischen besseren und schlechteren Lösungen wird im Folgenden mit dem von Brooks eingeführten Begriffspaar essentielle und akzidentelle Komplexität bezeichnet (Brooks 1986, S. 11).

2.4.3. Essentielle und akzidentelle Komplexität

Mit *essentieller Komplexität* bezeichnet Brooks die Art von Komplexität, die im Wesen einer Sache liegt, also Teil seiner Essenz ist. Im Gegensatz dazu verwendet Brooks den Begriff *akzidentelle Komplexität*, um auf Komplexitätsanteile hinzuweisen, die nicht notwendig sind und somit beseitigt bzw. verringert werden können (Brooks 1986, S. 11). Akzidentelle Komplexität kann sowohl aus Missverständnissen bei der Analyse des Gegenstandsbereichs als auch bei der Implementierung durch das Entwicklungsteam entstehen.

Bei der Analyse eines Gegenstandsbereichs versuchen Entwicklerinnen und Entwickler die essentielle Komplexität des Problems zu identifizieren, damit nur sie in die Lösung übertragen wird. Die einem Gegenstandsbereich innewohnende essentielle Komplexität führt zu einer entsprechend komplexen Lösung und lässt sich niemals auflösen oder durch einen besonders guten Entwurf vermeiden (Brooks 1986, S. 1070; Glass 2002, S. 19; McBride 2007, S. 75). Dieses Identifizieren der essentiellen Komplexität ist ein schwieriges Unterfangen und kann nach meiner Erfahrung nur gelingen, wenn eine sorgfältige Ist-Analyse des Gegenstandsbereichs vorgenommen wurde. Bei der Ist-Analyse werden die Gegenstände und Abläufe modelliert, wie sie zu Beginn des Entwicklungsprojekts bereits existieren. Von der Ist-Analyse wird die Soll-Modellierung unterschieden, die den Einsatz des zu entwickelnden Softwaresystems antizipiert.

In einer Reihe von Softwareentwicklungsmethoden wird der Ist-Analyse zu wenig Gewicht beigemessen. Als Beispiel sei hier der als Standard für Entwicklungsprozesse beschriebene *Unified Process* (UP) angeführt. Die Ergebnisse der Anforderungsanalyse bestehen im UP aus einem Use Case Model, in dem die zukünftigen Abläufe im Softwaresystem beschrieben werden sollen (Jacobson et al. 1999, S. 133ff). Die Abläufe und Kernkonzepte, die ohne Softwaresystem durchgeführt und bearbeitet werden, nehmen im Unified Process nur eine untergeordnete Rolle ein und werden nicht als Artefakte des Entwicklungsprozesse aufgefasst (Jacobson et al. 1999, S. 119ff). Dadurch verringert der Unified Process das Potential, die essentiellen und akzidentellen Anteile der problem-inhärenten Komplexität zu identifizieren. Seit den 1970er Jahren wurden alternative partizipative Vorgehensweisen entwickelt, deren Ziel es ist, durch Beteiligung der Benutzerinnen und Benutzer die essentiellen Anforderungen zu identifizieren (vgl. Beck 2005; Floyd et al. 1989; Lilienthal und Züllighoven 1997; Wetzel et al. 1998).

Die zweite Ursache für akzidentelle Komplexität ist bei der Entwicklung des Softwaresystems zu suchen. Wird bei der Entwicklung aus Unkenntnis oder mangelndem Überblick keine einfache Lösung gefunden, so ist das Softwaresystem überflüssigerweise komplex. Beispiele hierfür sind: Mehrfachimplementierungen, Einbau nicht benötigter Funktionalität und das Nichtbeachten softwaretechnischer Entwurfsprinzipien. Akzidentelle Komplexität kann von Entwicklerinnen und Entwicklern aber auch billigend in Kauf genommen werden, wenn sie z.B. gern neue und für das zu bauende Softwaresystem überflüssige Technologie ausprobieren wollen. Neben überflüssiger Technologie zähle ich auch Design auf Vorrat zur akzidentellen Komplexität. Hat ein Entwickler beispielsweise das schon erwähnte Rollenmuster in einem Projekt kennengelernt, kann es passieren, dass er es in seinem nächsten Projekt wieder einsetzt. Einerseits kennt und versteht er es, andererseits kann es sein, dass er den alten Programmtext kopiert und an das neue Problem anpasst. Dieses Weitertragen von bekannten Lösungen führt zu akzidenteller Komplexität, wenn der Gegenstandsbereich ähnliche, aber nicht dieselben Anforderungen beinhaltet wie das vorherige Projekt. Die Problematik wird besonders von agilen Methoden an den Pranger gestellt und durch ihre Vorschläge für

eine auf das wesentliche reduzierende Vorgehensweise bekämpft (vgl. Beck 2005; Lippert und Züllighoven 2002; Martin 2003).

Problem-inhärente, essentielle und akzidentelle Komplexität bilden die Grundlage, auf der im nächsten Kapitel der Begriff *Architekturkomplexität* eingeführt werden kann.

2.4.4. Ausdifferenzierung lösungs-abhängiger Komplexität

Neben der bisher herausgearbeiteten Unterteilung von Softwarekomplexität werden in der Literatur weitere Kategorien von Komplexität vorgeschlagen. Diese Kategorien fallen alle in den Bereich der lösungs-abhängigen Komplexität. Sie werden von den Autoren unterschieden, um verschiedene Aspekte von Komplexität zu identifizieren und passende Maße zu bestimmen:

- *Algorithmische Komplexität*: Algorithmen lassen sich daraufhin untersuchen, wie effizient sie in Sinne der O-Notation aus der Komplexitätstheorie der Informatik arbeiten (Fenton und Pfleegler 1997, S. 245, 274).
- *Datenkomplexität*: Die Komplexität von Elementen eines Softwaresystems wird als die von ihnen an Daten durchgeführte Verarbeitung betrachtet. Beispielsweise wird die Anzahl der Ein- und Ausgabevariablen von Elementen verglichen (Card und Glass 1990, S. 49f; Fenton und Pfleegler 1997, S, 280).
- *Prozedurale Komplexität*: Die Komplexität von einzelnen Prozeduren wird als prozedurale oder Kontrollfluss-Komplexität bezeichnet (Card und Glass 1990, S. 50f; Fenton und Pfleegler 1997, S. 280ff). Ein gängiges Beispiel ist die Berechnung der zyklomatischen Komplexität von McCabe, bei der anhand des Kontrollflussgraphen einer Prozedur die Anzahl der unabhängigen Pfade bestimmt wird (vgl. McCabe 1976).
- *Strukturelle Komplexität*: Die Vielzahl an Elementen und Beziehungen in einem Softwaresystem wird als komplexe Struktur betrachtet, und verschiedene Eigenschaften dieser Struktur werden vermessen. Beispielsweise wird die Anzahl der ein- und ausgehenden Beziehungen der Elemente gezählt und verglichen (Card und Glass 1990, S. 48f; Fenton und Pfleegler 1997, S. 245, 279ff; Henderson-Sellers 1996, S. 51; Zuse 1998, S. 41).

Diese vier Kategorien weisen einige Unterschiede auf: Algorithmische Komplexität unterscheidet sich stark von den anderen Kategorien, weil sie sich nicht mit statischen Eigenschaften von Softwaresystemen, sondern mit ihrer Effizienz befasst. Diese Art der Komplexität wird hier nicht untersucht. Daten- und prozedurale Komplexität beziehen sich auf Komplexitätseigenschaften von einzelnen Softwareelementen; strukturelle Komplexität hingegen untersucht die Komplexität zwischen Softwareelementen (Fenton und Pfleegler 1997, S. 245; Henderson-Sellers 1996, S. 58). Im nun folgenden Kapitel wird deutlich werden, dass die Architektur von Softwaresystemen

Strukturierungsmittel für die Beziehungen zwischen Softwareelementen vorgibt. Die in dieser Arbeit untersuchte lösungs-abhängige Komplexität ist daher strukturelle Komplexität. Prozedurale und Datenkomplexität werden im Weiteren nicht betrachtet.

2.5. Zusammenfassung

In diesem Abschnitt habe ich drei Ebenen eingeführt, auf denen Komplexität untersucht werden kann: Die Komplexität der Interaktion im Entwicklungsteam, Verstehenskomplexität beim Menschen und Softwarekomplexität im technischen Artefakt.

Wie ein Entwicklungsteam mit Komplexität umgeht, habe ich mithilfe der Theorie *sozialer Systeme* von Luhmann betrachtet. Entwicklungsteams müssen Kapazitätsreserven für den Umgang mit ihrer eigenen Komplexität und der ihrer Umwelt aufbauen, wenn sie Bestand haben wollen. Die Kapazitätsreserve wird dazu genutzt, im Rahmen einer bewussten Planung Strategien einzuführen, wie Komplexität reduziert werden kann.

Als zweite Ebene habe ich *Verstehenskomplexität* eingeführt. Verstehenskomplexität entsteht bei Entwicklerinnen und Entwicklern, die die Aufgabe haben, ein Softwaresystem zu entwickelt bzw. weiterzuentwickeln. Für diese Aufgabe benötigen sie Fachkompetenz, das Softwaresystem muss erschließbar sein und die Softwarekomplexität muss beherrschbar sein.

Diese dritte Ebene der *Softwarekomplexität* hat zwei verschiedene Quellen: problem-inhärente und lösungs-abhängige Komplexität. Der strukturelle Anteil der *lösungs-abhängigen Komplexität* steht in dieser Arbeit im Zentrum der Betrachtung. Schließlich habe ich die Begriffe *essentielle* und *akzidentelle Komplexität* eingeführt, mit denen zwischen notwendiger und überflüssiger Komplexität unterschieden werden kann.

Im nächsten Kapitel mache ich mit dem Thema Softwarearchitektur einen ersten Schritt in Richtung der Beherrschbarkeit von lösungs-abhängiger Komplexität. Softwarearchitekturen eröffnen höhere Betrachtungsebenen und schränken Entwicklungsteams bei der Implementierung ein, so dass weniger Komplexität entstehen kann.

3. Softwarearchitektur

In diesem Kapitel fasse ich den aktuellen Stand der Forschung im Bereich Softwarearchitektur zusammen. Dazu gehe ich von der Definition im ANSI/IEEE-Standard aus und beschreibe die Elemente objektorientierter Architekturen. Die Einschränkung auf objektorientierte Architekturen nehme ich vor, weil alle Fallstudien in Java implementiert worden sind. Weiterhin stelle ich die vier Architekturstile vor, die in den Fallstudien verwendet wurden. Danach führe ich die Unterscheidung zwischen Soll- und Ist-Architektur ein und schließe das Kapitel mit einer Definition für *Architekturkomplexität* ab.

3.1. Der Architekturbegriff in der Softwareentwicklung

Zu Beginn der 1960er Jahre wurde das Wort *Architektur* aus dem Bauwesen auf die Konstruktion von Computer-Systemen übertragen. Zemanek beschreibt in seinem Buch ausführlich, wie der Begriff *Computer-Architektur* im Jahr 1962 in einem Beitrag von Brooks mit dem Titel „Architectural Philosophy“ eine erste Definition bekommen hat (Zemanek 1992, S. 127f). Zu dieser Zeit wurde zwischen Hard- und Software noch nicht unterschieden und der Begriff Computer-Architektur wurde hauptsächlich für Hardware verwendet, auf der Software als oberste Schicht aufsetzte.

Dijkstra und Parnas machten jedoch bereits um 1970 deutlich, dass es Erfolg versprechend ist, sich Gedanken darüber zu machen, wie ein Softwaresystem gegliedert und strukturiert sein soll (vgl. Dijkstra 1968; Parnas 1979; Parnas 1972). Softwaresysteme sollten nach ihrer Meinung nicht nur mit dem Ziele programmiert werden, dass sie ein richtiges Ergebnis liefern, sondern sie brauchen, genau wie Hardwaresysteme, eine Architektur. Die Unterscheidung zwischen Systemarchitektur für die Hardware und *Softwarearchitektur* nahm hier ihren Anfang. Im weiteren Verlauf dieser Arbeit werde ich anstelle von Softwarearchitektur den Begriff Architektur verwenden. Die Systemarchitektur ist für diese Arbeit irrelevant.

Aufbauend auf den in diesem Zusammenhang von Dijkstra und Parnas veröffentlichten Erkenntnissen zu Schichtenbildung und Modularisierung, wurde in den 1970er und 1980er Jahren die Diskussion unter den Stichworten Modularisierung und „Programmierung im Großen“ weitergeführt (vgl. Myers 1978; Nagl 1990; Parnas et al. 1985). Schließlich erlebte das Thema *Architektur* seit Mitte der 1990er Jahre mit der Ausbreitung der objektorientierten Programmierung viel Beachtung. Eine Reihe von Autoren untersuchten, was der Begriff Architektur bedeutet, welchen Einfluss Architektur auf den Softwareentwicklungsprozess hat und wie man die Architektur eines Softwaresystems beschreiben kann (vgl. Gamma et al. 1994; Jacobson et al. 1999; Kruchten 1995; Soni et al. 1995).

Im Jahr 2000 wurde die Diskussion um Architektur in einem Standard zusammengefasst, so dass heute eine allgemein gültige Definition vorliegt (ANSI/IEEE-Standard-1471 2000).

Definition 3-1: Architektur nach ANSI/IEEE-Standard 1471-2000

> Architektur sind die grundlegende Organisation eines Systems, die in seinen Komponenten, ihren Beziehungen untereinander und zur Umgebung enthalten ist, sowie die Prinzipien, die den Entwurf und die Weiterentwicklung des Systems anleiten.

Diese Definition ist so allgemein gefasst, weil sie als Grundlage dienen soll, auf der verschiedenste Sichtweisen eines Softwaresystems beschrieben werden können. Um dies zu ermöglichen, definiert der Standard zusätzlich einen konzeptionellen Rahmen zur Beschreibung von Architekturen mit Hilfe von Modellen und Sichten. Eine Architekturbeschreibung besteht aus einer Menge von Modellen, die aus textuellen Spezifikationen oder grafischen Diagrammen bestehen. Um die Teilmengen der Modelle zusammenzufassen, wird das Konzept der Sichten eingeführt. Eine Sicht drückt nach dem ANSI/IEEE-Standard einen Aspekt eines Softwaresystems mit Hilfe von mehreren Modellen aus.

Diese abstrakte Definition wird in den verschiedenen Schulen, die im Bereich Architektur federführend sind, mit Leben gefüllt (vgl. Bass et al. 2003; Hofmeister et al. 2000; Kruchten 1995; Reussner und Hasselbring 2006). Die einzelnen Schulen beschreiben verschiedene Sichten auf Architektur sowie die dafür verwendeten Modelle. Drei Sichten lassen sich in allen Schulen unter jeweils eigenen Namen finden. In Anlehnung an Reussner und Hasselbring benenne ich diese drei Sichten wie folgt: *statische Sicht*, *dynamische Sicht* und *Verteilungssicht* (Reussner und Hasselbring 2006, S. 41) (s. Abbildung 3-1).

Diese drei Sichten reichen aus, um das Softwaresystem in seiner statischen Struktur und seinem dynamischen Verhalten zu beschreiben. Die Einbettung des Softwaresystems in den fachlichen Kontext des Anwendungsgebiets wird durch diese Sichten nicht vorgenommen. Um diesem Mangel zu begegnen, wird bei einigen Autoren eine weitere Sicht hinzugefügt: die fachliche Sicht (Hofmeister et al. 2000, S. 61; Kruchten 1995, S. 47; Züllighoven 2005, S. 101f) (s. Abbildung 3-1).

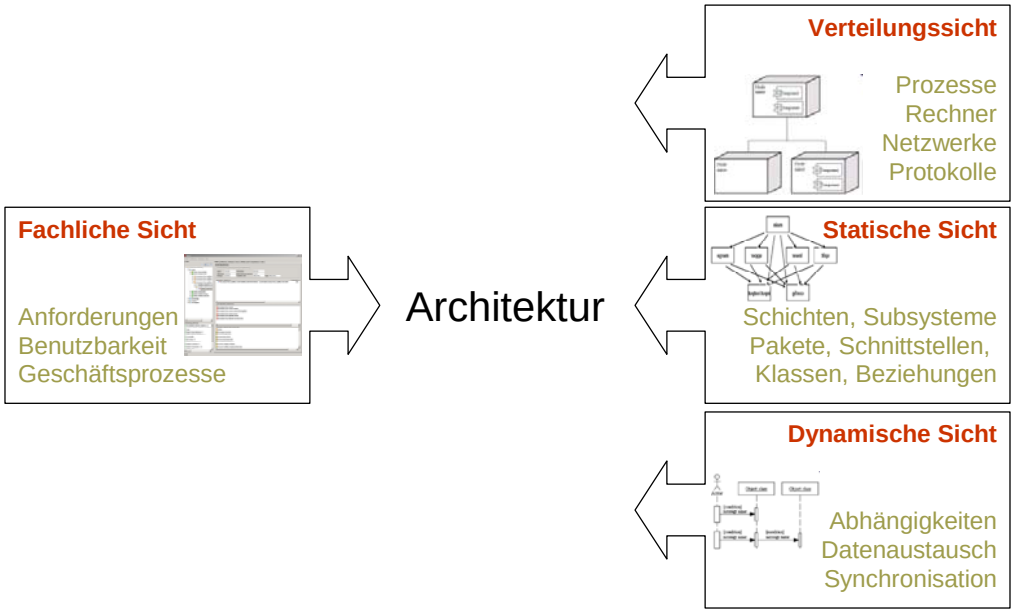

Abbildung 3-1: Sichten auf Architektur

Mit der *fachlichen Sicht* wird der Versuch unternommen, die problem-inhärente Komplexität zu beschreiben. Der Gegenstandsbereich und der Einsatzkontext mit den Kernbegriffen, den Geschäftsprozessen und die notwendige Unterstützung für die Anwender werden mit Hilfe der fachlichen Sicht modelliert. Dadurch wird die vorhandene essentielle Komplexität nicht aufgelöst – was unmöglich ist (s. Abschnitt 2.4.3) –, sondern sie wird in ihrem Ausmaß sichtbar gemacht.

Die *statische Sicht* auf Architektur beschreibt die Aufteilung des Programmtextes in Schichten, Subsystemen, Pakete und Klassen. Im Folgenden verwende ich in Anlehnung an Bass et al. den Begriff Architekturelemente, wenn ich von den „Komponenten" der statischen Sicht spreche. Der Begriff Komponente wird heute als Bezeichnung für Laufzeitkomponenten in der dynamischen und Verteilungssicht verwendet (Bass et al. 2003, S. 21). In der statischen Sicht wird die Aufteilung des Softwaresystems in Architekturelemente festgelegt und außerdem beschrieben, welche Architekturelemente Beziehungen zu welchen anderen Architekturelementen haben dürfen und welche Schnittstellen von den Architekturelementen zur Verfügung gestellt werden.

Die *dynamische Sicht* auf eine Architektur drückt aus, wie sich das Softwaresystem und seine einzelnen Komponenten zur Laufzeit verhalten und welche Laufzeitabhängigkeiten, Kontrollflüsse und Synchronisationspunkte es zwischen den einzelnen Komponenten gibt.

Die *Verteilungssicht* auf die Architektur macht deutlich, welche Teile des Softwaresystems als eigene Komponenten lauffähig sind, welchen Rechnern oder Prozessen sie zugeordnet werden und über welche Netzprotokolle sie miteinander kommunizieren.

Von diesen drei Sichten ist die statische Sicht die älteste, und ein Großteil der Diskussionen in den 1960er bis 1980er Jahren befasste sich ausschließlich mit der statischen Sicht. Diese Tatsache ist nicht verwunderlich, denn die mit der statischen Sicht dokumentierte Struktur eines Softwaresystems legt für die Verteilungs- und die dynamische Sicht die Entfaltungsmöglichkeiten fest. Ohne beispielsweise zu wissen, aus welchen Paketen ein Softwaresystem besteht, ist es nicht möglich, über die Verteilung dieser Pakete auf verschiedene Rechner zu sprechen. Die anderen Sichten können also ohne Wissen über die statische Sicht nicht entworfen, analysiert oder dokumentiert werden.

Auch in dieser Arbeit liegt der Fokus auf dem strukturellen Anteil der lösungs-abhängigen Komplexität und somit auf der statischen Sicht. Die Elemente der statischen Sicht werden im nächsten Abschnitt eingeführt.

3.2. Elemente objektorientierter Architekturen

Die statische Sicht auf Architektur enthält zwei Arten von *Architekturelementen*: Klassen und Subsysteme, sowie drei Arten von *Beziehungen*: Enthaltenseins-, Benutzt- und Vererbungsbeziehung. Zusätzlich zu Architekturelementen und Beziehungen verwende ich in dieser Arbeit als weiteres Ausdrucksmittel die *Schnittstelle*. Sowohl Klassen als auch Subsysteme legen Schnittstellen fest, über die sie oder die in ihnen enthaltenen Klassen benutzt werden können. Schnittstellen werden in der Softwaretechnik als wichtiges Mittel der Architektur betrachtet, um Entwurfsprinzipien, wie Abstraktion und Information Hiding, durchzusetzen (Nagl 1990, S. 88f). In neueren Arbeiten zu Architektur werden sie als drittes Element neben Architekturelementen und Beziehungen genannt (Bass et al. 2003, S. 21).

Im Folgenden werden zuerst die Architekturelemente der Klassen-Ebene, ihre Beziehungen und Schnittstellen eingeführt. Anschließend wird die Subsystem-Ebene hinzugefügt und ihr Zusammenspiel mit der Klassen-Ebene beschrieben.

3.2.1. Klassen-Ebene der Architektur

Die ersten objektorientierten Sprachen, wie Smalltalk und C++, griffen das Klassen- und Vererbungskonzept aus Simula 67 wieder auf und führten es mit dem Modul-Konzept zusammen. Mit dem Modul-Konzept von Parnas wurden Lokalität und prozedurale Schnittstellen als wesentliche Prinzipien für Programmstrukturierung eingeführt. Die in den letzten Jahren entstandenen objektorientierten Sprachen, wie Java und C#, führten schließlich, neben der Klasse, das Interface ein. Zwischen Klassen und Interfaces unterscheide ich in dieser Arbeit nicht, weil Interfaces in der statischen Sicht auf Architektur genau wie Klassen als Typ verwendet werden und Beziehungen zu anderen Klassen und Interfaces haben können (s. Abbildung 3-2).

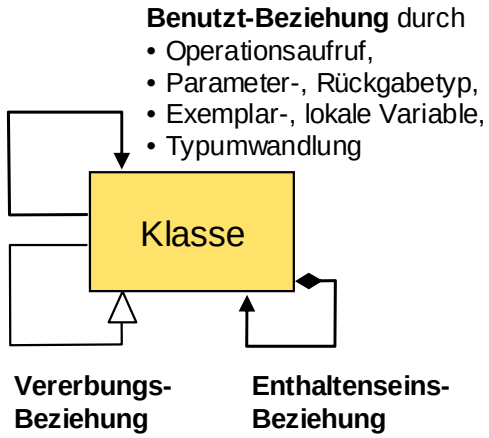

Abbildung 3-2: Ausdrucksmittel der Klassen-Ebene

In objektorientierten Programmiersprachen stehen drei *Beziehungen* zwischen Klassen zur Verfügung. Die Enthaltenseins-Beziehung macht die Schachtelung von Klassen und in sie eingebetteten Klassen möglich. Von einer Benutzt-Beziehung zwischen Klassen spricht man, wenn im Programmtext einer Klasse eine Operation einer anderen Klasse aufgerufen wird, sowie eine andere Klasse als Parameter- und Rückgabetyp, als Typ einer Exemplarvariable, als Typ einer lokalen Variable oder bei Typumwandlungen verwendet wird. Die Vererbungsbeziehung bewirkt, dass Architekturelemente Eigenschaften von anderen Architekturelementen übernehmen. Für Klassen lässt sich eine *Schnittstelle* festlegen, die angibt, welche der von ihr definierten Operationen von anderen Klassen aufgerufen werden dürfen.

3.2.2. Subsystem-Ebene der Architektur

Oberhalb der Klassen-Ebene führe ich eine weitere Ebene der Strukturierung ein, mit deren Hilfe Klassen zu größeren Einheiten zusammengefasst werden können. Das Bedürfnis, größere Einheiten in Softwaresystemen zu bilden, bestand bereits in den 1980er Jahren, als unter dem Stichwort Programmierung im Großen das Modul-Konzept in Programmiersprachen eingeführt wurde (Nagl 1990, S. 77ff) und von Parnas geschachtelte Module gefordert wurden (Parnas et al. 1985, S. 260). In objektorientierten Programmiersprachen, wie Java und C#, steht zu diesem Zweck das *Package* zur Verfügung.

Architekturelemente auf der Subsystem-Ebene

Um oberhalb von Klassen größere Einheiten möglich zu machen, führe ich als Architekturelement das *Subsystem* ein. Ein Subsystem ist eine Organisationseinheit im Softwaresystem, die auf einer niedrigeren Ebene stehende Architekturelemente zusammenfasst (Bischofberger et al. 2004, S. 1). Subsysteme werden mithilfe der Enthaltenseins-Beziehung rekursiv ineinander geschachtelt, so dass neben Klassen auch Subsysteme zu einem übergeordneten Subsystem zusammengefasst werden können (Brügge und Dutroit 2004, S. 262; Jacobson 1992, S. 52). Auf diese Weise entsteht eine beliebig tiefe Enthal-

tenseins-Hierarchie, mit der alle Arten von Architekturen in den folgenden Abschnitten beschrieben werden können (s. Abbildung 3-3). In Sprachen wie Java und C# bilden Packages bzw. Namespaces die erste Ebene der Subsysteme.

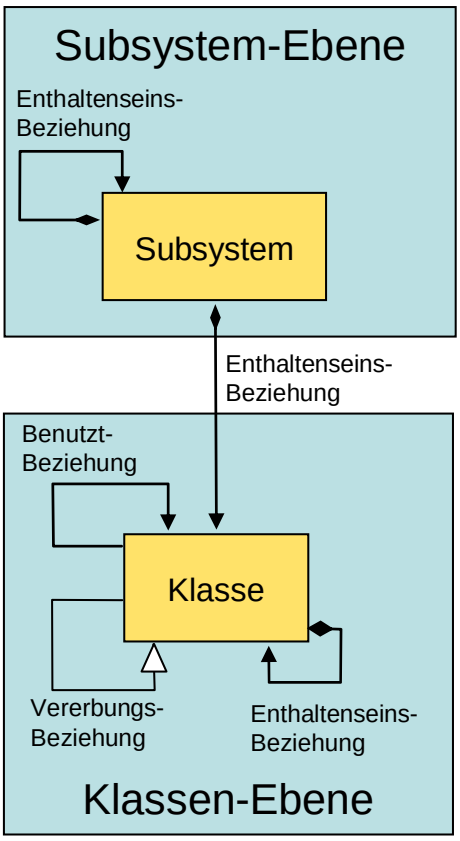

Abbildung 3-3: Ausdrucksmittel der Klassen- und Subsystem-Ebene

Das Schnittstellenkonzept wurde für Subsysteme adaptiert. Die unterste Ebene der Subsysteme (Packages) hat eine Schnittstelle, die aus der Menge aller in ihnen enthaltenen öffentlichen Klassen besteht. In der Schnittstelle aller höherer Subsystem-Ebenen wird festlegt, welche der in ihnen enthaltenen Architekturelemente von außen zugreifbar sind. Hier können Subsysteme oder Klassen in der Schnittstelle angeboten werden.

In den verschiedenen Schulen, die es im Bereich Architektur gibt, werden unterschiedliche Begriffe für Subsysteme verwendet. Gebräuchlich sind die Begriffe: Modul, Modellierungseinheit oder Komponente. Bass et al. bezeichnen Subsysteme als Module (Bass et al. 2003, S. 36). Der Begriff *Modul* wird in dieser Arbeit nicht verwendet, weil er einerseits eng mit der imperativen und objektbasierten Programmierung verknüpft ist und bei der Betrachtung von objektorientierten Sprachen Module oft mit Klassen gleichgesetzt werden. Andererseits wird unter einem Modul bei manchen Autoren eher ein abstraktes Konzept einer Entwurfseinheit und ihren Eigenschaften, wie Information Hiding und Schnittstelle verstanden, das sich ebenfalls auf Packages oder

Subsysteme abbilden lässt (Brügge und Dutroit 2004, S. 263; Nagl 1990, S. 77ff; Parnas 1972, S. 1054; Parnas et al. 1985, S. 409; Züllighoven 2005, S. 42).

Züllighoven et al. verwenden den Begriff *Modellierungseinheit* für Subsysteme (Züllighoven 2005, S. 330). Da dieser Name in keiner anderen Tradition vorhanden ist, wird er in dieser Arbeit nicht eingesetzt.

Von Reussner et al. wird der Begriff *Komponente* benutzt, um modulare Teile eines Systems zu benennen, die „ein komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen" (Reussner und Hasselbring 2006, S. 48). Komponenten können wiederum andere Komponenten zusammenfassen. Der Begriff Subsystem wird bei Reussner et al. verwendet, um die oberste Ebene der Komponenten zu benennen, die die Untereinheiten eines Systems darstellen und „nicht direkt angesprochen werden können, sondern mit anderen Komponenten interagieren oder kommunizieren" (Reussner und Hasselbring 2006, S. 48). Siedersleben verwendet ebenfalls den Begriff Komponente (Siedersleben 2004, S. 41f). Komponente wird in dieser Arbeit ausschließlich verwendet, um die Elemente der dynamischen und der Verteilungssicht zu benennen (s. Kapitel 3.1).

In der Tradition von Hofmeister (Hofmeister et al. 2000, S. 100f), bei Jacobson (Jacobson et al. 1999, S. 52ff), bei Brügge/Dutroit (Brügge und Dutroit 2004, S. 60) und in der Arbeit von Ciupke zu Softwarestrukturen (Ciupke 2002, S. 19) wird der Begriff *Subsystem* genauso verwendet wie in dieser Arbeit.

Aggregation von Beziehungen auf der Subsystem-Ebene

Formal betrachtet gibt es auf der Subsystem-Ebene lediglich Enthaltenseins-Beziehungen zwischen Subsystemen und Subsystemen und Klassen (s. Abbildung 3-3). Um die Subsysteme nicht nur als Strukturierungsmittel zu verwenden, sondern Aussagen über ihr Zusammenspiel machen zu können, werden Benutzt- und Vererbungsbeziehungen auf der Subsystem-Ebene entlang der Enthaltenseins-Beziehung aggregiert. Das Wort *Aggregation* wird hier anders verwendet als in der UML-Notation[1]. Unter Aggregation wird in dieser Arbeit das Zusammenfassen von Beziehungen auf die nächst höhere Ebene entlang der Enthaltenseins-Beziehung verstanden (Bischofberger et al. 2004, S. 1; Ciupke 2002, S. 73).

Ein Subsystem aggregiert entlang der Enthaltenseins-Beziehung alle Beziehungen zwischen in ihm enthaltenen Klassen, so dass es möglich wird, die Beziehungen zwischen Subsystemen ohne Rückgriff auf Klassen zu untersuchen. Durch die Enthaltenseins-Beziehung zwischen Subsystemen werden analog alle Beziehungen der jeweils enthaltenen Subsysteme aggregiert (s. Abbildung 3-4).

[1] In UML wird eine Art der Benutzt-Beziehung als Aggregationsbeziehung bezeichnet. Damit wird in UML ausgedrückt, dass Objekte einer Klasse zu den Objekten einer anderen Klasse gehören. Wird das übergeordnete Objekt gelöscht, so werden auch alle Objekte gelöscht, zu denen das Objekt eine Aggregationsbeziehung hat (Booch et al. 1999, S. 67).

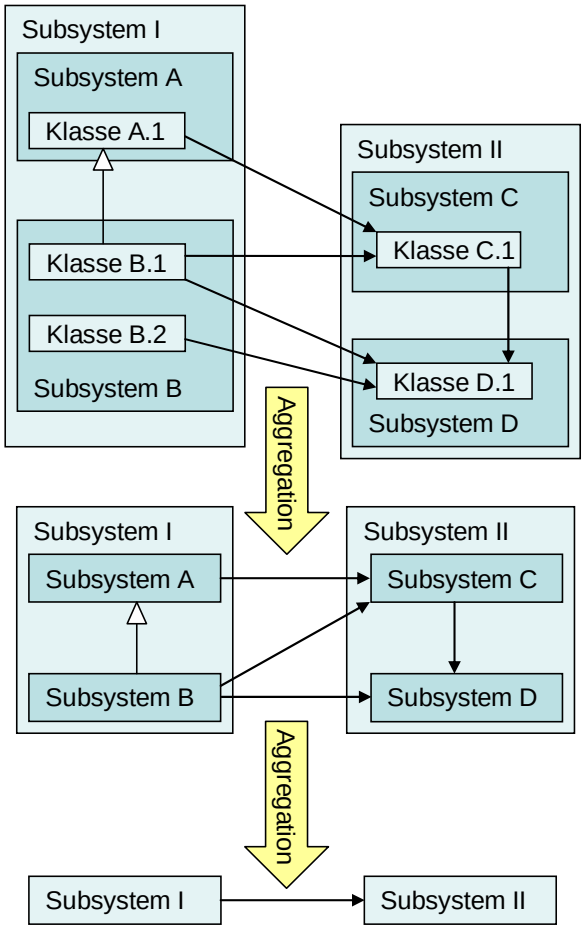

Abbildung 3-4: Aggregation auf der Subsystem-Ebene

Diese Überlegungen zur Aggregation von Beziehungen machen deutlich, dass die Enthaltenseins-Beziehung eine andere Art von Beziehung ist als die Benutzt-Beziehung. Die Enthaltenseins-Beziehung besteht immer zwischen jeweils einem übergeordneten und einem oder mehreren untergeordneten Architekturelementen. Benutzt- und Vererbungsbeziehungen sind im Gegensatz dazu Beziehungen zwischen Architekturelementen auf derselben Ebene. Mit Hilfe der Enthaltenseins-Beziehung werden Benutzt- und Vererbungsbeziehungen auf die in der Enthaltenseins-Hierarchie jeweils höheren Ebene aggregiert.

Im Weiteren fasse ich Benutzt- und Vererbungsbeziehungen unter dem Begriff *Beziehungen* zusammen und verwende den Begriff *Enthaltenseins-Beziehung* getrennt. Falls die Unterscheidung zwischen Benutzt- und Vererbungsbeziehungen notwendig ist, setze ich die speziellen Begriffe ein.

Die in diesem Abschnitt vorgestellten Architekturelemente (Klasse und Subsystem) sowie ihre Beziehungen (Enthaltenseins-, Benutzt- und Vererbungsbeziehung) und das Konzept der Schnittstellen bilden die Grundlage, auf der über die Komplexität von Architektur diskutiert werden kann. Einen

Schritt weiter gehen Architekturstile, die unterschiedliche Vorgaben für Architekturelemente, Beziehungen und Schnittstellen machen. Welche Architekturstile sich in der Praxis beobachten lassen und welche Vorgaben sie machen, werde ich im nächsten Abschnitt zeigen.

3.3. Architekturstile

Als *Architekturstil* wird eine prinzipielle Lösungsstruktur bezeichnet, die für ein Softwaresystem durchgängig und unter weitgehendem Verzicht auf Ausnahmen angewandt werden sollte (Reussner und Hasselbring 2006, S. 87). Die durch einen Architekturstil vorgegebene Lösungsstruktur lässt nicht mehr alle in der Programmiersprache möglichen Kombinationen von Architekturelementen und Beziehungen zu. Von der Programmiersprache durchaus zugelassene Beziehungen oder Schnittstellen werden durch den Architekturstil mit Hilfe von Architekturregeln verboten. Man kann daher sagen, dass ein Architekturstil eine Einschränkung des Designraums darstellt.

Der häufig eingesetzte Architekturstil *Schichtenarchitektur* verbietet zum Beispiel Beziehungen zwischen Subsystemen, die in der Schichtung von unten nach oben gehen würden. Der Architekturstil Schichtenarchitektur stammt bereits aus den 1960er Jahren, wurde also vor der objektorientierten Programmierung entwickelt (Dijkstra 1968, S. 344). Im Weiteren beschränke ich mich auf die objektorientierten Varianten von den Architekturstilen, die ich in den Fallstudien angetroffen habe.

Der Begriff *Architekturstil* wird in der Literatur synonym zu Architekturmuster (Bass et al. 2003, S. 24f; Reussner und Hasselbring 2006, S. 342) oder zur Mustersprache verwendet (Fowler 2003, S. 11; Riehle und Züllighoven 1995, S. 10). Für diese Arbeit verstehe ich unter dem Begriff Architekturstil in Anlehnung an die Begriffsbildung in der Literatur (Bass et al. 2003, S. 25; Brügge und Dutroit 2004, S. 271; Garlan 2000, S. 97; Hofmeister et al. 2000, S. 7ff; Reussner und Hasselbring 2006, S. 87) das Folgende: Ein Architekturstil legt Arten von Architekturelementen fest und gibt Architekturregeln an, die die Schnittstellen und Beziehungen der Architekturelemente einschränken.

Der Architekturstil *Pipes & Filter* beispielsweise enthält zwei Arten von Architekturelementen: Filter als ausführende Elemente, die Daten transformieren, und Pipes als Verbindungsströme, die die Daten zwischen den Filtern weiterleiten. Eine Regel dieses Stils ist, dass Pipes nur mit Filtern verknüpft werden dürfen und nicht untereinander.

Hat ein Architekturstil sich bei der Entwicklung einiger Softwaresysteme als sinnvoll erwiesen, so kann er wieder verwendet werden (Garlan et al. 1994, S. 177). Diese Wiederverwendung liegt einerseits im Bereich des Entwurfs und führt dazu, dass bewährte Lösungen in anderen Projekten adaptiert werden können. Ein Architekturstil kann sich aber auch in wieder verwendbarem Programmtext niederschlagen, wenn die unveränderlichen Aspekte des Architekturstils in einem Rahmenwerk manifestiert worden sind. Beispiel

hierfür sind das JWAM-Rahmenwerk[2] des WAM-Ansatzes und die fertigen Softwarekomponenten des Quasar-Ansatzes[3] (vgl. Siedersleben 2004; Züllighoven 2005).

Wird ein Architekturstil wiederverwendet, so entsteht eine Familie von Architekturen, die alle weitestgehend den vorgegebenen Einschränkungen genügen (Bass et al. 2003, S. 25). Diese Gleichartigkeit ist für Entwicklerinnen und Entwickler, die in der Wartung an solchen Softwaresystemen arbeiten, sehr hilfreich, weil sie aus ihrer Erfahrung mit dem Architekturstil auf die prinzipielle Struktur des Softwaresystems schließen können, und sich so schneller zurecht finden (Garlan et al. 1994, S. 177).

Im Folgenden werden die vier für diese Arbeit relevanten Architekturstile vorgestellt: die Schichtenarchitektur, die Subsystemarchitektur, die erweiterte Schichtenarchitektur sowie die Referenzarchitektur.

3.3.1. Schichtenarchitektur

Einer der am häufigsten in der Literatur beschriebenen und in vielen Unternehmen eingesetzten Architekturstile ist die *Schichtenarchitektur* (Bass et al. 2003, S. 62; Brügge und Dutroit 2004, S. 267f; Fowler 2003, S. 17; Hofmeister et al. 2000, S. 101; Jacobson 1992, S. 73f; Nagl 1990, S. 131; Züllighoven 2005, S. 345ff). Bei einer Schichtenarchitektur wird die oberste Ebene der Subsysteme als Schicht bezeichnet, und es wird gefordert, dass höhere Schichten nur Beziehungen zu den unter ihnen liegenden Schichten haben dürfen. Die einzelnen Schichten haben genau wie Subsysteme Schnittstellen, über die sie von den anderen Schichten benutzt werden. Die Einschränkungen durch die Schichtenarchitektur beziehen sich nur auf die Subsystem-Ebene der Architektur und machen keine Vorgaben für die Klassen-Ebene, so dass innerhalb der Subsysteme keine Einschränkungen für die dort enthaltenen Architekturelemente vorgeschrieben werden und innerhalb der Subsysteme beliebige Beziehungen und Schnittstellen möglich sind.

Das erste Softwaresystem, das Schichten hatte, war das THE-System von Dijkstra (Dijkstra 1968, S. 344). Bei diesem Softwaresystem wurde eine strenge Schichtung vorgenommen, so dass jede Schicht nur die direkt unter ihr liegende verwenden durfte. Der Übergang von einer Schicht zur nächsten wurde damals als Abstraktion hin zu einer höheren Sprachebene verstanden. Betriebssysteme werden nach dieser strengen Schichtung entwickelt, wie am ISO/OSI-Schichtenmodell deutlich wird (Tanenbaum 1995, S. 55f). Anwendungssysteme haben im Allgemeinen keine so strenge Schichtung, so dass eine Schicht nicht nur die direkt unter ihr liegende Schicht, sondern mehrere bis alle unter ihr liegenden Schichten verwenden darf. Diese schwächere Form der Schichtung kommt dadurch zustande, dass nicht mehr die Schaffung von höheren Sprachebenen das Ziel ist, sondern die Vermeidung von Aufwärts-Beziehungen zwischen Schichten. Keine der vierzehn Fallstudien mit Schichtenarchitektur, die für diese Arbeit untersucht wurden, hatten eine strenge

[2] www.jwam.de
[3] www.openquasar.de

Schichtung, so dass ich die Unterscheidung zwischen strenger und schwacher Schichtung im weiteren Verlauf vernachlässige. Abbildung 3-5 stellt eine Schichtenarchitektur schematisch dar. Die Schichten und Subsysteme sind als Rechtecke und die Beziehungen als Pfeile dargestellt. Die Schnittstellen werden durch den gestrichelten Bereich in den Subsystemen verdeutlicht.

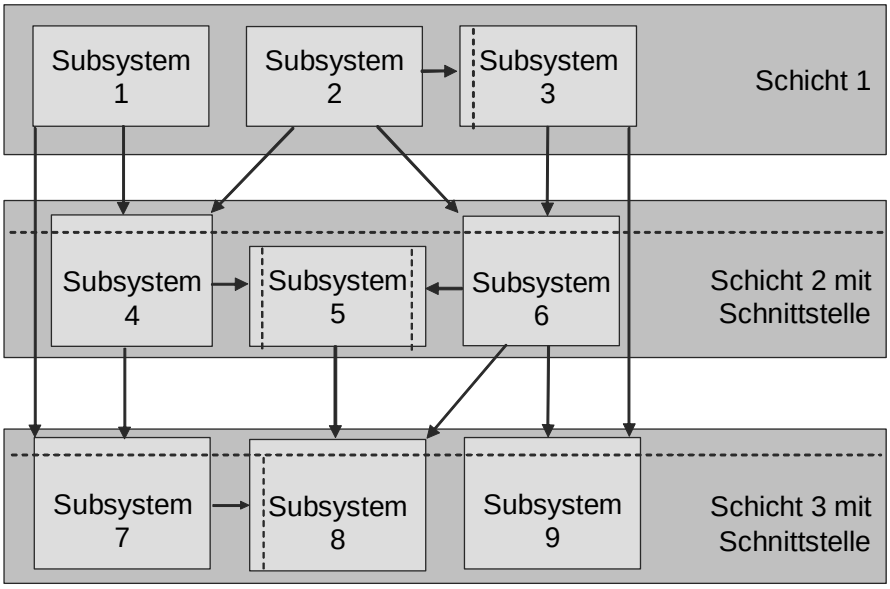

Abbildung 3-5: Schichtenarchitektur

Parnas hat in seinem Artikel „On a ‚Buzzword': Hierarchical Structure" kritisiert, dass viele Autoren von hierarchischer Struktur in ihren Systemen sprechen, aber nicht deutlich sagen, wie das System in Teile zerlegt wird und welche Arten von Beziehungen es gibt (Parnas 1974, S. 337). Für die eben gegebene schematische Abbildung der Schichtenarchitektur ist zwar aufgrund der Einführung zu Beginn dieses Kapitels klar, welche Arten von Beziehungen es gibt (Benutzt- und Vererbungsbeziehung), aber wie das Softwaresystem in Subsysteme und Schichten zerlegt wird, ist in dieser allgemeinen Darstellung unspezifisch. Beliebige Teile des Softwaresystems können zu einem Subsystem und beliebige Subsysteme wiederum zu einer Schicht zusammengefasst werden. Die gleiche Kritik wie Parnas äußern andere Autoren (Evans 2004, S. 109; Siedersleben 2004, S. 44).

Diesem Dilemma begegnen die Vertreter der Schichtenarchitektur zumindest teilweise, indem sie für Schichten bestimmte Aufgaben festlegen. Die Drei-Schichten-Architektur (Three-Tier Architecture) hat beispielsweise die Schichten Oberfläche, Geschäftslogik und Datenhaltung (Brügge und Dutroit 2004, S. 277; Fowler 2003, S. 19f; Züllighoven 2005, S. 299ff) Für die Mehr-Schichten-Architektur (Multi-Tier Architecture) werden verschiedene weitere Schichten vorgeschlagen (Bass et al. 2003, S. 62; Hofmeister et al. 2000, S. 104ff; Jacobson et al. 1999, S. 73; Züllighoven 2005, S. 309ff). Bei diesen Schichtenarchitekturen wird die Zerlegung des Softwaresystems durch semantische Anreicherung der Schichten mit fachlichen Gesichtspunkten herbeigeführt und so Parnas' Kritik zumindest auf Ebene der Schichten begegnet. In den

Fallstudien war der Schnitt von Subsystemen und Schichten immer wieder Thema. Diese Problematik werden ich in Abschnitt 6.4.1 in Zusammenhang mit dem Begriff *Modularität* wieder aufgreifen.

3.3.2. Subsystemarchitektur

Bei den für diese Arbeit durchgeführten Untersuchungen sowie in Praxisvorträgen auf Konferenzen wird häufig ein Architekturstil sichtbar, der in der wissenschaftlichen Literatur im Vergleich zur Schichtenarchitektur wenig Beachtung findet. Ich bezeichne diesen Architekturstil als *Subsystemarchitektur*, weil seine Vorgaben nur die Subsysteme betreffen. Subsystemarchitekturen werden bei großen Softwaresystemen mit Produktlinien und für Service-orientierte Architekturen eingesetzt.

Eine Subsystemarchitektur legt für jedes Subsystem fest, was seine Schnittstelle ausmacht und zu welchen anderen Subsystemen Beziehungen erlaubt oder verboten sind (s. Abbildung 3-6).

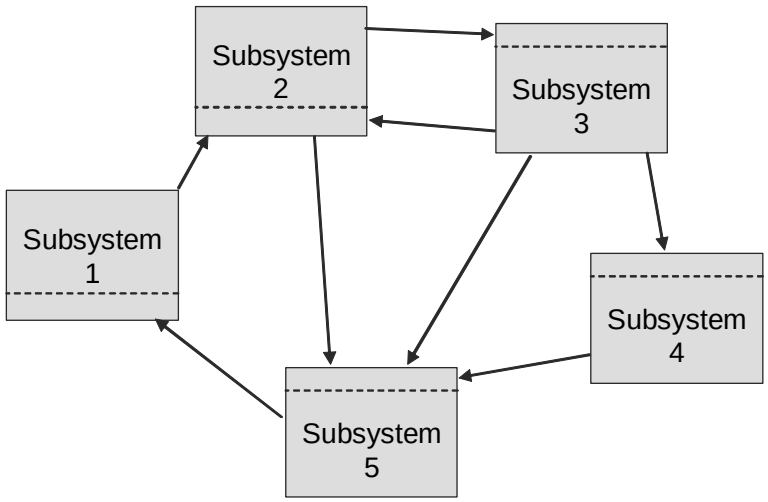

Abbildung 3-6: Subsystemarchitektur

Bei einer Subsystemarchitektur stehen die Dienste, die von den einzelnen Subsystemen bereitgestellt werden, im Mittelpunkt, d. h., die Beschreibung der vom Subsystem angebotenen Klassen und ihr Verhalten auf einer höheren Ebene als Application Programming Interface (API). Ist eine Subsystemarchitektur wirklich detailliert ausgearbeitet, besteht sie aus einer großen Anzahl von Schnittstellen und Erlaubt- und Verboten-Beziehungen zwischen den Subsystemen. Eine Subsystemarchitektur kann in ihrer gesamten Struktur das Bild einer Schichtenarchitektur ergeben. Die Schichtung ist aber nicht zwingend, wie sich bei den drei Fallstudien mit Subsystemarchitektur zeigen wird.

3.3.3. Erweiterte Schichtenarchitektur

Bei großen Softwaresystemen ist die klassische Schichtenarchitektur nicht ausreichend, um alle von ihren Architektinnen und Architekten gewünschten Einschränkungen der Beziehungen zwischen den in den Schichten enthaltenen Subsystemen abzubilden. Sind in einem Softwaresystem Systemteile

enthalten, die verschiedene Sparten in einer Organisation unterstützen, wie Kreditgeschäft, Anlagegeschäft, Wertpapiergeschäft, oder auch übergreifende Funktionalitäten, wie Kundenverwaltung, CRM, anbieten, so sind diese zusammengehörigen Subsysteme auf verschiedene Schichten verteilt (Bäumer et al. 1997, S. 54; Brügge und Dutroit 2004, S. 262; Larman 2004, S. 238).

Für Softwaresysteme mit solchen über mehrere Schichten verteilten, aber zusammengehörigen Subsystemen reicht die durch die Schichtenarchitektur vorgegebene Einschränkung der Beziehungen von oben nach unten nicht aus. Beispielsweise wollen Architektinnen und Architekten bei solchen Softwaresystemen festlegen, dass Subsysteme mit zentralen Funktionalitäten, wie das Kundengeschäft, andere Subsysteme, wie Kreditgeschäft, Anlagegeschäft, nicht benutzen sollen. Die Subsysteme für das Kundengeschäft hingegen dürfen von den Subsystemen anderer übergeordneter Geschäftsfelder verwendet werden. Dieser Architekturstil lässt sich auf zwei Arten umsetzen: Schichtenarchitektur mit Schnitten, Schichtenarchitektur kombiniert mit einer Subsystemarchitektur.

Für die Schichtenarchitektur mit Schnitten werden die Schichten in vertikale Schnitte aufgeteilt, die quer zu den Schichten liegen. Die über die Schichten von oben nach unten verteilten Subsysteme, die zu einem Geschäftsfeld gehören, werden als Einheit betrachtet, und es können Einschränkungen für die erlaubten Beziehungen zwischen Geschäftsfeldern festgelegt werden. Beziehungen zwischen den Schnitten sind genau wie bei den Schichten nur in einer Richtung erlaubt, so dass hier eine Hierarchie der Schnitte entsteht. In Abbildung 3-7 sind Subsysteme und Schichten als Rechtecke zu erkennen. Schnitte habe ich als Rechtecke mit abgerundeten Ecken dargestellt. Bei erweiterten Schichtenarchitekturen haben die Subsysteme sowohl eine Schnittstelle zu Subsystemen in höheren Schichten als auch zu anderen Schnitten, die sie verwenden dürfen.

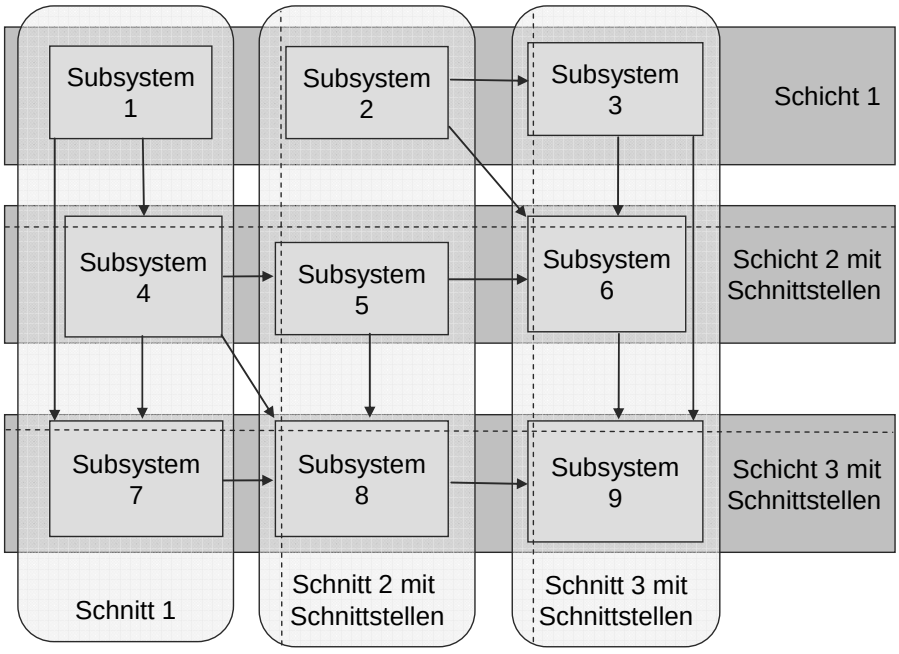

Abbildung 3-7: Erweiterte Schichtenarchitektur

Für den Begriff *Schnitt* gibt es bisher keine allgemein übliche Benennung. Bei Brügge/Dutroit und Larman findet sich der Begriff *Partitionierung* und Züllighoven et al. verwenden den Begriff *Produktbereich* (Brügge und Dutroit 2004, S. 262; Larman 2004, S. 238; Züllighoven 2005, S. 342).

Wählt man anstelle der Schichtenarchitektur mit Schnitten eine Schichtenarchitektur in Kombination mit einer Subsystemarchitektur, so werden die Erlaubt- und Verboten-Beziehungen getrennt von der Schichtenarchitektur auf der gesamten Menge der Subsysteme definiert. Die von der Schichtenarchitektur verbotenen Beziehungen bleiben weiterhin verboten. Dieser Ansatz bietet mehr Freiheit, die Erlaubt- und Verboten-Beziehungen zwischen den Subsystemen verschiedener Sparten und zentraler Funktionen festzulegen. Für Architektinnen und Architekten in großen Projekten werden die kombinierten Schichten- und Subsystemarchitekturen aufgrund ihrer Mächtigkeit allerdings oft unübersichtlich.

3.3.4. Referenzarchitekturen

Für flexiblere Softwaresysteme wurden in den letzten Jahren vermehrt komplexere Architekturstile diskutiert und eingesetzt (vgl. Evans 2004; Fowler 2003; Siedersleben 2004; Züllighoven 2005). Solche Architekturstile definieren Architekturelemente auf der Klassen-Ebene und enthalten Regeln, die sich nicht auf Schichten oder eine festgelegte Menge an Subsystemen und deren Beziehungen und Schnittstellen abbilden lassen (Lilienthal 2007, S. 322). Im

Folgenden werden diese Architekturstile als *Referenzarchitekturen*[4] bezeichnet (Bass et al. 2003, S. 25; Reussner und Hasselbring 2006, S. 358).

Bei einer Referenzarchitektur werden die Architekturelemente auf der Klassen-Ebene in Elementarten, wie Werkzeuge, Entitäten, Value-Objects etc. eingeteilt. Es werden Regeln angegeben, welche Verantwortlichkeiten die einzelnen Elementarten haben und wie die Elementarten miteinander interagieren dürfen.

Grundsätzlich lassen sich drei Arten von Regeln definieren: *Elementregeln*, die nur eine Elementart betreffen, legen Einschränkungen für die Schnittstelle dieser Elementart fest oder spezifizieren, aus welchen Klassen sie aufgebaut ist. *Gebotsregeln* schreiben Beziehungen zwischen bestimmten Elementarten vor, und *Verbotsregeln* verbieten die Beziehungen zwischen bestimmten Elementarten (Becker-Pechau et al. 2006, S. 27; Knodel und Popescu 2007, S. 15). Gebots- und Verbotsregeln betreffen die in Abschnitt 3.2.1 diskutierten Benutzt- und Vererbungsbeziehung zwischen Klassen. Ein Beispiel für eine Verbotsregel ist: „Services dürfen Werkzeuge nicht benutzen“. Durch diese Festlegungen reichern Referenzarchitekturen die Elemente der Klassen-Ebene semantisch an, so dass eine Klasse nicht nur eine einfache Klasse, sondern eine bestimmte Art von Klasse ist. Die Entwicklerinnen und Entwickler müssen die Elementarten und Regeln kennen, um das Zusammenspiel der Klassen zu verstehen und um nur Veränderungen vorzunehmen, die die Regeln nicht verletzen.

Beispiele für Referenzarchitekturen sind Quasar (vgl. Siedersleben 2004), Domain-Driven Design (vgl. Evans 2004) und die WAM-Architektur[5] (vgl. Züllighoven 2005). In den Fallstudien wurden mehrere Softwaresysteme mit WAM-Architektur und ein weiteres mit einer Quasar-Architektur untersucht (s. Tabelle C-1 im Anhang), daher werden die Elementarten und Regeln dieser beiden Referenzarchitekturen im Folgenden vorgestellt.

WAM-Architektur

Der WAM-Ansatz ist eine allgemeine Herangehensweise und Sichtweise für die Softwareentwicklung, der auf Anwendungsorientierung und hohe Gebrauchsqualität der Software ausgerichtet ist (Züllighoven 2005, S. 4ff). Neben einer Anleitung für die Durchführung von iterativen Softwareprojekten mit Prototyping und für die Analyse von Arbeitsabläufen bietet der WAM-Ansatz Leitbilder, Arbeitsplatztypen und Entwurfsmetaphern an, die Entwicklerinnen und Entwickler beim Entwurf unterstützen.

Für die Konstruktion bietet der WAM-Ansatz eine Referenzarchitektur an. Als Elementarten stehen in der WAM-Architektur Fachwerte, Materialien, Services und Werkzeuge zur Verfügung. Werkzeuge sind zusammengesetzte Elementarten und die folgenden Elementarten werden in zwei Varianten

[4] In der deutschsprachigen Literatur werden synonym zu Referenzarchitektur die Begriffe Modellarchitektur und Musterarchitektur verwendet (Reussner und Hasselbring 2006, S. 358; Züllighoven und Raasch 2006, S. 325). In dieser Arbeit wird auf den in der internationalen Literatur üblichen Begriff Referenzarchitektur zurückgegriffen.

[5] WAM steht für Werkzeug-Automat-Material

kombiniert: Gui und Monotool sowie Gui, Interaktionskomponente, Funktionskomponente und Tool. Die WAM-Elementarten und ihre Beziehungen lassen sich schematisch wie folgt darstellen (Becker-Pechau et al. 2006, S. 29):

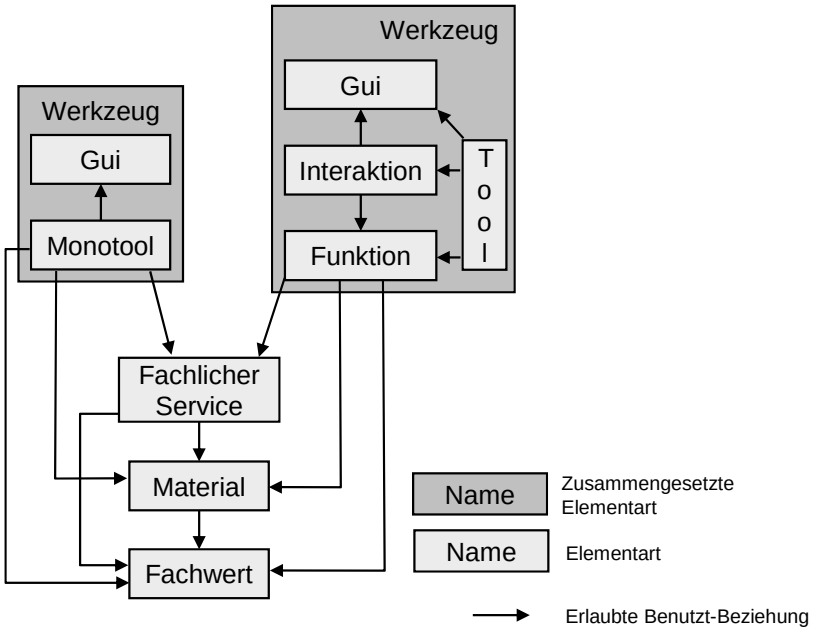

Abbildung 3-8: WAM-Architektur

Die Pfeile in Abbildung 3-8 stehen für die Benutzt-Beziehungen, die in der WAM-Architektur zwischen den Elementarten erlaubt sind. Vererbungsbeziehungen sind in der WAM-Architektur zwischen den Elementarten grundsätzlich verboten und können nur zu allgemeinen Oberklassen oder Klassen aus einem Rahmenwerk, wie dem JWAM-Rahmenwerk, bestehen. Diese Einschränkungen lassen sich über Gebots- und Verbotsregeln abbilden. Wie die Schnittstellen der einzelnen Elementarten aussehen müssen, wird mit Hilfe von Element-Regeln festgelegt (vgl. Becker-Pechau et al. 2006; Knodel und Popescu 2007; Scharping 2006).

Quasar-Architektur

Quasar ist die Standardarchitektur der Firma sd&m (software design & management, München). Der Name Quasar ist die Abkürzung für Qualitätssoftware-Architektur. Quasar wurde von der Forschungsabteilung sd&m-Research aus den bei sd&m durchgeführten Projekten zusammengetragen und wird ständig weiterentwickelt (vgl. Hess et al. 2006; Siedersleben 2004). Quasar ist Opensource und bietet einige Lösungen für Standardprobleme, wie Datenbankanschluss, Rechteverwaltung und Sessionmanagement, als Opensource-Komponenten[6] an.

Das zentrale Anliegen von Quasar ist, Zuständigkeiten zu trennen sowie das Programmieren gegen Schnittstellen und das Denken in Komponenten zu fördern. Die Trennung von Zuständigkeiten wird realisiert, indem die

[6] www.openquasar.de

Software-Komponenten, die Quasar *Subsysteme* nennt, in Kategorien eingeteilt werden, die möglichst wenig vermischt werden dürfen. Die grundsätzlichen Kategorien – metaphorisch auch Blutgruppen genannt – sind A- und T-Software. A-Software wird durch den fachlichen Anwendungszusammenhang bestimmt und ist unabhängig von technischen Aspekten. T-Software hingegen ist allein auf die Technik ausgerichtet: Ein technischer Baustein kennt mindestens eine technische Schnittstelle (API) und ist damit abhängig von ihr (Siedersleben 2004, S. 3). Für A-Software werden in Quasar bestimmte Arten von Klassen und Komponenten definiert, die sich wie folgt darstellen lassen (vgl. Hess et al. 2006; Siedersleben 2004).

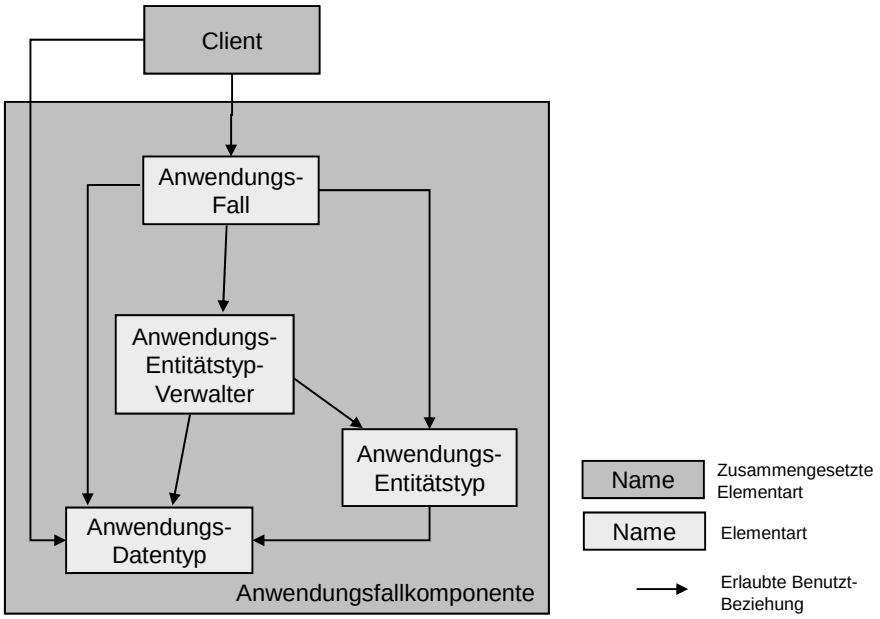

Abbildung 3-9: Quasar-Architektur

Genau wie bei der WAM-Architektur sind bei der Quasar-Architektur nur bestimmte Benutzt-Beziehungen zwischen Elementarten erlaubt, und es sind zusammengesetzte Elementarten vorhanden. Vererbungsbeziehungen zwischen Elementen sind, wie bei der WAM-Architektur, ausgeschlossen. Die erlaubten und verbotenen Beziehungen lassen sich über Gebots- und Verbotsregeln ausdrücken. Für die Elementarten selbst sind Elementregeln vorhanden, die ihre Schnittstellengestaltung beschreiben (vgl. Hess et al. 2006).

Neben den hier vorgestellten Referenzarchitekturen lassen sich in Softwareprojekten *projekteigene Referenzarchitekturen* finden (Kim und Garlan 2006, S. 71f; Scharping 2006, S. 11). Diese projekteigenen Referenzarchitekturen folgen keiner externen Vorgabe aus der Literatur, sondern sind auf der Erfahrung des Entwicklungsteams oder einer gesamten Organisation aufgebaut. Sie legen ebenfalls Arten von Architekturelementen fest und schränken ihre Beziehungen über Regeln ein. Insgesamt vier projekteigene Referenzarchitekturen konnte ich im Rahmen dieser Arbeit analysieren.

Die Quasar-Architektur, die WAM-Architektur und die projekteigenen Referenzarchitekturen unterscheiden sich wesentlich von Entwurfsmustern, weil sie für alle Teile eines Softwaresystems Vorgaben machen (Becker-Pechau et al. 2006, S. 28f; Züllighoven 2005, S. 282). Entwurfsmuster legen zwar auch Regeln für Klassen und ihre Zusammenarbeit fest. Diese Regeln beziehen sich aber immer nur auf einzelne Systemteile und gelten nicht für alle Klassen, aus denen sich das Softwaresystem zusammensetzt (Gamma et al. 1994; Reussner und Hasselbring 2006, S. 342f). Insgesamt ist die Grenze zwischen Architekturstil und Entwurfsmuster aber fließend. Zu der Zeit, als Entwurfsmuster aufkamen, wurden auch Architekturstile als Mustersprache beschrieben (Riehle und Züllighoven 1995). Für diese Arbeit werden Entwurfsmuster als lokale Konstruktionsanleitungen betrachtet, während Architekturstile Vorgaben für die gesamte Struktur von Softwaresystemen machen.

3.3.5. Architekturstile für die Objektorientierung

Lässt man die vier Architekturstile Revue passieren, so fällt auf, dass die ersten drei Architekturstile: Schichtenarchitektur, Subsystemarchitektur und Erweiterte Subsystemarchitektur Vorgaben für die Subsystem-Ebene der Architektur machen. Referenzarchitekturen, egal ob WAM, Quasar oder im Projekt selbst entwickelt, haben ihren Schwerpunkt hingegen auf der Klassen-Ebene. Ansätze dieser Differenzierung lassen sich bei Garlan finden, der in Bezug auf Architekturstile sagt, dass es zwei Arten von Architekturstilen gibt: Solche, die Aussagen über die grundsätzliche Organisation eines Softwaresystems machen, und solche, die feingranulare Vorgaben für die Konfiguration von Architekturelementen und ihren Beziehungen machen (Garlan et al. 1994, S. 176).

Ob diese unterschiedliche Wirkungsebene der Architekturstile Auswirkungen auf die Komplexität der mit ihnen entwickelten Architektur hat, wird im weiteren Verlauf dieser Arbeit eine wichtige Frage sein, die durch die Fallstudien beantwortet werden kann.

An dieser Stelle darf allerdings nicht verschwiegen werden, dass in den Veröffentlichungen zur WAM- und zur Quasar-Architektur neben den Vorgaben für die Klassen-Ebene auch Regeln für die Subsystem-Ebene, wie der Einsatz einer Schichtenarchitektur, vorgeschlagen werden. Große Softwaresysteme können allein mit den Regeln einer Referenzarchitektur kaum entwickelt und gewartet werden, und Architekturstile für die Subsystem-Ebene müssen hinzugenommen werden. In den Fallstudien hat sich jedoch gezeigt, dass es zum einen Projekte gibt, die tatsächlich nur die Regeln für die Klassen-Ebene der Architektur aus einer Referenzarchitektur einsetzen; zum anderen wurde deutlich, dass Referenzarchitekturen in Kombination mit Schichten- oder Subsystemarchitekturen andere Auswirkungen haben als Schichten- bzw. Subsystemarchitekturen allein.

3.4. Soll- und Ist-Architektur

Will man die Architektur eines Softwaresystems untersuchen, so stehen verschiedene Quellen zur Verfügung. Zum einen hat das Entwicklungsteam eine Architektur geplant und sie in Dokumenten festgehalten oder informell abgesprochen. Zum anderen hat das Entwicklungsteam das Softwaresystem implementiert und dabei die geplante Architektur umgesetzt. Die vom Entwicklungsteam geplante Architektur bezeichne ich im Folgenden als *Soll-Architektur* und die im Softwaresystem implementierte Architektur als *Ist-Architektur* (Simon und Meyerhoff 2002, S. 35).

Die eigentliche Komplexität eines Softwaresystems findet sich in seiner Ist-Architektur, die aus dem Programmtext abgeleitet werden muss. Um die Ist-Architektur zu analysieren, werden aus dem Programmtext die vorhandenen Architekturelemente, wie Packages, Klassen und ihre Beziehungen, extrahiert, so dass ein Modell des Programmtextes zur Verfügung steht (Murphy et al. 2001, S. 365; Simon und Meyerhoff 2002, S. 29).

Außerdem wird eine *Abbildungsvorschrift* benötigt, die alle Architekturelemente der Soll-Architektur und die Elementarten des Architekturstils auf entsprechende Klassen oder Teilbereiche des Programmtextes überträgt. In Java ist mit den Packages nur eine schwache Möglichkeit vorhanden, Subsysteme im Programmtext abzubilden. Die für Architekturstile benötigten Ausdrucksmittel, wie Schichten, Schnitte, Schnittstellen von Subsystemen oder Elementarten aus Referenzarchitekturen, sind gar nicht vorhanden. Die Abbildung der Soll-Architektur und des Architekturstils auf die Ist-Architektur kann daher nur im Programmtext über Konventionen sichtbar gemacht werden.

Eine übliche Konvention ist, die Subsysteme einer Architektur in Java im Package-Baum des Softwaresystems nachzubilden. Abbildung 3-10 zeigt einen Ausschnitt des Package-Baum vom OpenSource-System JCommSy[7], das an der Universität Hamburg entwickelt wurde. Unterhalb des Package-Knotens `jcommsy` sind bei diesem Softwaresystem Subsysteme angeordnet. Jedes Subsystem umfasst einen Package-Knoten und den sich unter ihm aufspannenden Teilbaum. In Abbildung 3-10 sind die zu einem Subsystem gehörenden Teilbäume farblich gekennzeichnet.

[7] www.commsy.net

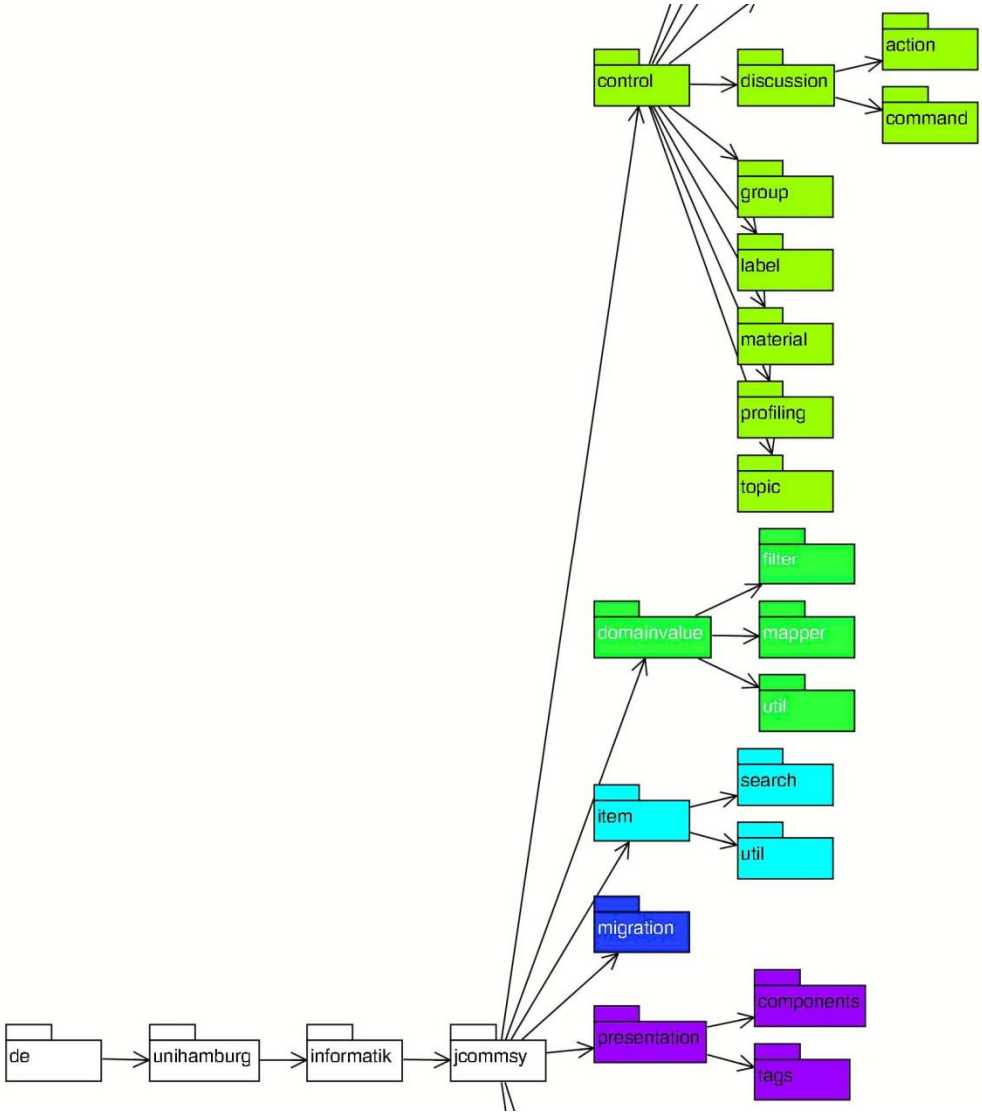

Abbildung 3-10: Package-Baum mit Subsystemen

Packages können in Java analog zur Verzeichnisstruktur ineinander geschachtelt werden, so dass, wie in Abbildung 3-10 dargestellt, ein hierarchischer Baum entsteht. Diese Schachtelung hat auf der programmiersprachlichen Ebene allerdings keine der Enthaltenseins-Beziehung vergleichbare Bedeutung. Der Package-Mechanismus von Java wurde so implementiert, dass ein Package zwar einen eigenen Namensraum für die in ihm enthaltenen Klassen eröffnet und den Zugriff auf diese Klassen einschränken kann. Auf die im Package-Baum unter ihm angesiedelten Packages beziehen sich diese Mechanismen aber nicht.

Diese durch den Package-Baum unterstützte Konvention zur Abbildung von Subsystemen aus der Soll-Architektur auf den Programmtext ist eine Möglichkeit einer Abbildungsvorschrift. Für andere Architekturelemente der Soll-Architektur und des Architekturstils müssen weitere Abbildungsvorschriften abgeleitet werden.

Mithilfe der Abbildungsvorschrift können die Architekturelemente der Soll-Architektur und des Architekturstils schließlich den Modellelementen, die aus dem Programmtext extrahiert wurden, zugeordnet werden (s. Abbildung 3-11).

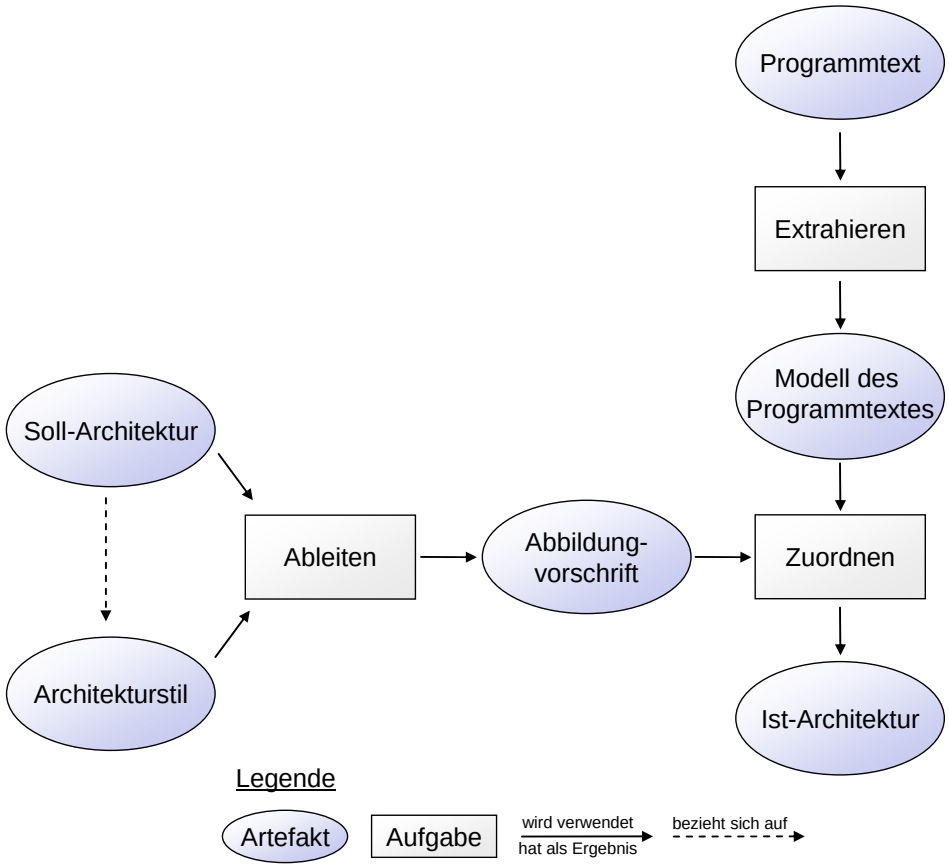

Abbildung 3-11: Erarbeiten der Ist-Architektur

In der englischsprachigen Literatur werden das Modell des Programmtextes als source model, source view oder concrete (module) view und die Abbildungsvorschrift als mapping oder target view bezeichnet. Für die Soll-Architektur werden die Begriffe high-level model, architectural model, hypothesized (module) view verwendet (Becker-Pechau und Bennicke 2007, S. 1f; Deursen et al. 2004, S. 124; Knodel und Popescu 2007, S. 13; Koschke und Simon 2003, S. 37; Murphy et al. 1995, S. 19).

Hat man die Ist-Architektur eines Softwaresystems mit Hilfe des in Abbildung 3-11 dargestellten Verfahrens konstruiert, so wird es möglich, die Ist-Architektur mit der Soll-Architektur und dem Architekturstil zu vergleichen und Differenzen zu identifizieren. In keiner der Fallstudien entsprach die Ist-Architektur der Soll-Architektur bzw. wurden alle Regeln des Architekturstils eingehalten.

Die Ursachen hierfür sind vielfältig: Fachliche und technische Details wurden bei der Planung der Soll-Architektur übersehen, und konkurrierende Qualitätsziele, wie Performanz und Lose Kopplung, werden erst spät im Entwick-

lungsprozess offensichtlich (Bass et al. 2007, S. 17ff). Die essentielle Komplexität des Problems und der Lösung wurde unterschätzt (s. Abschnitt 2.4.2) und macht Änderungen in der Architektur notwendig. Hinzu kommen der für Entwicklungsprojekte übliche Zeitdruck, aber auch Missverständnisse oder Unkenntnis über die Soll-Architektur, die dazu führen, dass die Ist- Architektur von der Soll-Architektur und dem Architekturstil abweicht. In das Softwaresystem werden beispielsweise Beziehungen zwischen Subsystemen eingebaut, die Schnittstellen missachten oder der Schichtung des Softwaresystems zuwiderlaufen. Oder, Programmtext wird kopiert, anstatt über Abstraktionen und Wiederverwendung nachzudenken.

Dieses Auseinanderlaufen von Ist-Architektur, Soll-Architektur und Architekturstil erzeugt für das Entwicklungsteam Komplexität. Zum Teil ist diese Komplexität essentiell, weil die Soll-Architektur zu unscharf war und eine Anpassung der Soll-Architektur notwendig wird. Zum Teil ist sie akzidentell, weil das Entwicklungsteam die Vorgaben der Soll-Architektur oder des Architekturstils verletzt hat, obwohl eine andere Implementierung möglich gewesen wäre.

3.5. Zusammenfassung zur Architekturkomplexität

In diesem Kapitel habe ich die Grundlage für die Analyse und Beurteilung von Architekturkomplexität gelegt. Als erstes habe ich die Elemente objektorientierter Architekturen auf Klassen- und Subsystem-Ebene vorgestellt. Anschließend habe ich vier Architekturstile beschrieben und die Unterscheidung zwischen Soll- und Ist-Architektur eingeführt. Diese verschiedenen Aspekte und ihre Komplexität fasse ich nun in einer Definition zu Architekturkomplexität zusammen.

Definition 3-2: Architekturkomplexität

> *Architekturkomplexität* ist der strukturelle Anteil der lösungsabhängigen Komplexität, der sich aus der Wahl der Architekturelemente und ihrer Beziehungen untereinander ergibt.
>
> Architekturkomplexität speist sich aus zwei Quellen: der *Struktur der Ist-Architektur* und der *Abbildung von Soll-Architektur und Architekturstil auf den Programmtext*.

Die *Struktur der Ist-Architektur* besteht aus einer Vielzahl von Klassen mit Beziehungen, die zu Subsystemen zusammengefasst werden. Aufgrund der großen Anzahl der Elemente ist die Ist-Architektur grundsätzlich schwer zu verstehen. Je nachdem, welchen Architekturstil und welche Soll-Architektur das Entwicklungsteam gewählt hat, werden die einzelnen Klassen und Subsysteme der Ist-Architektur unterschiedlich komplex und unterschiedlich stark vernetzt sein.

Außerdem muss von Entwicklerinnen und Entwicklern eine *Abbildung von Soll-Architektur und Architekturstil auf den Programmtext* vorgenommen werden, damit sie sich bei ihrer Arbeit zurechtfinden. Diese Aufgabe ist an sich

komplex, weil die Elemente der Soll-Architektur und des Architekturstils nur mithilfe einer Abbildungsvorschrift in der Ist-Architektur zu identifizieren sind. Diese Abbildung wird aber noch erschwert, wenn die Ist-Architektur von der Soll-Architektur und dem eingesetzten Architekturstil abweicht. In diesem Fall muss das Entwicklungsteam nicht nur die Soll-Architektur und den Architekturstil auf die Ist-Architektur übertragen, sondern zusätzlich mit den Differenzen zwischen beiden umgehen.

Im nächsten Kapitel werde ich besonders den ersten Aspekt von Architekturkomplexität weiter verfolgen, indem ich Ergebnisse der kognitiven Psychologie zu Rate ziehe, die Aussagen darüber machen, wie Menschen mit komplexen Strukturen umgehen. Im Lichte dieser Erkenntnisse wird deutlich werden, wie objektorientierte Architekturen und Architekturstile zu geringer Architekturkomplexität beitragen. Der zweite Aspekt der Übertragbarkeit von Soll-Architektur und Architekturstil auf die Ist-Architektur und die Gefahr des Auseinanderdriftens wird in Kapitel 6 wieder aufgenommen.

4. Komplexe Strukturen aus Sicht der kognitiven Psychologie

Objektorientierte Programmiersprachen, Entwurfsmuster und Architekturstile werden in Ausbildung und Praxis eingesetzt, weil eine Reihe von Entwicklerinnen und Entwicklern sie als hilfreich erfahren haben. Dieses Wissen über „best practises" ist in der Regel nicht wissenschaftlich belegt, sondern basiert auf Erfahrungen aus Projekten. Einen Beweis antreten zu wollen, dass bestimmte „best practises" zu weniger komplexen Softwaresystemen führen, ist unter den vielfältigen Randbedingungen, die Softwareprojekte beeinflussen, sehr schwierig.

Aus diesem Grund gewinnen die Erkenntnisse der kognitiven Psychologie für diese Arbeit an Bedeutung. Die kognitive Psychologie befasst sich mit menschlichen Vorgehensweisen beim Verstehen, beim Wissenserwerb und bei der Problemlösung – alles Aufgaben, mit denen Entwicklerinnen und Entwickler beim Umgang mit Architekturkomplexität konfrontiert sind. Die hier vorgestellten Erkenntnisse fassen zusammen, was Menschen bei der Strukturierung von komplexen Zusammenhängen im alltäglichen Leben tun. Mit ihrer Hilfe lässt sich argumentierten, welche Strukturen für Menschen leichter zu verstehen sind als andere. Der eine Anteil der im letzten Kapitel beschriebenen Architekturkomplexität – die in der Struktur der Ist-Architektur enthalte Komplexität – kann auf der Basis dieser Erkenntnisse zwar nicht aufgelöst werden, aber es sind Empfehlungen möglich, wie dieser Anteil der Architekturkomplexität wenigstens gemindert werden kann.

Für diese Arbeit sind von besonderer Bedeutung drei vom Menschen angewendete strukturbildende Prozesse: Das mit *Chunking* bezeichnete Zusammenfassen von kleineren zu größeren Einheiten, die *Bildung von Hierarchien* und der *Aufbau von Schemata* zur Abbildung von allgemeinen Kategorien. Im Folgenden stelle ich die für diese Arbeit relevanten Untersuchungen vor und reichere die Erläuterungen um Erkenntnisse aus der Informatik an. Ich schließe das Kapitel mit einem Zwischenfazit ab, mit dem ich die Erkenntnisse zu Chunking, der Bildung von Hierarchien und dem Aufbau von Schemata auf Architekturkomplexität beziehe.

4.1. Chunking

Damit Menschen in der Menge der Informationen, mit denen sie konfrontiert sind, zurechtkommen, müssen sie auswählen und Teilinformationen zu größeren Einheiten gruppieren. Dieses Bilden von höherwertigen Abstraktionen, die immer weiter zusammengefasst werden, nennt man *Chunking* (Anderson 1989, S. 134ff; Davis 1984, S. 119ff; Henderson-Sellers 1996, S. 169ff; Kluwe 1992, S. 144ff; Miller 1956).

Beim Chunking werden neue Wissenseinheiten (Chunks) aus mehreren, vorher separaten im Gedächtnis gespeicherten Wissenseinheiten gebildet. Als Beispiel soll hier eine Person dienen, die das erste Mal mit einem Telegrafen

arbeitet. Sie hört die übertragenen Morsezeichen als kurze und lange Töne und verarbeitet sie am Anfang als getrennte Wissenseinheiten. Nach einiger Zeit wird sie in der Lage sein, die Töne zu Buchstaben – und damit zu neuen Wissenseinheiten – zusammenzufassen, so dass sie schneller verstehen kann, was übermittelt wird. Einige Zeit später werden aus einzelnen Buchstaben Wörter, welche wiederum größere Wissenseinheiten darstellen, und schließlich ganze Sätze.

Dieser Vorgang wird auch als *Recoding* bezeichnet (Boisot und Child 1999, S. 239; Kluwe 1992, S. 144f). Recoding läuft folgendermaßen ab: Am Anfang liegen Wissenseinheiten in einem Code vor, der viele Wissenseinheiten mit wenig Information pro Wissenseinheit enthält. Diese Wissenseinheiten werden in einen neuen Code überführt, der mehr Information pro Wissenseinheit enthält. Bei Recoding werden also vorhandene Wissenseinheiten verdichtet und als eine neue Einheit abgespeichert. Die einfachste Art des Recodings ist, mehrere Wissenseinheiten zu gruppieren, der Gruppe einen Namen zu geben und sich die neue Wissenseinheit unter diesem Namen zu merken. Die resultierende, neue Wissenseinheit gilt als ein Element, und die in ihr zusammengefassten Wissenseinheiten werden vom Gehirn gemeinsam aktiviert und verarbeitet. Da die Kapazität des Kurzzeitgedächtnisses begrenzt ist – in der Regel werden als Obergrenze 7 (+/-2) Wissenseinheiten angenommen –, ist Chunking ein wichtiger Vorgang der Kapazitätsentlastung.

Recoding darf man sich allerdings nicht als einen diskreten Prozess aus Einzelschritten vorstellen. Vielmehr wird ein kontinuierlicher Lernprozess durchlaufen, bei dem der erreichte Grad des Recodings für die vorhandenen Wissenseinheiten unterschiedlich weit fortgeschritten ist und sich überlappt.

Entwicklerinnen und Entwickler, die sich unbekannte Programme erschließen müssen, wenden Chunking automatisch an. Der Programmtext wird im Detail gelesen, und die gelesenen Zeilen werden zu Wissenseinheiten gruppiert und so behalten. Die Wissenseinheiten werden immer weiter zusammengefasst, bis ein Verständnis des benötigten Programmtextes erreicht ist. Diese Herangehensweise an Programme wird als *Bottom-Up Programmverstehen* bezeichnet und wird von Entwicklerinnen und Entwicklern in der Regel anwendet, wenn ihnen ein Softwaresystem und sein Anwendungsgebiet unbekannt ist und sie sich das Verständnis erst erarbeiten müssen. Bei Kenntnis des Anwendungsgebiets und des Softwaresystems wird von Entwicklerinnen und Entwicklern eher Top-down Programmverstehen eingesetzt. *Top-Down Programmverstehen* bedient sich hauptsächlich der strukturbildenden Prozesse Bildung von Hierarchien und Aufbau von Schemata, die ich in den folgenden Abschnitten einführen werde (Mayrhauser und Vans 1997, S. 158ff; Storey et al. 1999, S. 173ff; Wallnau et al. 1996, S. 178f).

Eine andere Chunking-Variante konnte an *Experten* beobachtet werden (Anderson 1989, S.228; Kluwe 1992, S. 145). Bei dieser Variante werden die neuen Wissenseinheiten nicht einzeln im Kurzzeitgedächtnis abgespeichert und dann verdichtet, sondern direkt durch Aktivierung bereits gespeicherter

Wissenseinheiten zusammengefasst. Experten zeichnen sich dadurch aus, dass ihre Wissenseinheiten deutlich mehr Informationen verdichten als jene von unerfahrenen Problemlösern. Bei einem Experiment wurden Schachmeistern und Schachanfängerinnen und -anfängern für ca. fünf Sekunden Spielstellungen auf einem Schachbrett gezeigt. Handelte es sich um eine sinnvolle Aufstellung der Figuren, so waren die Schachmeister in der Lage, die Positionen von mehr als zwanzig Figuren zu rekonstruieren, während die schwächeren Spielerinnen und Spieler nur die Position von vier oder fünf Figuren rekonstruieren konnten. Handelte es sich um eine beliebige Verteilung der Figuren, so waren die Schachmeister nicht im Vorteil. Sie konnten die für sie sinnlose Verteilung der Figuren nicht zu ihnen bekannten größeren Wissenseinheiten verdichten.

An dieser Untersuchung wird ein wichtiger Aspekt beim Chunking deutlich. Wissenseinheiten können nur aus anderen Wissenseinheiten gebildet werden, die für die Versuchsperson sinnvoll zusammengehören. Dieser Aspekt wird an dem Versuch mit den Schachaufstellungen deutlich und konnte ebenfalls in Experimenten mit sinnvollen und sinnlosen Wortgruppen festgestellt werden (Anderson 1989, S. 135). An Entwicklerinnen und Entwicklern lassen sich diese Erkenntnisse ebenfalls nachweisen. Bei entsprechenden Untersuchungen wurde deutlich, dass es Entwicklerinnen und Entwicklern umso leichter fällt, die untersuchten Teile der Software zu verstehen, je stärker die Programmiersprache und die Struktur eines Softwaresystems sinnvoll zusammenhängende Einheiten anbietet, die zu einer höherwertigen Abstraktion zusammengefasst werden können. Für das Chunking ist allerdings nicht die Nähe der Einheiten entscheidend, sondern ob sie sinnvoll zusammenhängen. Programmeinheiten wie Klassen oder Module, die beliebige Operationen zusammenfassen, so dass für die Entwicklerinnen und Entwickler nicht erkennbar ist, warum sie zusammengehören, lassen sich nicht in Wissenseinheiten codieren (Mayrhauser und Vans 1997, S. 158; Wallnau et al. 1996, S. 179).

4.2. Bildung von Hierarchien

Untersuchungen in verschiedenen Teilbereichen der kognitiven Psychologie machen deutlich, dass *Hierarchien*[8] beim Wahrnehmen und Verstehen von komplexen Strukturen und beim Abspeichern von Wissen eine wichtige Rolle spielen. Als hierarchische Strukturen werden in den Experimenten Bäume (Monohierarchie) oder gerichtete azyklische Graphen (Polyhierarchie) betrachtet. Für die Ordnung der Elemente wurden unterschiedliche Relationen verwendet: kategoriale Unterordnung, wie sie bei Ober- und Unterbegriffen üblich ist (Vererbungs-Hierarchie), Teil-Ganzes-Beziehung, wie sie für die Bildung von größeren Einheiten aus kleineren Einheiten verwendet wird (Enthaltenseins-Hierarchie), oder auch Gedankenkarten (engl. mind map), bei denen Begriffe in Beziehungen stehen (Benutzt-Beziehung). Die verschiedenen

[8] Das Wort Hierarchie stammt aus dem Griechischen: ιεραρχία, was wiederum zusammengesetzt ist aus ιερή, "hieré" - heilige und αρχή, "arché" - Herrschaft, Ordnung, Prinzip.

Autoren stellen fest, dass Menschen dann Wissen gut aufnehmen, es wiedergeben und sich darin zurecht finden können, wenn es in hierarchischen Strukturen vorliegt (Anderson 1989, S. 172ff; Kluwe 1992, S. 166).

Als Beispiele führen die Autoren Untersuchungen zum Lernen von zusammengehörenden Wortkategorien, zur Organisation von Lernmaterialien, zum Textverstehen, zur Textanalyse und zur Textwiedergabe an. Bei der Reproduktion von langen Begriffslisten war die Gedächtnisleistung der Versuchspersonen deutlich höher, wenn ihnen Entscheidungsbäume mit kategorialer Unterordnung angeboten wurden. Lerninhalte wurden von den Versuchspersonen mit Hilfe von hierarchischen Kapitelstrukturen oder Gedankenkarten deutlich schneller gelernt (Anderson 1989, S. 174, 336; Kluwe 1992, S. 167). Lag keine hierarchische Struktur vor, so bemühten sich die Versuchspersonen, den Text selbständig hierarchisch anzuordnen. Bei der Wiedergabe von in Hierarchien geordneten Textstrukturen wurde festgestellt, dass Informationen, die in der Textstruktur auf höherer Ebene liegen, gewöhnlich besser erinnert und damit weiterverarbeitet wurden, als solche, die sich in der Textstruktur weiter unten befinden. Das Textverständnis hing dabei entscheidend davon ab, ob die Versuchspersonen durch den Text unterstützt wurden, die hierarchisch höher stehenden, den Text organisierenden Strukturen zu erkennen (Anderson 1989, S. 313).

Aus diesen Untersuchungen wird für die Strukturierung von Wissen die Konsequenz gezogen, dass hierarchisch geordnete Inhalte für Menschen leichter zu erlernen und zu verarbeiten sind und dass aus einer hierarchischen Struktur effizienter Inhalte abgerufen werden können (Anderson 1989, S. 174; Kluwe 1992, S. 167). Darüber hinaus werden Hierarchien und das bereits eingeführte Chunking vom Menschen oft in Kombination eingesetzt, indem die Wissenseinheiten im Gedächtnis mithilfe von hierarchischen Strukturen abgebildet werden. Ein Haus kann beispielsweise als eine Wissenseinheit abgespeichert werden und jeweils kleinere Wissenseinheiten für die einzelnen Elemente des Hauses enthalten (Anderson 1989, S. 101; Kluwe 1992, S. 146).

Die Vorteile von Hierarchien wurden ebenfalls in den Untersuchungen zum *Top-Down Programmverstehen* deutlich, bei denen festgestellt wurde, dass Entwicklerinnen und Entwickler Hierarchien verwenden, um Softwaresysteme auf verschiedenen Ebenen zu betrachten, indem sie Teilbereiche des Softwaresystems und seiner Struktur ausblenden (Mayrhauser und Vans 1997, S. 158). Waren den Entwicklerinnen und Entwicklern die Hierarchien des Softwaresystems bekannt, so konnten sie die Wartungsarbeiten schneller erledigen als die Entwicklerinnen und Entwickler, die sich das unbekannte Softwaresystem über Bottom-Up Programmverstehen erst erarbeiten mussten (Storey et al. 1999, S. 173ff; Wallnau et al. 1996, S. 179).

Disziplinübergreifend wurde das Thema Bildung von Hierarchien schon frühzeitig von Herbert A. Simon untersucht. Dabei steht nicht die menschliche Wissensverarbeitung im Vordergrund, sondern es werden verschiedene Arten von Systemen und ihre Strukturierung betrachtet.

Simon (Simon 1994, S. 146ff) diskutiert eine Vielzahl von Eigenschaften komplexer Systeme in Natur, Wissenschaft und Gesellschaft und kommt zu dem Schluss, dass komplexe Systeme häufig als Hierarchie erscheinen und aus stabilen Teilsystemen bestehen. Simon definiert eine Hierarchie als ein System, das aus untereinander verbundenen Subsystemen zusammengesetzt ist, wobei jedes der letzteren wiederum eine hierarchische Struktur aufweist, bis man zu einer untersten Schicht elementarer Subsysteme gelangt. Simon vertritt dabei die Ansicht, dass Hierarchie die Form ist, in der komplexe Systeme hauptsächlich in der Welt erscheinen. Dabei wirft er die Frage auf, ob Menschen Systeme, die nicht hierarchisch organisiert sind, überhaupt wahrnehmen und verarbeiten können.

Luhmann ist nicht Simons Meinung und weist mit Belegen aus verschiedenen Bereichen darauf hin, dass die Hierarchie ein Sonderfall einer Systemstruktur ist und dass man nicht davon ausgehen kann, dass „die Evolution Komplexität mehr oder weniger zwangsläufig in die Form von Hierarchien bringt“ (Luhmann 1987, S. 39). Als einen Beleg führt Luhmann den in der Informatik oft zitierten Architekten Christopher Alexander an, der in seinem Artikel „A City is not a Tree“ deutlich macht, dass Städte nicht als Bäume abbildbar sind (Alexander 1982, S. 397f). Allerdings stellen sowohl Luhmann als auch Alexander fest, dass die Hierarchie ein gutes Mittel ist, um Systeme zu vereinfachen. Luhmann zitiert an dieser Stelle Howard H. Pattee mit den Worten: „hierarchical constraints as self-simplification of initially chaotic, very complex systems” (Luhmann 1987, S. 39).

Zu Simons Meinung und Luhmanns und Alexanders Einschränkung passen die Aussagen der kognitiven Psychologie, die ich bisher zusammengetragen habe: Systeme sind nicht automatisch hierarchisch, aber Menschen bevorzugen hierarchische Strukturen.

4.3. Aufbau von Schemata

Ein weiteres Mittel, das von Menschen eingesetzt wird, um komplexe Zusammenhänge zu strukturieren, sind Schemata. Unter einem Schema werden Wissenseinheiten verstanden, die sowohl abstrahiertes als auch konkretes Wissen umfassen. Schemata enthalten auf der abstrakten Ebene typische Eigenschaften der von ihnen schematisch abgebildeten Zusammenhänge und legen für diese Eigenschaften Wertebereiche fest. Auf der konkreten Ebene beinhaltet ein Schema eine Reihe von Exemplaren, die prototypische Ausprägungen des Schemas darstellen (Anderson 1989, S. 120ff; Dutke 1994, S. 22ff; Kluwe 1992, S. 155ff; Norman 1982, S. 51ff).

Wenn wir beispielsweise darüber informiert werden, dass eine Person den Beruf eines Lehrers ausübt, dann enthält unser Lehrer-Schema auf der abstrakten Ebene verschiedene Annahmen und Vorstellungen hinsichtlich seiner Tätigkeit: Lehrer sind an einer Schule beschäftigt, haben keinen 8-Stunden-Tag, geben Unterrichtstunden von 45 Minuten Länge und müssen Arbeiten korrigieren. Konkret werden wir andere Lehrer kennen, die wir als Prototypen des Lehrer-Schemas abgespeichert haben. Aus unserem Wissen

über diese Prototypen werden wir ebenfalls Rückschlüsse auf die andere Person ziehen.

Die in der Softwareentwicklung vielfältig eingesetzten Entwurfsmuster (vgl. Gamma et al. 1994) nutzen die Stärke des menschlichen Gehirns, mit Schemata zu arbeiten. Das für Entwurfsmuster entwickelte Beschreibungsschema enthält eine abstrahierte Darstellung des Musters mit verschiedenen Varianten der Umsetzung, die als prototypische Ausprägungen verstanden werden können. Haben Entwicklerinnen oder Entwickler bereits mit einem Entwurfsmuster gearbeitet und daraus ein Schema gebildet, so können sie Programmtexte und Strukturen schneller erkennen und verstehen, die dieses Entwurfsmuster einsetzen.

In der Entwurfsmusterdiskussion wird immer wieder darauf hingewiesen, dass Entwurfsmuster nicht erfunden, sondern entdeckt und aus Erfahrung gewonnen werden. Genau die gleiche Erkenntnis wurde in der kognitiven Psychologie gewonnen: Schemata lassen sich nur durch Erfahrungen herausbilden und durch neue ähnliche Erfahrungen verstärken und verfeinern. Die Änderung von definierenden Merkmalen eines Schemas ist in der Regel schwierig, weil sie das gesamte Schema in Frage stellt (Anderson 1989, S. 122). Beispielsweise war es kurz nach der Einführung von mobilen Telefonen für mich überraschend, wenn ich von einem einzelnen, einen Monolog haltenden Menschen überholt wurde. Lief hinter mir jemand, der sprach, so erwartete ich, dass ich von zwei oder mehr Personen, die miteinander ein Gespräch führten, überholt werde. Diese Erfahrung musste ich mehrere Male machen, bis sich mein Schema angepasst hatte und einzelne redende Menschen mit einem Mobiltelefon am Ohr keine Überraschung mehr darstellten.

Hat ein Mensch auf einem Gebiet umfassende Erfahrung gemacht, so bilden sich zwischen den einzelnen Schemata dieses Gebiets hierarchische Beziehungen heraus. Dabei werden zwei verschiedene Arten von Hierarchien unterschieden: Teil-Ganzes-Hierarchien und Generalisierungshierarchien (Anderson 1989, S. 101, 121). Bei einer Teil-Ganzes-Hierarchie sind Schemata auf einer Ebene derart codiert, dass sie sich aus Unterschemata zusammensetzen. Das Schema für ein Haus setzt sich zum Beispiel aus Schemata für Zimmer, Stockwerke etc. zusammen (Anderson 1989, S. 101) Bei der Generalisierungshierarchie werden Schemata in Bezug auf Ober- und Unterbegriffe aufgebaut. Für das Haus-Schema bedeutet dies, dass es neben der Aufteilung in Unterschemata über die Generalisierung mit dem Gebäude-Schema verknüpft ist (Anderson 1989, S. 121). Auf diese Weise ist der Mensch in der Lage, parallele Hierarchien von Teil-Ganzes und Generalisierung aufzubauen.

Für das Top-Down Programmverstehen sind diese Schachtelungen und Abstraktionen von Schemata von entscheidender Bedeutung. Auf diese Weise werden für verschiedene Abstraktionsebenen eines Softwaresystems jeweils die passenden Schemata erkannt und können in die nächst niedrigere Ebene verfeinert bzw. in die nächst höhere Ebene abstrahiert werden.

Mithilfe einer Gruppe von Schemata zu einem bestimmten Sachverhalt ist ein Experte in dem entsprechenden Gebiet in der Lage, sich schneller zurechtzufinden, als ein Neuling, der über keine entsprechenden Schemata verfügt. Dieser Geschwindigkeitsgewinn kommt dadurch zustande, dass ein Experte durch den Einsatz von Schemata Gedächtnisinhalte direkt abrufen kann, um Muster in seiner Umwelt zu erkennen. Diese Möglichkeit unterscheidet ihn wesentlich vom Laien, der ohne entsprechende Schemata gezwungen ist, durch eine deduktive Herangehensweise Wissenseinheiten zu bilden und eigene Schemata zu entwickeln. Deduktives Verstehen von Zusammenhängen führt zu einer seriellen Verarbeitung im Gehirn, für die das menschliche Gehirn nach Einschätzung verschiedener kognitiver Psychologen weniger geeignet ist. Besser geeignet ist das Gehirn für parallele Verarbeitung, bei der sich Informationen ausbreiten und Muster erkannt werden. Die Entwicklung von Sachkenntnis auf einem Gebiet geht dementsprechend einher mit einem Übergang von deduktiven Lösungsstrategien zu Lösungsstrategien wie Mustererkennung, an die das menschliche Gehirn besser angepasst ist (Anderson 1989, S. 218, 240f).

Anhand der in den letzten drei Abschnitten angeführten Untersuchungen wurde erläutert, wie Menschen Chunking, die Bildung von Hierarchien und den Aufbau von Schemata einsetzen, um Softwaresysteme top-down und bottom-up zu verstehen. Man geht davon aus, dass Top-Down und Bottom-Up Programmverstehen von Experten i. d. R. in Kombination eingesetzt werden (Wallnau et al. 1996, S. 179). Die heute entwickelten und im Rahmen dieser Arbeit untersuchten Softwaresysteme haben eine Größe erreicht, die es den Mitgliedern von Entwicklungsteams unmöglich macht, Experten für alle Bereiche des von ihnen gewarteten Softwaresystems zu sein und passende Hierarchien, Schemata und Chunks vorrätig zu haben. Neben der Größe machen es die bei der Wartung von Softwaresystemen typischen punktuellen Arbeiten, unterbrochen von längeren Pausen, grundsätzlich schwer, Experte für alle Details eines Softwaresystems zu sein.

Entwicklerinnen und Entwickler müssen vielmehr – abhängig von der Aufgabenstellung – die jeweils zu ändernden Softwaresysteme und Programmteile identifizieren, sich gegebenenfalls ein neues Verständnis derselben und ihrer Zusammenarbeit erarbeiten sowie die durch sie möglicherweise ausgelösten Seiteneffekte lokalisieren. Experten werden typischerweise Hierarchien und Schemata verwenden, um sich das Softwaresystem so weit es geht, top-down zu erschließen. Lassen sich die Schemata und Hierarchien an einer bestimmten Stelle nicht weiter verwenden, so gehen sie zum Bottom-Up Programmverstehen über und setzen vermehrt Chunking ein.

Zusammenfassend lässt sich sagen, dass komplexe Strukturen hierarchisch aufgebaut sein und sinnvoll zusammenhängende Einheiten auf verschiedenen Abstraktionsniveaus enthalten sollten, die das Chunking erleichtern. Ferner sollten sich auf verschiedenen Abstraktionsniveaus wiederkehrende Muster finden lassen, aus denen Schemata gebildet werden können.

4.4. Umgang mit komplexen Architekturen

Als ein Aspekt von Architekturkomplexität habe ich im letzten Kapitel die Struktur der Ist-Architektur identifiziert (s. Definition 3-2). Aus dem Blickwinkel der kognitiven Psychologie ist die Ist-Architektur eine komplexe Struktur, mit der die Mitglieder eines Entwicklungsteams umgehen müssen. Daraus ergibt sich die Frage: Wie und in welchem Maße tragen Klassen- und Subsystem-Ebene sowie Architekturstile dazu bei, dass Chunking, die Bildung von Hierarchien und der Aufbau von Schemata unterstützt werden?

Chunking wird auf der Klassen- und Subsystem-Ebene unterstützt, indem oberhalb von Anweisungen zusammenfassende Einheiten als Programmiermittel angeboten werden. Schon die Möglichkeit, eine längere Berechnung als Operation zu implementieren und dadurch die einzelnen Anweisungen als höherwertige Wissenseinheiten abzuspeichern, führt zu einem deutlichen Kapazitätsgewinn im Kurzzeitgedächtnis. Klassen und Subsysteme führen diese Möglichkeit weiter, so dass entlang der Enthaltenseins-Hierarchie immer größere Wissenseinheiten gebildet werden können.

Von Schichtenarchitekturen und erweiterten Schichtenarchitekturen wird Chunking nicht stärker unterstützt als durch die Einführung von Subsystemen auf der Subsystem-Ebene. Schichten und Schnitte sind lediglich eine Umwidmung der obersten Ebene von Subsystemen, für die bestimmte Beziehungen verboten werden. Über den inneren Aufbau oder die Gestaltung der Schnittstelle wird bei (erweiterten) Schichtenarchitekturen keine Aussage gemacht.

Chunking kann, wie Abschnitt 4.1 gezeigt hat, nur wirksam sein, wenn die Architekturelemente sinnvoll zusammenhängende Einheiten darstellen. Beliebige Gruppierungen von Operationen zu Klassen, von Klassen zu Subsystemen und Subsystemen zu Schichten oder Schnitten erleichtern das Chunking nicht. Vielmehr müssen die Entwicklerinnen und Entwickler, obwohl sie Klassen normalerweise als eine Wissenseinheit betrachten, doch mehrere Wissenseinheiten aus den verschiedenen, zusammen gewürfelten Teilaspekten bilden.

Subsystemarchitekturen bieten in diesem Punkt Unterstützung für das Chunking, weil sie durch die starke Fokussierung auf Dienste und ihre Schnittstellen deutlich machen, dass Subsysteme und insbesondere ihre Schnittstellen aus sinnvoll zusammengehörenden Elementen bestehen sollten. Referenzarchitekturen fördern Chunking auf der Klassen-Ebene, indem sie Elementarten einführen, die eine bestimmte Zuständigkeit haben. Außerdem lassen sie Elementarten zu, die ihrerseits aus mehreren Klassen bestehen können, wie z. B. Werkzeuge, die mehrere Teilelemente haben.

Der *Aufbau von Hierarchien* wird durch die Vererbungsbeziehung und die Enthaltenseins-Beziehung direkt umgesetzt. Sie unterstützen Entwicklerinnen und Entwickler darin, Teil-Ganzes- oder Ober-Unterbegriffs-Hierarchien zu bilden. Im Gegensatz dazu kann die Benutzt-Beziehung allein oder in Kombination mit der Vererbungsbeziehung verwendet werden, um nicht

hierarchische Strukturen zu implementieren. Erbt beispielsweise Klasse A von Klasse B und Klasse B benutzt Klasse A, so entsteht eine nicht hierarchische Struktur. Die Betonung liegt hier auf dem Wort „kann“, denn es ist auch möglich, die Benutzt-Beziehungen hierarchisch einzusetzen.

Bei diesen Beziehungen setzen daher die meisten Architekturstile an. Der *Aufbau von Hierarchien* bei der Benutzt- und Vererbungsbeziehung wird von allen Architekturstilen bis auf die Subsystemarchitektur begünstigt. Der Fokus von Subsystemarchitekturen liegt nicht auf der Hierarchisierung von Beziehungen, sondern auf der Frage, ob bestimmte Beziehungen erlaubt sind oder nicht. Schichtenarchitekturen schreiben auf der Ebene von Schichten Hierarchien vor, und erweiterte Schichtenarchitekturen legen fest, dass zusätzlich die Schnitte hierarchisch geordnet sind. Der Aufbau von Beziehungs-Hierarchien wird also durch beide Architekturstile auf der Subsystem-Ebene unterstützt. Referenzarchitekturen gehen von der Klassen-Ebene aus und schreiben vor, dass die Beziehungen zwischen Klassen hierarchisch sein sollen. Für Packages und Subsysteme machen Referenzarchitekturen keine Vorgaben, so dass hier automatisch keine Hierarchie erzwungen wird.

Der *Aufbau von Schemata* wird besonders von Referenzarchitekturen unterstützt. Dadurch, dass in Referenzarchitekturen ein reicher Satz an Elementarten festgelegt wird, können Schemata für diese einzelnen Elementarten aufgebaut werden. Dadurch, dass Referenzarchitekturen Elementarten definieren, entsteht wie bei Entwurfsmustern eine erweiterte Sprachebene, die auf den neuen Schemata aufbaut. Schichten- und Subsystemarchitekturen bieten keine so feingranularen Schemata wie Referenzarchitekturen an. Allerdings kann die Einteilung eines Softwaresystems in Schichten oder Subsysteme als Schema verstanden werden, das für Entwicklerinnen und Entwickler einen Anhaltspunkt für ihr Verständnis bietet.

4.5. Zusammenfassung

Die in diesem Kapitel zusammengefassten Erkenntnisse unterstützen die Vermutung, dass objektorientierte Programmiersprachen und Architekturstile zur Verringerung von Architekturkomplexität beitragen. Sie unterstützen Chunking und fördern die Bildung von Hierarchien und den Aufbau von Schemata.

Allerdings werden einige Empfehlungen nur zum Teil umgesetzt. Chunking kann von Menschen nur dann eingesetzt werden, wenn ihnen sinnvoll zusammenhängende Einheiten angeboten werden. Die Bildung von Hierarchien wird für Benutzt- und Vererbungsbeziehungen erst auf Ebene der Architekturstile angegangen. Schemata werden schließlich am stärksten von Referenzarchitekturen unterstützt.

Für Architekturkomplexität nach Definition 3-2 ist daher zu erwarten, dass die *Struktur der Ist-Architektur* durch den Einsatz von Architekturstilen weniger Komplexität aufweist, die vier Architekturstile aber unterschiedliche Auswirkungen haben. Für den zweiten Aspekt von Architekturkomplexität, die

Abbildung auf den Programmtext, sind besonders der Aufbau von Schemata und ihre Abbildung wichtig. Werden von der Soll-Architektur und dem Architekturstil eine Reihe von Schemata vorgeschlagen, finden sich dann aber nicht im Programmtext wieder, so wird die Komplexität eher verstärkt als gemindert.

Diese Überlegungen bilden die Grundlage für die von mir durchgeführte Untersuchung mit insgesamt 24 Fallstudien. Dabei habe ich mir die Fragen gestellt, welche Architekturstile in Softwaresystemen verwendet werden und ob sie sichtbare Effekte auf die Architekturkomplexität haben.

5. Untersuchung zur Architekturkomplexität

Die Untersuchung zur Architekturkomplexität habe ich von April 2005 bis Februar 2007 durchgeführt. Insgesamt bin ich mit 28 Entwicklungsteams in Kontakt gewesen und habe für diese Arbeit 24 der dabei entstandenen Fallstudien ausgewertet. Wie ich vorgegangen bin und welche Fälle ich ausgewählt habe, beschreibe ich in diesem Kapitel.

Zuerst stelle ich die verwendete Methodik vor und erläutere, nach welchen Kriterien die Fallstudien ausgewählt wurden. Die darauf folgenden Abschnitte zeigen, wie die Fallstudien durchgeführt und die Ergebnisse für die Auswertung vorbereitet wurden. Den Abschluss des Kapitels bildet eine Zuordnung der Fallstudien zu den jeweils verwendeten Architekturstilen.

5.1. Forschungsmethodik

Als ich im April 2005 mit der ersten Untersuchung begonnen habe, hatte ich kein fertiges Modell für Architekturkomplexität, und der Fragenkatalog für die Interviews mit den Architektinnen und Architekten war noch relativ klein und inhaltlich weit gefächert. Mit jeder weiteren Fallstudie haben sich meine Einsichten und Erkenntnisse entwickelt und verändert, so dass ich schließlich diese Arbeit schreiben konnte. Die Methode, an die ich mein Vorgehen angelehnt habe, wird als *Grounded Theory* bezeichnet. Das von mir eingesetzte Verfahren für die Auswahl der Fallstudien ist das *Theoretische Sampling*. Im Folgenden stelle ich Methode und Auswahlverfahren vor und beschreibe meinen Umgang damit.

5.1.1. Grounded Theory

Die Methode *Grounded Theory* hat zum Ziel, in einem sich entfaltenden Forschungsprozess eine Theorie zu entwickeln, die den untersuchten Zusammenhang erklärt. Im Gegensatz zu anderen Methoden wird bei Grounded Theory nicht von einer Theorie ausgegangen, die bestätigt werden soll, sondern die Fragestellung entwickelt sich im Laufe der Forschung (vgl. Cockburn 2003; Glaser und Strauss 1967; Strauss und Corbin 1998).

Man beginnt mit einer Untersuchung in einem Forschungsbereich, sammelt Daten, wertet sie aus und erlaubt dabei den relevanten Fragestellungen „aufzutauchen“ (engl. to emerge) (Cockburn 2003, S. 31). Bei diesem Vorgehen ist das Sammeln und Auswerten von Daten kein sequentieller Prozess, sondern die Aktivitäten überlappen sich. Sind bei der Durchsicht der gesammelten Daten erste Ideen entstanden, so werden weitere Untersuchungen durchgeführt. Dabei kann sich die Fragestellung verschieben oder weiter ausdifferenzieren. Aus den gesammelten Rohdaten werden inhaltliche Ähnlichkeiten zu Einheiten bzw. Begriffen (sog. categories) zusammengefasst und ihre Beziehungen herausgearbeitet. Auch nach diesem Schritt können weitere Untersuchungen durchgeführt werden, um zusätzliche Informationen zu gewinnen, die sich z. B. auf Variationen der Begriffe beziehen oder das Begriffsgebäude festigen sollen (Bortz und Döring 2006, S. 332).

Um den Begriff *Architekturkomplexität* mit Leben zu füllen, habe ich mich an der Grounded Theory orientiert. Der Forschungsbereich war leicht zu identifizieren: Softwarearchitektur und der Umgang von Entwicklungsteams mit Komplexität. Im meinem Forschungsprozess habe ich Softwaresysteme analysiert und mit ihren Architektinnen und Architekten über die von ihnen wahrgenommene Komplexität diskutiert. Neben den Architekturanalysen habe ich mich mit dem Thema Komplexität und ihrer Bewältigung in der Informatik, der kognitiven Psychologie und der Soziologie beschäftigt. Die sich dabei und aus den Analysen entwickelnden Erkenntnisse habe ich zusammengetragen und in Beziehung gesetzt.

Das im nächsten Kapitel vorgestellte Modell für Architekturkomplexität habe ich im Laufe dieses Prozesses verfeinert, präzisiert und immer wieder überprüft. Parallel dazu entstand die Idee, das Vorgehen der Entwicklungsteams als Strategien zu formulieren. Das Ausarbeiten der Strategien und des Modells für Architekturkomplexität haben mir immer wieder Stoff für neue Fragen und Überlegungen geliefert, die ich in den nächsten Architekturanalysen genauer untersuchen konnte. Schließlich versiegte der Strom neuer Erkenntnisse langsam, und das für das Auswahlverfahren *Theoretisches Sampling* typische Phänomen der *Theoretischen Sättigung* trat ein.

5.1.2. Theoretisches Sampling

Die Auswahl der Fallstudien habe ich schrittweise durch ein Verfahren vorgenommen, welches in der qualitativen Sozialforschung als Theoretisches Sampling bezeichnet wird (Bortz und Döring 2006, S. 333; Flick 2006, S. 108; Lamnek 1995, S. 195). Dabei werden Entscheidungen über die Auswahl und Zusammensetzung des empirischen Materials (Fälle, Untersuchungsgruppen) im Prozess der Datenerhebung und Auswertung getroffen (Flick 2006, S. 102). Theoretisches Sampling orientiert sich bei der Fallauswahl an dem für die zu entwickelnde Theorie zu erwartenden Neuigkeitsgehalt. Daraus ergeben sich Konsequenzen für die Zusammensetzung des Sample:

> „Bei qualitativer Forschung geht es nicht um eine große Zahl von Fällen, sondern um für die Fragestellung typische Fälle. Daraus leitet sich ab, dass Repräsentativität kein entscheidendes Auswahlkriterium ist. Angemessenheit für die theoretische Fragestellung ist entscheidend. Deshalb werden auch keine statistisch-wahrscheinlichkeitstheoretisch bestimmten Stichproben gezogen, sondern die Fälle werden nach theoretischen Vorstellungen in die Analyse einbezogen. […] Die Fälle können willkürlich unter dem Aspekt ausgewählt werden, eine Theorie zu entdecken und / oder zu erweitern (grounded theory) oder gezielt, um die Theorie anhand mutmaßlich abweichender Fälle zu kontrollieren und / oder zu revidieren (analytische Induktion). Die Stichprobengröße ist vorher nicht festgelegt. Die Einbeziehung weiterer Fälle kann beendet werden, wenn eine ‚theoretische Sättigung' erreicht ist. Dies hat den Vorteil, dass die Auswahl – anders als in echten Zufallsstichproben – während der Untersuchung den theoretischen Bedürfnissen folgend erweitert werden kann." (Lamnek 1995, S. 195)

Zu Beginn meiner Untersuchung war nicht klar, wie viele Fallstudien zu meinem Sample gehören würden. Fest stand jedoch, dass für die Fallstudien nur Entwicklungsteams in Frage kommen, die ein Softwaresystem kontinuierlich weiterentwickeln, das für den Einsatz in der Praxis bestimmt ist. In einer solchen Situation entsteht für das Entwicklungsteam die Notwendigkeit, sich mit Architekturkomplexität auseinander zu setzen und Strategien zu entwickeln, um sie zu reduzieren. Werden lediglich Prototypen erstellt, treten die in dieser Arbeit untersuchten Komplexitätsprobleme nur selten auf. Aus diesem Grund habe ich mich an Unternehmen gewandt, die objektorientierte Softwaresysteme entwickeln, und im universitären Kontext nach vergleichbaren Projekten gesucht.

Da sich die Architektur eines Softwaresystems selten aus dem Programmtext alleine erschließen lässt, habe ich mich um Projekte bemüht, bei denen ich Zugang zu einer oder mehreren Personen bekommen konnte, die die Soll-Architektur, den Architekturstil und die Abbildungsvorschrift des jeweiligen Softwaresystems kennen. Eine zufällige Auswahl von Fällen, wie sie von Collberg et al. über Suchmaschinen im Internet vorgenommen wurde (Collberg et al. 2004, S. 2) oder eine Einschränkung auf OpenSource-Systeme, wie sie von Baxter et al. gewählt wurde (Baxter et al. 2006, S. 398), war für diese Arbeit nicht sinnvoll, weil ich Kontakt zu den Architektinnen und Architekten brauchte.

5.2. Auswahl relevanter Fälle

Am Anfang habe ich alle Softwaresysteme untersucht, zu denen ich Zugang bekam. Schnell wurde klar, dass ich mich auf Softwaresysteme mit Architekturstil oder zumindest mit einer Soll-Architektur einschränken wollte, um in allen Fallstudien die gleichen Aspekte von Architekturkomplexität analysieren zu können. Vier Softwaresysteme wurden daher aus der Untersuchung ausgeschlossen. Für diese Softwaresysteme existierte weder auf Papier noch bei den Entwicklerinnen und Entwicklern eine Strukturierung ihres Softwaresystems in Subsysteme und/oder Schichten, geschweige denn Elementarten für Klassen im Sinne einer Referenzarchitektur.

Als die Anzahl der Fallstudien auf ca. fünfzehn angewachsen war, begann ich nach Softwaresystemen mit Eigenschaften zu suchen, die in meinem Sample noch fehlten. Einerseits war mir wichtig, dass sich die Softwaresysteme in ihrer Größe und dem bei ihnen eingesetzten Architekturstil unterschieden. Andererseits habe ich ähnliche Softwaresysteme untersucht, um zu sehen, ob sich bestimmte Konstellationen identifizieren lassen, die Einfluss auf Architekturkomplexität haben. Als deutlich wurde, dass weitere Fallstudien keine neuen Erkenntnisse bringen würden, habe ich das Sample geschlossen.

Die Größe der Softwaresysteme im Sample liegt zwischen 10.000 bis 7 Millionen relevanter Programmzeilen (RLOC – Relevant Lines of Code) (s. Abbildung 5-1). Unter relevanten Programmzeilen werden alle Zeilen verstanden, die mit einem Zeilenumbruch enden, abzüglich Leerzeilen, Kommentarzeilen und Zeilen mit eingerückten geschweiften Klammern. Die

exakte Anzahl der relevanten Programmzeilen und die Anzahl aller Programmzeilen incl. Kommentarzeilen etc. lassen sich aus Tabelle A-1 im Anhang entnehmen.

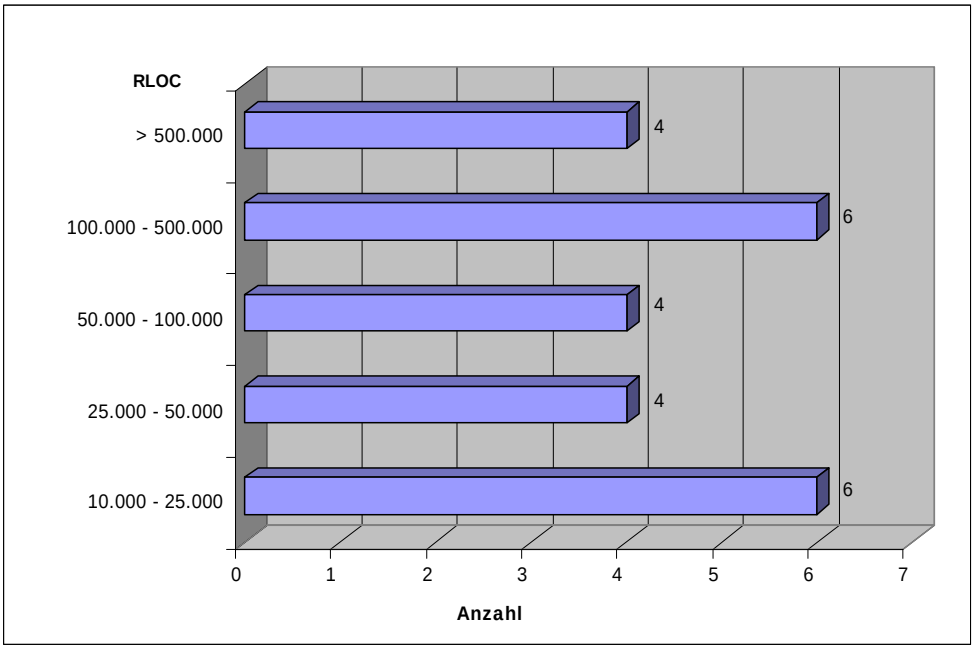

Abbildung 5-1: Größenverteilung der Fallstudien

Einundzwanzig der vierundzwanzig Softwaresysteme werden in der Wirtschaft entwickelt und in Unternehmen eingesetzt. Zwei Softwaresysteme stammen aus dem universitären Kontext und werden in der Lehre und in Projekten mit Studierenden verwendet. Ein weiteres Softwaresystem wird von der Schulbehörde für die Verwendung in Schulen entwickelt.

Das Alter der Softwaresysteme lag zum Zeitpunkt der Untersuchung zwischen einem Jahr und sieben Jahren (s. Abbildung 5-2). Die Altersangaben zu jeder Fallstudie lassen sich aus Tabelle A-1 im Anhang entnehmen.

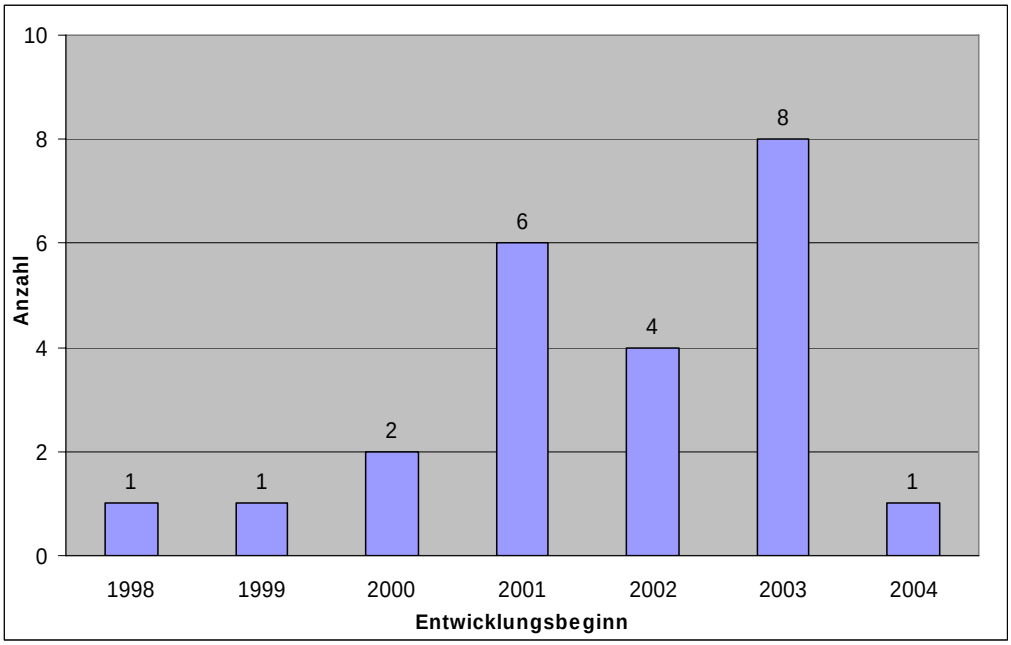

Abbildung 5-2: Beginn der Softwareentwicklung

Für die Unternehmen, die sich zu einer Untersuchung bereit erklärt haben, war Geheimhaltung absolute Voraussetzung, denn Softwaresysteme werden von vielen Unternehmen als einer ihrer wichtigsten Vermögenswerte betrachten. Die Fallstudien wurden daher anonymisiert und sind der Einfachheit halber im weiteren Verlauf der Größe (RLOC) nach durchnummeriert (s. Tabelle A-1). Insgesamt konnte ein breites Spektrum an Einsatzkontexten abgedeckt werden: Bank, Versicherung, Abrechnung und Buchung für Ticket-Verkauf und Vermietung, Koordination und Planung von logistischen Problemstellungen, ContentManagement (CMS), Behördenverwaltung, Arztpraxis, CallCenter, Kommunikationsplattform, Simulation und Software-Entwicklungsumgebung. Betriebssysteme, Systemsoftware oder Software, die ohne Benutzungsschnittstelle nur auf einem Server läuft, wurden nicht untersucht.

Bei vier Fallstudien (5, 11, 14 und 15) haben die mir zugänglichen Mitglieder des Entwicklungsteams auf die Frage nach dem Zustand ihres Softwaresystems gesagt, dass sie große Schwierigkeiten bei der Wartung und Weiterentwicklung hätten und davon ausgingen, dass ihr Softwaresystem viel Architekturkomplexität besitze. Diese Einschätzung hat sich bei der Untersuchung der vier Softwaresysteme bestätigt und wird bei den Ergebnissen in Kapitel 8 sowie den Auswertungen zur architekturzentrierten Softwareentwicklung in Kapitel 9 mehrfach sichtbar werden. Die Entwicklungsteams der anderen 20 Fallstudien waren mit dem Zustand ihres Softwaresystems zufrieden und hielten es grundsätzlich für wartbar und erweiterbar.

5.3. Vorgehen bei den Fallstudien

Hatte ein Unternehmen eingewilligt, dass eins seiner Softwaresysteme untersucht werden durfte, so habe ich als erstes einen Termin für eine Architekturanalyse vereinbart. Wie genau die Architekturanalysen abgelaufen sind, beschreibe ich im folgenden Abschnitt. Im Anschluss an die Architekturanalyse habe ich bei etwa der Hälfte der Fallstudien Interviews mit den jeweiligen Architektinnen oder Architekten durchgeführt, um Fragen aus der Architekturanalyse zu klären. War die Architekturanalyse aussagekräftig genug, so habe ich auf das Interview verzichtet.

Bei zwei Fallstudien (19 und 24) war es aus organisatorischen oder firmeninternen Gründen nicht möglich, die Softwaresysteme selbst zu untersuchen. Das Softwaresystem aus Fallstudie 24 ließ sich aufgrund seiner Größe von 7 Millionen RLOC nicht in angemessener Zeit analysieren, so dass ich lediglich den Chefarchitekten interviewt habe. Fallstudie 19 enthält ein Interview, das ich mit seiner Architektin auf einer Konferenz geführt habe. Obwohl in beiden Fällen keine Analyse und damit keine direkten Untersuchungen der Architektur möglich waren, habe ich diese beiden Interviews in das Sample mit aufgenommen, weil sie wichtige Beiträge für die Auswertung zur Architekturkomplexität und der Beschreibung der Strategien liefern (s. Kapitel 8 und 10).

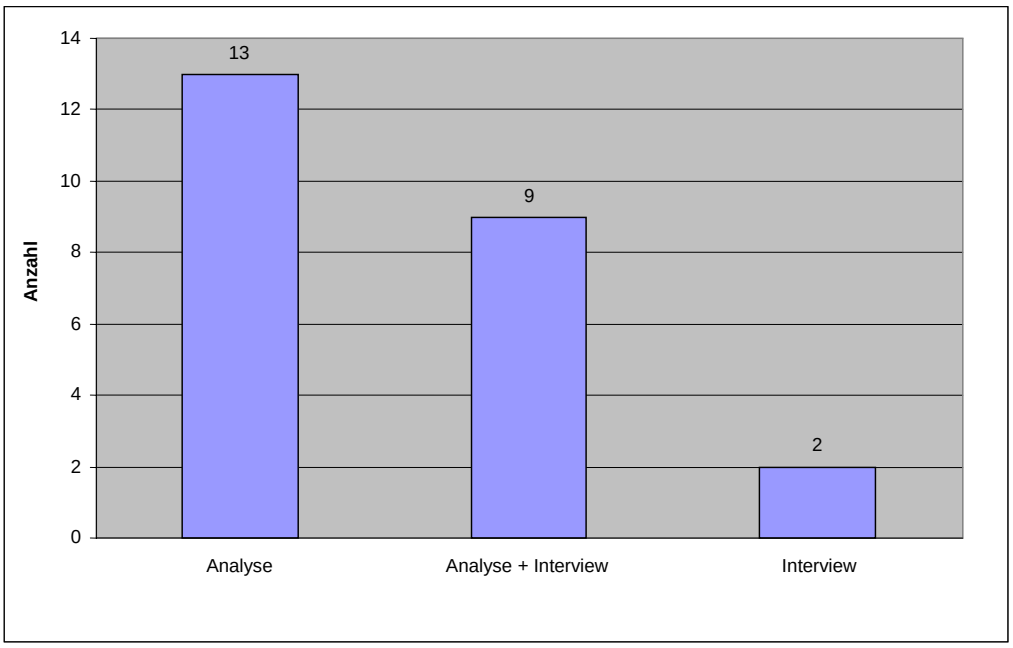

Abbildung 5-3: Auswertungen der Fallstudien

Abbildung 5-3 gibt einen Überblick über die Anzahl der Analysen und Interviews. Aus Tabelle A-1 im Anhang lässt sich entnehmen, bei welchen Fallstudien eine Architekturanalyse und bei welchen ein Interview durchgeführt wurde.

5.3.1. Architekturanalyse

Die Architekturanalyse dauerte je nach Größe des Softwaresystems zwischen einem und vier Tagen. Bei der Architekturanalyse habe ich als Analysewerkzeug den Sotographen[9] eingesetzt (s. Abschnitt 5.4.1). Während der Analyse konnte ich die Architektinnen und Architekten des jeweiligen Softwaresystems hinzuziehen, um die Architektur mit ihnen gemeinsam zu untersuchen und zu diskutieren. Besonders für die Einteilung des Softwaresystems in Subsysteme und Schichten war ich auf die Aussagen der Architektinnen und Architekten angewiesen, weil diese Ebene dem Programmtext nicht zu entnehmen ist (s. Kapitel 3.2.2).

Die zeitliche Länge der Architekturanalyse ließ es nicht sinnvoll erscheinen, eine Aufnahme auf Tonband vorzunehmen. Stattdessen habe ich wichtige Details protokolliert und im Anschluss an die Analyse einen Kurzbericht mit der folgenden Struktur erstellt:

- Überblick über das Projekt
- Ablauf der Analyse
- Entwicklungssituation
- Ergebnisse der Vermessung
- Bewertung auf der Subsystem-Ebene
- Bewertung auf der Klassen-Ebene
- Erkenntnisse aus der Untersuchung

Vier Analyseberichte sind im Anhang B zu finden.

5.3.2. Interview

Bei den Interviews habe ich die Architektinnen und Architekten darum gebeten, den Zustand ihres Softwaresystems einzuschätzen. Außerdem habe ich mit ihnen über die in ihrem Softwaresystem vorhandene Architekturkomplexität diskutiert und die im Team eingesetzten Strategien zur Komplexitätsreduktion erfragt.

Die Interviews wurden als Experten-Interviews durchgeführt. Ein Experten-Interview kann als eine spezielle Form des Leitfaden-Interviews verstanden werden, bei der die Befragten in ihrer Rolle als Expertinnen oder Experten über ein bestimmtes Handlungsfeld Auskunft geben sollen (Bortz und Döring 2006, S. 237; Flick 2006, S. 139). In diesem Sinne repräsentiert der Befragte eine Gruppe von bestimmten Experten, wodurch solche Interviews thematisch stärker fokussiert und begrenzt sind als z.B. biographische Interviews, bei denen der Interviewte die eigene Lebensgeschichte erzählt.

Im Experten-Interview habe ich einen Fragenkatalog verwendet. Der Fragenkatalog diente primär zu meiner Orientierung im Hinblick darauf, welche

[9] www.software-tomography.com

Themen sich situativ in den bisherigen Gesprächsverlauf einflechten ließen. Zum Teil habe ich aber auch Fragen aus den Architekturdiskussionen erneut aufgegriffen, um die in dieser Arbeit untersuchte Architekturkomplexität genauer zu hinterfragen. Der Fragenkatalog wurde in folgende Themenbereiche untergliedert:

- Einleitung
- Vorerfahrung des Interviewten
- Projektspezifischer Einstieg
- Architekturvorstellung im Entwicklungsteam
- Zustand des Softwaresystems
- Komplexität des Softwaresystems
- Strategien im Entwicklungsteam
- Abschluss

Die Interviews dauerten im Durchschnitt eineinhalb Stunden und wurden auf Tonband aufgenommen und später transkribiert.

5.4. Ergebnisanalyse

Während der Architekturanalyse habe ich die Softwaresysteme mit der Software-Analyseumgebung Sotograph der Firma Software-Tomography GmbH[10] untersucht. Im Folgenden stelle ich den Sotographen vor und lege dar, wie ich die Daten aus den Architekturanalysen für die Auswertung aufbereitet habe.

5.4.1. Analysewerkzeug

Der Sotograph ist eine Plattform, die eine schnelle und komfortable Untersuchung der inneren Struktur von Softwaresystemen in C/C++, Java oder C# ermöglicht. Abbildung 5-4 veranschaulicht den Aufbau der Sotograph-Plattform, die ich in der Version 2.6 verwendet habe. Der Quelltext des zu analysierenden Softwaresystems wird in eine relationale Datenbank (Repository) eingelesen, welche als mySQL[11]-Datenbank mitgeliefert wird. Dort werden die Architekturelemente, ihre Beziehungen und weitere Strukturmerkmale des Softwaresystems abgelegt und können anschließend mit den verschiedenen Werkzeugen des Sotographen ausgewertet werden.

[10] www.software.tomography.com
[11] www.mysql.com

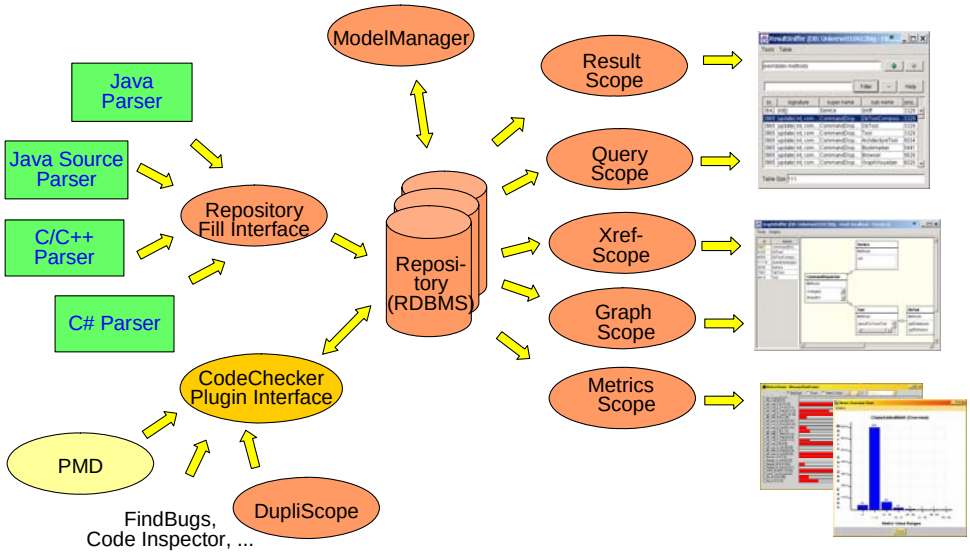

Abbildung 5-4: Aufbau des Sotographen

Nachdem ein Softwaresystem eingelesen worden ist, werden für verschiedene Qualitätseigenschaften Messwerte auf der Klassen- und Package-Ebene berechnet. Dazu setzt der Sotograph einerseits den OpenSource-CodeChecker PMD als Plugin ein und verwendet andererseits eine Reihe in SQL implementierte Metriken sowie ein eigenes Auswertungsprogramm (DupliScope), um duplizierten Programmtext zu identifizieren.

Um die Abbildung der Soll-Architektur auf den Programmtext vorzunehmen (s. Abschnitt 3.4), steht im Sotographen ein eigenes Werkzeug (ModelManager) zur Verfügung. Dort wird die Abbildungsvorschrift in Java programmiert, indem Packages Subsystemen zugeordnet und Schnittstellen für die Subsysteme definiert werden. Diese Information wird im Subsystemmodell festgehalten. Der Sotograph bietet in der Version 2.6 nur eine Ebene von Subsystemen an, so dass Subsysteme nicht, wie in Kapitel 3.2.2 gefordert, ineinander verschachtelt werden können. Um an dieser Stelle trotzdem Flexibilität zu ermöglichen, können im Sotographen beliebig viele Subsystemmodelle erstellt werden, so dass verschiedene Aspekte der Strukturierung durch mehrere Subsystemmodelle ausgedrückt werden können. Oberhalb der Subsysteme besteht die Möglichkeit, weitere Strukturierungen mit Hilfe von beliebig vielen Schichten- und Subsystemarchitekturen vorzunehmen. Für eine Schichtenarchitektur werden Subsysteme in Schichten zusammengefasst. Bei einer Subsystemarchitektur werden Regeln für die Benutzung zwischen den Subsystemen festgelegt.

Für die Untersuchung von Klassen- und Subsystem-Ebene bietet der Sotograph verschiedene Werkzeuge an: den MetricScope mit ca. 300 Softwaremaßen, den Query-Scope mit ca. 100 vorgefertigten Abfragen, den XRefScope für beliebige relationale Abfragen, den GraphScope zur Visualisierung von Zusammenhängen auf allen Ebenen des Softwaresystems und den QueryDeveloper für die Entwicklung von eigenen Abfragen. Weitere Möglichkeiten

des Sotographen, wie Trendanalyse und Reporting, die in dieser Arbeit nicht eingesetzt wurden, lasse ich deshalb weg.

Bei dem Versuch, Unternehmen davon zu überzeugen, dass sie mir die Analyse eines ihrer Softwaresysteme gestatten sollten, war der Sotograph eine wertvolle Unterstützung. Einige Entwicklungsteams kannten den Sotographen und hatten großes Interesse, die Ist-Architektur des eigenen Softwaresystems mit Hilfe des Sotographen „sehen" zu können. War der Sotograph noch nicht bekannt, so konnte ich durch eine Präsentation in den meisten Fällen das Interesse wecken.

Neben dem Sotographen stehen im OpenSource- und kommerziellen Bereich weitere Werkzeuge, wie JDepend[12], XRadar[13], Lattix[14] und SonarJ[15] zur Verfügung, um Architektur zu analysieren. Keines dieser Werkzeuge, auch nicht der Sotograph, wertet Architekturen auf die Weise aus, wie ich es für das Modell der Architekturkomplexität gebraucht habe. Der Sotograph hat im Vergleich zu den anderen Werkzeugen den entscheidenden Vorteil, dass er die Strukturdaten der analysierten Softwaresysteme in eine Datenbank ablegt. Das Datenbankmodell ist offen gelegt, und der Sotograph bietet eine auf SQL-basierende Abfragesprache an, mit der spezielle Auswertungen programmiert werden können. Diese Möglichkeit habe ich genutzt, um die Auswertungen zur Architekturkomplexität vorzunehmen.

Eine detaillierte Auseinandersetzung mit weiteren Vor- und Nachteilen der heute verfügbaren Analysewerkzeuge kann in verschiedenen an der Universität Hamburg durchgeführten Bachelor- und Diplomarbeiten nachgelesen werden (Scharping 2006, S. 14f; Schreier 2006, S. 47ff).

5.4.2. Datenaufbereitung

Da die mit dem Sotographen durchgeführte Analyse darauf ausgerichtet war zu überprüfen, welche und wie viele komplexe Strukturen in Softwaresystemen enthalten sind, wurden nur die Teile der Softwaresysteme betrachtet, die die Entwicklerinnen und Entwickler tatsächlich bearbeiten. Aus diesem Grund habe ich Bibliotheken, von denen nur die Schnittstelle (API) verwendet wird, wie JDK, Hibernate, Struts etc., und Systemteile, die generiert werden, von der Analyse ausgenommen.

Darüber hinaus habe ich Architekturverletzungen – also Abweichungen zwischen Soll- und Ist-Architektur bzw. dem Architekturstil –, die leicht behebbar waren, während der Analyse korrigiert. Solche leicht behebbaren Befunde, sog. falsch Positive, waren bei der Analyse mit den Architektinnen und Architekten nicht zu übersehen. In der Regel kündigte sich ein solcher Fund mit den Worten an: „Was ist denn das da? Das darf aber ganz bestimmt nicht sein! Können wir uns das mal ansehen, bitte? Ach so, na ja, da hat mal wieder jemand … Das lässt sich leicht reparieren!".

[12] clarkware.com/software/JDepend.html
[13] xradar.sourceforge.net
[14] www.lattix.com
[15] www.hello2morrow.de

In praktisch allen Fallstudien traten die folgenden Arten von falsch Positiven auf:

- Flüchtigkeitsfehler durch Kopieren: Kopierfehler wurden durch die korrekten Aufrufe ersetzt.
- Aufrufe von Konstanten: Die Konstanten wurden in eine andere Klasse verschoben.
- Falsche Zuordnung: Eine oder mehrere Klassen bzw. Packages wurden im Package-Baum verschoben.
- Falsche Schnittstelle: Offensichtlich fehlende Klassen wurden nachträglich zur Schnittstelle hinzugefügt.

Nachdem die falsch positiven Befunde im Sotographen behoben worden waren oder in einem Analyseprotokoll vermerkt waren, habe ich die für die Untersuchung benötigten Messwerte mit dem Sotographen erhoben. Die Ergebnisse der einzelnen Vermessungen wurden aus dem Sotographen nach Excel exportiert und dort weiterverarbeitet. Um die Ergebnisse der einzelnen Fallstudien vergleichbar zu machen, habe ich die jeweils gemessenen Werte normiert, indem ich z. B. die Anzahl der Klassen in Zyklen durch die Gesamtanzahl der Klassen geteilt habe.

5.5. Fallstudien und Architekturstile

Die Zuordnung der Fallstudien zu den in Abschnitt 3.3 eingeführten Architekturstilen habe ich während der Architekturanalyse mit den Architektinnen und Architekten vorgenommen. Ausgangspunkt der Unterscheidung war, wie sie die Architektur ihres Softwaresystems beschrieben, wenn ich ihnen die Frage gestellt habe: „Welche Architektur hat Ihr Softwaresystem?“. Die üblichen Antworten waren: „Das System hat Schichten.“, „Unser System besteht aus sieben Komponenten.“, „Wir haben unser System in Subsysteme aufgeteilt.“ und „Das System hat eine Quasar bzw. WAM-Architektur.“.

Die Verteilung der verschiedenen Architekturstile lässt sich aus Abbildung 5-5 entnehmen. Alle Fallstudien, bei denen mehr als ein Architekturstil eingesetzt wurde, habe ich als kombinierte Stile gezählt. Am Ende dieses Abschnitts und im Anhang sind alle Fallstudien mit ihren Architekturstilen aufgelistet (s. Tabelle 5-1 und Tabelle A-1).

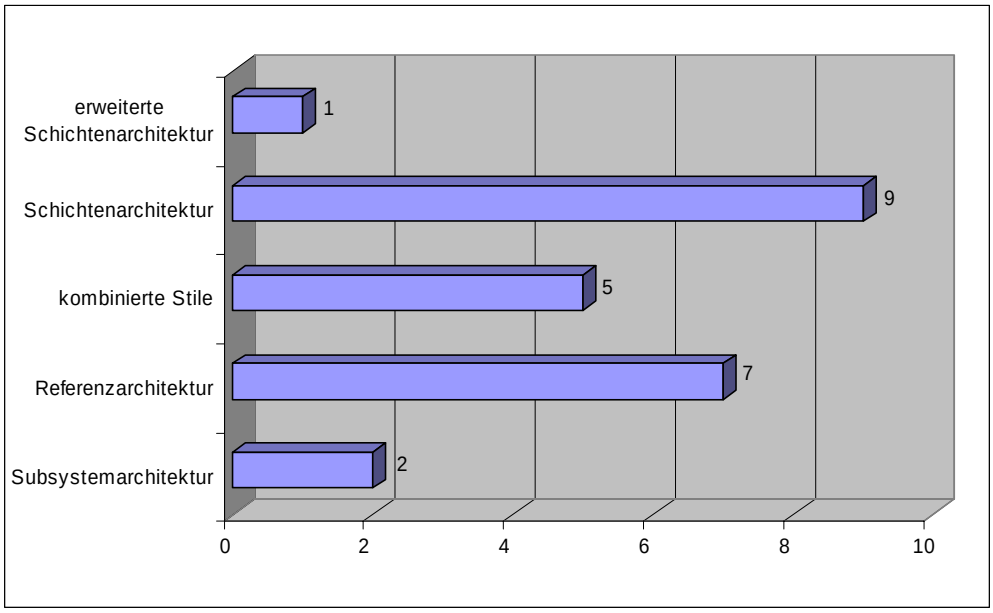

Abbildung 5-5: Verteilung auf die Architekturstile

Schichtenarchitektur (s. Abschnitt 3.3.1) und Subsystemarchitektur (s. Abschnitt 3.3.2) waren leicht zu unterscheiden, weil die Architektinnen und Architekten bei Schichtenarchitekturen immer von mehreren Schichten sprachen und es ihnen wichtig war, dass die tieferen Schichten die höhere nicht benutzen dürfen. Als Schichtenarchitekturen habe ich daher alle Softwaresysteme gezählt, bei denen im Interview explizit von Schichten gesprochen wurde und kaum Elementarten erwähnt wurden, wie sie bei Referenzarchitekturen zu finden sein müssten (Fallstudie 6, 11, 13, 14, 15, 16, 18, 19, 23).

Bei Subsystemarchitekturen betonten die Architektinnen und Architekten, dass Schichten in ihrem Softwaresystem keine Rolle spielen würden. Dementsprechend haben wir in der Architekturanalyse keine Schichten, sondern Subsysteme modelliert. Eine Architektin sagte im Interview: „Ja, vielleicht ist dies (keine Schichten) damit begründet, dass wir mit diesem Komponentendenken angefangen haben. Drei Leute, drei Aufgaben, drei wirklich abgeschlossene Komponenten.“ Schnittstellen, die bei Schichtenarchitekturen oft wenig Aufmerksamkeit bekamen, waren bei diesen Architekturen sehr wichtig, so dass die Definition der Schnittstellen und eine Überprüfung, ob sie eingehalten werden, für diese Architektinnen und Architekten am spannendsten waren (Fallstudie 8, 20 21).

Zwei Softwaresysteme (Fallstudie 8 und 20) wurden von ihrer Architektin bzw. ihrem Architekt als reine Subsystemarchitektur beschrieben. Für ein Softwaresystem (Fallstudie 21) stellte sich bei der Analyse heraus, dass es einen kombinierten Architekturstil aus Subsystemarchitektur und Referenzarchitektur besitzt.

Bei einigen Softwaresystemen sprachen die Architektinnen und Architekten nicht nur von Schichten, sondern auch davon, dass die Schichten in einzelne

Teile unterteilt sein sollten, die für jeweils einen Anwendungsbereich im Unternehmen stehen. Diese Softwaresysteme wurden dem Architekturstil *Erweiterte Schichtenarchitektur* zugeordnet (s. Abschnitt 3.3.3). Als erweiterte Schichtenarchitekturen konnten insgesamt drei Softwaresysteme (Fallstudie 4, 22, 24) identifiziert werden. In Abbildung 5-5 ist nur eine Fallstudie als erweiterte Schichtenarchitektur aufgeführt, weil bei den zwei anderen Fallstudien (4 und 22) die erweiterte Schichtenarchitektur mit einer Referenzarchitektur kombiniert ist.

Eine Aufteilung eines Softwaresystems in mehrere Schnitte wird in der Regel erst bei größeren Softwaresystemen wie Fallstudie 22 und 24 notwendig. Fallstudie 4 (ca. 22.000 RLOC) stellte in dieser Untersuchung eine Besonderheit dar, weil dieses relativ kleine Softwaresystem einen hoch kooperativen Arbeitszusammenhang abbildet und insgesamt für vier an der Kooperation beteiligte Rollen zugeschnittene Funktionalitäten zur Verfügung stellt. Diese vier Arbeitsplätze wurden von den Architekten als getrennte Systemteile konzipiert und implementiert, so dass eine erweiterte Schichtenarchitektur entstanden ist.

Softwaresysteme mit einer Referenzarchitektur (s. Abschnitt 3.3.4) habe ich explizit gesucht, um neben Architekturstilen, die auf der Subsystem-Ebene angesiedelt sind, auch Architekturstile untersuchen zu können, die von der Klassen-Ebene ausgehen. Insofern war bei einer Reihe von Softwaresystemen im Vorfeld klar, dass sie eine Referenzarchitektur haben. Es konnten sieben Softwaresysteme mit WAM-Architektur (Fallstudie 1, 2, 4, 5, 7, 9, 10) und ein Softwaresystem mit einer Quasar-Architektur analysiert werden (Fallstudie 21). Bei diesen Softwaresystemen sprachen die Architektinnen und Architekten namentlich von der gewählten Referenzarchitektur, so dass eine direkte Zuordnung des Softwaresystems zum jeweiligen Architekturstil möglich war.

Bei vier Softwaresystemen wurde erst während der Diskussion mit den Architektinnen und Architekten klar, dass eine projekteigene Referenzarchitektur eingesetzt wird (Fallstudie 3, 12, 17, 22). Die Architekten beriefen sich nicht auf eine außerhalb ihres Unternehmens bekannte Referenzarchitektur; sie besaßen aber ein ausgefeiltes Repertoire an Elementarten für Klassen und Regeln, die für die Benutzung zwischen Klassen gelten. Zusätzlich stehen in diesen Unternehmen Schulungsunterlagen zur Verfügung, mit denen neuen Mitarbeiterinnen und Mitarbeitern die verschiedenen Arten von Klassen, ihre Schnittstellen, Aufgaben und Beziehungen vermittelt werden. Diese vier Fallstudien habe ich zu den Referenzarchitekturen gezählt.

Auch in einigen der anderen Fallstudien mit Schichten- oder Subsystemarchitekturen konnte ich Elementarten für Klassen beobachten. Verschiedene Architektinnen und Architekten sprachen von „Business-Objects“ oder „Value-Objects“. Für diese Arbeit habe ich anhand der folgenden Fragen entschieden, ob ein Softwaresystem eine Referenzarchitektur bzw. eine projekteigene Referenzarchitektur hat oder nicht:

- Sprechen die Architektinnen und Architekten des Softwaresystems in Elementarten von Klassen?
- Gibt es neben Klassenarten insbesondere Regeln, die die Beziehungen zwischen den verschiedenen Klassen einschränken und somit zu einer hierarchischen Struktur führen?
- Sind die Klassenarten und Regeln für das gesamte Entwicklungsteam verbindlich oder werden sie nur in manchen Teilen des Softwaresystems eingesetzt?

Mit Hilfe dieser Fragen habe ich Referenzarchitekturen von Schichten- und Subsystemarchitekturen abgegrenzt.

In Abbildung 5-5 sind sieben reine Referenzarchitekturen vermerkt. Fünf der Softwaresysteme mit Referenzarchitektur (Fallstudie 1, 4, 17, 21 und 22) wurden als eine Kombination aus Referenzarchitektur mit einer einfachen, einer erweiterten Schichtenarchitektur oder einer Subsystemarchitektur eingeordnet.

Zum Abschluss dieses Kapitels gebe ich mit Tabelle 5-1 einen Überblick über die Fallstudien. Im Anhang befindet sich eine herausnehmbare Pappe mit dieser Tabelle auf der einen und dem Komplexitätsmodell auf der anderen Seite, um den Leserinnen und Lesern für die Präsentation der Ergebnisse eine Orientierung zu geben.

Auf die Zuordnung der Fallstudie zu den Architekturstilen in Tabelle 5-1 und Tabelle A-1 werde ich in Kapitel 8 zurückgreifen, wenn ich die Ergebnisse der verschiedenen Fallstudien im Rahmen des Modells für Architekturkomplexität interpretiere.

5.6. Zusammenfassung

Auf der Basis von Grounded Theory und Theoretischem Sampling konnten in dieser Arbeit vierundzwanzig Fallstudien ausgewählt und durchgeführt werden. Dabei habe ich verschiedene Ansätze gewählt, um mich dem Thema Architekturkomplexität zu nähern. Einerseits habe ich den Sotographen bei Architekturanalysen eingesetzt und dabei quantitative Ergebnisse beim Vermessen der Ist-Architektur gewonnen. Andererseits habe ich mit den Architektinnen und Architekten diskutiert und so qualitative Erkenntnisse erzielt. Diese Vielfalt war die Voraussetzung dafür, dass das Modell für Architekturkomplexität entstehen konnte.

Tabelle 5-1: Fallstudien und Architekturstile

Nr	LOC	RLOC	Start	Architekturstil
1	26.051	10.679	2005	**WAM-Architektur** Schichtenarchitektur ohne Schnittstellen
2	33.935	13.219	2000	**WAM-Architektur**
3	36.838	16.314	2003	**Projekteigene Referenzarchitektur**
4	64.156	21.591	2003	**WAM-Architektur** Erweiterte Schichtenarchitektur ohne Schnittstelle
5	53.087	22.957	2001	**WAM-Architektur**
6	47.167	23.478	2002	Schichtenarchitektur ohne Schnittstellen
7	108.057	33.651	2001	**WAM-Architektur**
8	83.709	36.671	2003	*Subsystemarchitektur*
9	99.405	43.969	2003	**WAM-Architektur**
10	137.418	49.428	1999	**WAM-Architektur**
11	121.137	65.284	2003	Schichtenarchitektur ohne Schnittstellen
12	130.316	70.266	2001	**Projekteigene Referenzarchitektur**
13	169.567	74.344	2003	Schichtenarchitektur ohne Schnittstellen
14	188.113	93.593	2003	Schichtenarchitektur ohne Schnittstellen
15	221.284	112.675	2001	Schichtenarchitektur ohne Schnittstellen
16	238.637	120.964	2003	Schichtenarchitektur ohne Schnittstellen
17	269.609	133.095	2002	**Projekteigene Referenzarchitektur** Schichtenarchitektur ohne Schnittstellen
18	294.414	143.101	2000	Schichtenarchitektur ohne Schnittstellen
19	~500.000	~250.000	2002	Schichtenarchitektur mit Schnittstellen
20	789.938	344.911	2001	*Subsystemarchitektur*
21	1465.441	629.674	2003	**Quasar-Architektur** *Subsystemarchitektur*
22	2.919.550	1.296.455	1998	**Projekteigene Referenzarchitektur** Erweiterte Schichtenarchitektur mit Schnittstelle
23	2.869.566	1.384.260	2001	Schichtenarchitektur mit Schnittstellen
24	~14 Mio	~8 Mio	2002	Erweiterte Schichtenarchitektur mit Schnittstelle

Legende zu Tabelle 5-1:

Referenzarchitektur
Schichtenarchitektur
Subsystemarchitektur

6. Modell für Architekturkomplexität

Im diesem Kapitel stelle ich mein Modell für Architekturkomplexität – kurz: Komplexitätsmodell – vor. Es ist parallel zu den Fallstudien entstanden und nimmt Bezug auf die in den letzten Kapiteln eingeführten Konzepte für Softwarearchitektur (Kapitel 3) und den Umgang von Menschen mit komplexen Strukturen (Kapitel 4). Mithilfe dieses Modells lässt sich das Ausmaß an Architekturkomplexität in Softwaresystemen beurteilen, und es können Maßnahmen definiert werden, wie die Architekturkomplexität reduziert werden kann.

Als erstes stelle ich vor, wie Softwarequalitätsmodelle üblicherweise aufgebaut sind und wie sie konkretisiert werden. Anschließend entwickle ich das Komplexitätsmodell über Faktoren und Kriterien hin zu konkreten Fragen, die sich auf einzelne Aspekte von Architekturkomplexität beziehen.

6.1. Softwarequalitätsmodelle

Das Komplexitätsmodell orientiert sich in seinem Aufbau an Softwarequalitätsmodellen. Qualitätsmodelle dienen dazu, den allgemeinen Begriff Qualität durch die Ableitung von Unterbegriffen zu operationalisieren (Balzert 1998, S. 258). Ein relativ bekanntes Qualitätsmodell wird in der DIN ISO 9126 (DIN9126 1991) beschrieben (s. Abbildung 6-1).

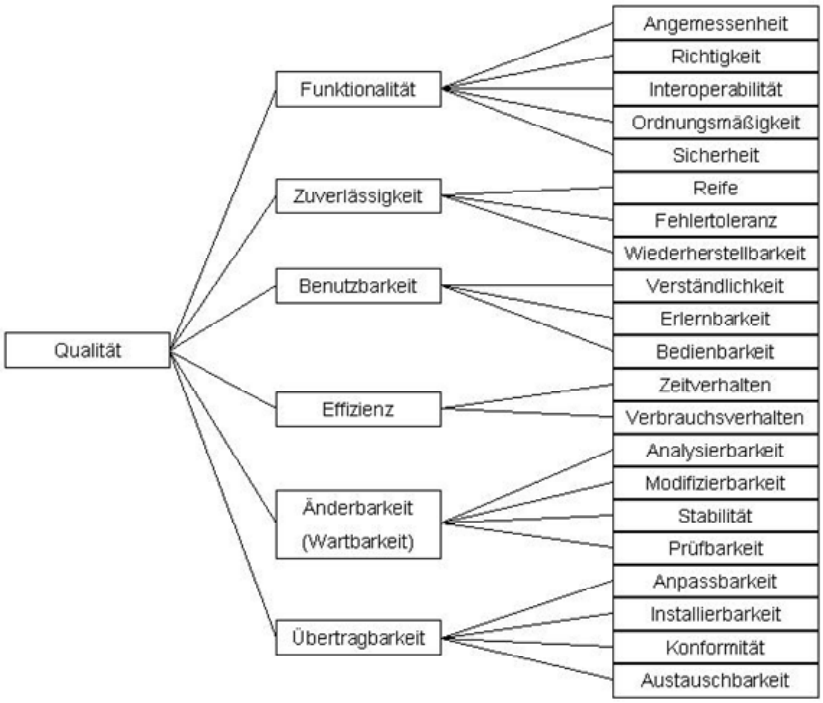

Abbildung 6-1: Softwarequalität nach der DIN ISO 9126

Ein so aufgebautes Modell bezeichnet man als Factor-Criteria-Metric-Modell (FCM-Modell) (McCall et al. 1977). Qualität wird bei einem FCM-Modell auf Faktoren zurückgeführt, von denen sie direkt abhängt. Solche Faktoren sind in

der Regel relativ abstrakt, wie beispielsweise Änderbarkeit, Effizienz und Benutzbarkeit in der DIN ISO 9126. Um die jeweiligen Faktoren weiter zu konkretisieren, werden Kriterien herausgearbeitet, wie z. B. die Kriterien Zeitverhalten und Verbrauchsverhalten für Effizienz. Die unterste Ebene eines FCM-Modells bilden schließlich Maße[16], die aus den Kriterien ableitet werden. Für das Kriterium Verbrauchsverhalten könnte beispielsweise gemessen werden, wie groß der erforderliche Plattenspeicherplatz und der benötigte Arbeitsspeicher sind. Alternativ zu Maßen können auf der untersten Ebene auch Checklisten oder andere Überprüfungstechniken eingesetzt werden (Simon 2001, S. 7f).

Um diesen letzten Schritt der Ableitung von Maßen etc. zu unterstützen, verwende ich den Goal-Question-Metric-Ansatz (GQM-Methode), der als eine Methode zur systematischen Software-Messung entwickelt wurde (Basili 1995; vgl. Basili und Weiss 1984). Diese Methode schlägt vor, dass ausgehend von einem Ziel Fragen formuliert werden müssen, die das Ziel konkretisieren. Zu den Fragen werden Maße definiert, die zur Beantwortung der Fragen beitragen sollen. Die GQM-Methode selbst ist nicht auf die Qualitätsmodellbestimmung von Softwareprodukten beschränkt, sondern lässt sich allgemein für ein vertieftes Verständnis von Prozessen, Ressourcen, Modellen und Maßen einsetzen (Simon 2001, S. 14).

Bei der Ausarbeitung des Modells für Architekturkomplexität habe ich mich an FCM-Modellen orientiert und für die Konkretisierung der Kriterien die GQM-Methode eingesetzt. Im Folgenden werde ich die Ebene der Faktoren und Kriterien sowie die aus ihnen abgeleiteten Fragen vorstellen. Maße werden im nächsten Kapitel definiert und mithilfe der Messtheorie validiert.

6.2. Faktoren zur Architekturkomplexität

Am Ende von Kapitel 3 standen die beiden Quellen für Architekturkomplexität fest: die Struktur der Ist-Architektur sowie die Abbildung von Soll-Architektur und Architekturstil auf den Programmtext. Kapitel 4 brachte die Erkenntnis, dass der Mensch mit Chunking, der Bildung von Hierarchien und dem Aufbau von Schemata auf komplexe Strukturen reagiert. Ein Softwaresystem, das von seiner Architektur her diese Prozesse unterstützt, hat geringe Architekturkomplexität.

Im Komplexitätsmodell habe ich diese beiden Stränge zusammengeführt und Architekturkomplexität durch die Faktoren Modularität, Mustertreue und Geordnetheit konkretisiert (s. Abbildung 6-2). Diese drei Begriffe korrespondieren mit den strukturbildenden Prozessen und beziehen die beiden Quellen von Architekturkomplexität ein. Mustertreue (Schemata) befasst sich mit der Abbildung auf den Programmtext. Modularität und Geordnetheit untersu-

[16] In der Literatur werden die Begriffe Softwaremaß und Softwaremetrik synonym verwendet. In Anlehnung an Zuse (Zuse 1998, S. 15ff) benutze ich den Begriff Maß. Zuse argumentiert, dass ein Maß empirische Objekte auf formale Objekte abbildet. Eine Metrik hingegen ist in einem Raum definiert und misst eine Distanz.

chen die Struktur der Ist-Architektur in ihren Elementen (Chunking) und Beziehungen (Hierarchie).

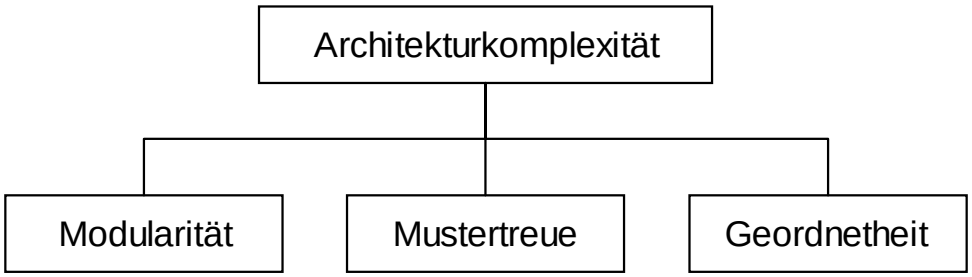

Abbildung 6-2: Ziel und Faktoren des Komplexitätsmodells[17]

Der *Aufbau von Schemata* spiegelt sich in der Softwaretechnik in der Verwendung von Mustern wieder, die im Kleinen als Entwurfsmuster und im Großen als Architekturstile vorkommen. Muster haben für die Entwicklerinnen und Entwickler den Vorteil, dass sie den Lösungsraum einschränken und vorhandene Lösungen schneller verständlich machen. Man kann also sagen, ihr Einsatz reduziert Architekturkomplexität.

Im Rahmen des Komplexitätsmodells ist aber nicht der Einsatz von Mustern die entscheidende Frage. Viel interessanter ist, ob die im Entwicklungsteam geplanten Muster tatsächlich im Programmtext umgesetzt worden sind. Lassen sich die geplanten Subsysteme und Schichten wiederfinden? Sind die Klassen, die zu bestimmten Elementarten einer Referenzarchitektur gehören, entsprechend der Architekturregeln gebaut? Um diese Fragen zu untersuchen, führe ich im Komplexitätsmodell den Faktor *Mustertreue* ein. Ein Softwaresystem weist dann geringe Architekturkomplexität auf, wenn der Programmtext konform zu den verwendeten Mustern ist.

Der zweite Faktor *Modularität* berücksichtigt den Prozess des *Chunkings*. Für Chunking ist wichtig, dass Architekturelemente als sinnvoll zusammenhängende Einheiten in der Architektur zu identifizieren sind. Kann die Entwicklerin oder der Entwickler eine Klasse oder ein Subsystem als einen Chunk betrachten, so wird das Kurzzeitgedächtnis entlastet und weitere Informationen können aufgenommen werden.

Das Schaffen von Einheiten mit einer klaren Zuständigkeit wird in der Softwaretechnik üblicherweise mit Modularität bezeichnet (vgl. Myers 1978; Nagl 1990; Parnas 1972; Parnas et al. 1985). Fragen, die im Zusammenhang mit Modularität untersucht werden müssen, sind: Ist die Architektur auf Klassen- und Subsystem-Ebene in Architekturelemente zerlegt, die jeweils als eine zusammengehörige Entwurfseinheit betrachtet werden können? Sind Schnittstelle und Implementierung der Architekturelemente so getrennt, dass die Entwicklerinnen und Entwickler nur die Schnittstelle kennen müssen, um die Dienstleistungen zu verstehen, die sie verwenden möchten? Geringe

[17] Der Einfachheit halber habe ich als Wurzelknoten des Komplexitätsmodells „Architekturkomplexität" angegeben. Streng genommen müsste hier analog zu „Qualität" als positives Ziel „geringe Architekturkomplexität" stehen.

Architekturkomplexität kann aus Sicht dieses Faktors nur erreicht werden, wenn die Architektur modular aufgebaut ist.

Der dritte kognitive Prozess, *Bildung von Hierarchien* (s. Abschnitt 4.2), wird zwar in der Softwaretechnik bei Vererbungs- und Enthaltenseins-Hierarchien berücksichtigt; bei der Benutzt-Beziehung entsteht hingegen nicht automatisch eine Hierarchie. Benutzt-Beziehungen können allein oder in Kombination mit Vererbungsbeziehungen zu verflochtenen Strukturen führen, die aus Sicht der kognitiven Psychologie schwer verständlich sind. In der Softwaretechnik wird mit Schichtenarchitekturen und Referenzarchitekturen wie Quasar und WAM genau dieser Schwachpunkt adressiert. In der Praxis gelingt es meistens nicht, vollständig hierarchische Strukturen zu implementieren, deshalb nehme ich als dritten Faktor *Geordnetheit* in das Komplexitätsmodell auf. Ist eine Architektur geordnet, so weist sie geringe Architekturkomplexität auf.

Als ich zu Beginn dieses Dissertationsvorhabens über Architekturkomplexität gesprochen und sie untersucht habe, bin ich von den drei Begriffen *Schemata*, *Chunking* und *Hierarchie* ausgegangen. Die Namen, die die Faktoren jetzt im Komplexitätsmodell haben, sind erst in einem längeren Diskussionsprozess zwischen Christiane Floyd, Claus Lewerentz und mir entstanden. Wichtig war uns, dass die drei Begriffe allgemein verständlich und positiv formuliert sind. Gut lässt sich das am Begriff *Geordnetheit* zeigen, für den die Alternativen Zyklenfreiheit oder Hierarchie in der Diskussion waren. Zyklenfreiheit verweist auf die Abwesenheit von Zyklen und scheidet somit als negativer Begriff aus. Der Begriff *Hierarchie* ist nicht allgemein verständlich, denn er wird von vielen Entwicklerinnen und Entwicklern eher mit Monohierarchien als mit Polyhierarchien in Verbindung gebracht, um die es bei der Geordnetheit geht.

Im Folgenden leite ich aus den Faktoren Kriterien ab, begründe ihre Wahl und formuliere Fragen zu den Kriterien. Die Fragen zu den Kriterien sind auf einen Vergleich zwischen verschiedenen Softwaresystemen ausgerichtet. Soll ein einzelnes Softwaresystem daraufhin untersucht werden, wie die Komplexität zwischen den in ihm enthaltenen Architekturelementen verteilt ist, müssen andere Fragen formuliert werden. Eine Abbildung des gesamten Komplexitätsmodells ist am Ende des Kapitels zu finden.

6.3. Mustertreue

Eine Ist-Architektur ist dann mustertreu, wenn die in der Soll-Architektur oder im verwendeten Architekturstil vorgegebenen Muster nicht verletzt werden. Die in dieser Arbeit untersuchten Architekturstile haben entweder eine Ausrichtung auf die Klassen- oder die Subsystem-Ebene von Architektur (s. Abschnitt 3.3.5). Mustertreue wird deshalb in zwei Kriterien verfeinert: Mustertreue auf der Subsystem-Ebene und Mustertreue auf der Klassen-Ebene.

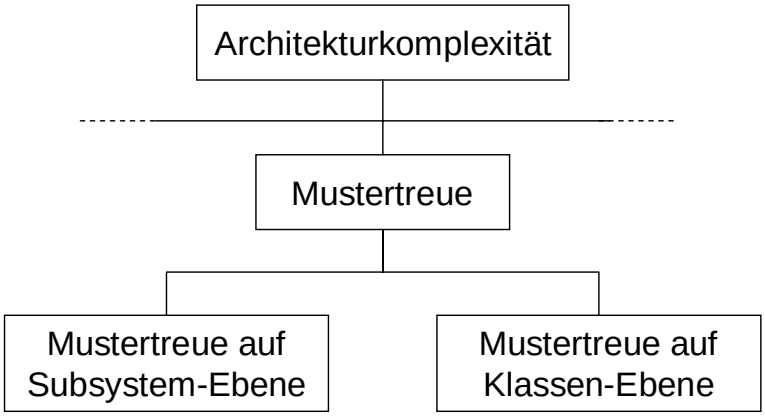

Abbildung 6-3: Kriterien zur Mustertreue

Für die Subsystem-Ebene wird untersucht, ob sich die mit der Soll-Architektur geplante Aufteilung des Softwaresystems in Subsysteme und bei einer Schichtenarchitektur ggf. Schichten und Schnitten tatsächlich im Programmtext wiederfinden lässt. Auf der Klassen-Ebene wird festgestellt, inwieweit die von der Referenzarchitektur vorgegebenen Elementarten und Regeln im Programmtext eingehalten werden.

6.3.1. Mustertreue auf Subsystem-Ebene

Für die Mustertreue auf der Subsystem-Ebene ist es wichtig, dass sich die Subsystem-Ebene der Soll-Architektur auf den Programmtext abbilden lässt. Das einzige Mittel, das heute im Programmtext in Java für die Abbildung der Subsystem-Ebene zur Verfügung steht, ist der Package-Baum (s. Abschnitt 3.4). Entspricht der Package-Baum in seinem Aufbau den Subsystem-Strukturen der Soll-Architektur, so wird die Orientierung für die Entwicklerinnen und Entwickler wesentlich erleichtert.

Für ein Softwaresystem, das mit dem Architekturstil Schichtenarchitektur entwickelt wird, ist es von großem Nutzen, wenn sich die in der Soll-Architektur geplanten Schichten und Subsysteme in der Struktur des Package-Baums wiederfinden. Abbildung 6-4 stellt den Package-Baum eines noch relativ jungen Softwaresystems mit insgesamt 51 Packages dar, die in einer Fallstudie von einem Architekten in sechs Schichten mit insgesamt sieben Subsystemen aufgeteilt wurden. Die Package-Knoten eines Subsystems haben in Abbildung 6-4 jeweils dieselbe Farbe. Eine der sechs Schichten sollte zwei Subsysteme enthalten; bei den anderen fünf fallen Schicht und Subsystem zusammen.

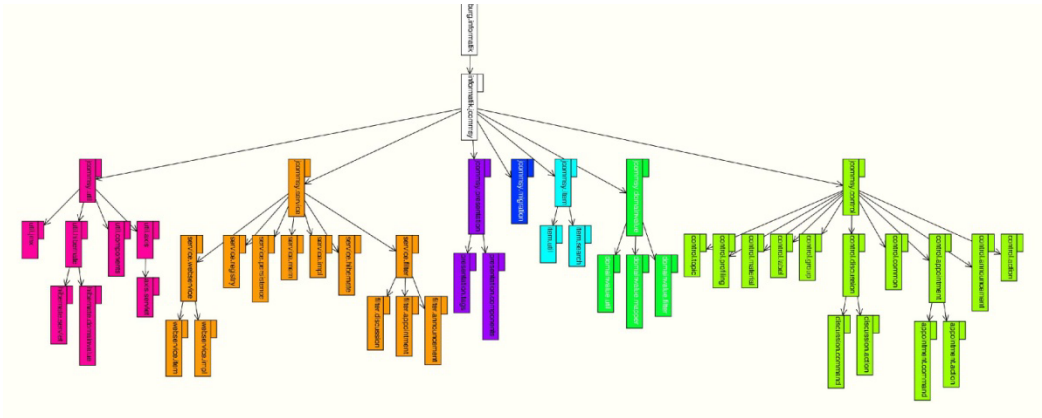

Abbildung 6-4: Package-Baum ohne Knoten für Schichten

Abbildung 6-4 zeigt, dass die Subsysteme direkt unterhalb eines gemeinsamen Knotens im Package-Baum liegen. Die von den Architekten geplanten Schichten sind im Package-Baum nicht als eigene Knoten vorhanden. Diese schwache Ausprägung der Schichten im Package-Baum begründeten die Architekten damit, dass das Softwaresystem noch relativ klein ist und Package-Knoten für Schichten bisher nicht notwendig waren.

Dieses Argument ist verständlich; für den Aufbau von Schemata ist die unscharfe Trennung zwischen Subsystemen und Schichten allerdings nicht hilfreich, so dass der Programmtext in diesem Punkt nicht mustertreu ist. Wird das Softwaresystem weiter ausgebaut, so ist zu erwarten, dass sich die Anzahl der Subsysteme pro Schicht erhöht und für die Architekten die Notwendigkeit entsteht, die Schichten als eigene Ebene im Package-Baum einzuführen. Ein solcher Schritt wird dazu führen, dass die Entwicklerinnen und Entwickler ihre bisher gebildeten Schemata anpassen müssen, was zusätzlichen Aufwand bedeutet.

Um zu untersuchen, wie gut die Abbildung der Subsystem-Ebene auf den Package-Baum gelungen ist, nehme ich die folgenden Fragen in das Komplexitätsmodell auf:

- Nach welchen Gesichtspunkten ist der Package-Baum aufgebaut?
- Finden sich die Soll-Architektur und der oder die gewählten Architekturstile im Package-Baum wieder?
- Welche anderen Muster der Strukturierung finden sich im Package-Baum?
- Welche Schwierigkeiten gab es, die Packages den Subsystemen zuzuordnen?
- Was hindert das Entwicklungsteam daran, den Package-Baum an Veränderungen in der Soll-Architektur anzupassen?

6.3.2. Mustertreue auf Klassen-Ebene

Ist eine Architektur auf der Klassen-Ebene mustertreu, so können die Entwicklerinnen und Entwickler die Elementarten, die von der Referenzarchitektur vorgegeben werden, im Programmtext leicht wiedererkennen. Dazu ist einerseits wichtig, dass die Klassen signalisieren, zu welcher Elementart sie gehören. Zum anderen müssen die Regeln der Referenzarchitektur eingehalten werden, damit die Muster zur Entfaltung kommen können. In Abbildung 6-5 sind zwei Werkzeuge (Behandlungsplaner und Patientenbearbeiter) einer WAM-Architektur zu sehen, die gemeinsam auf zwei Materialien (Behandlungsplan und Patient), einem Service (PatientenService) und einem Fachwert (dvKVNummer) arbeiten. Bis auf die Materialien verdeutlichen alle Klassen durch einen Namenszusatz, zu welcher Elementart sie gehören.

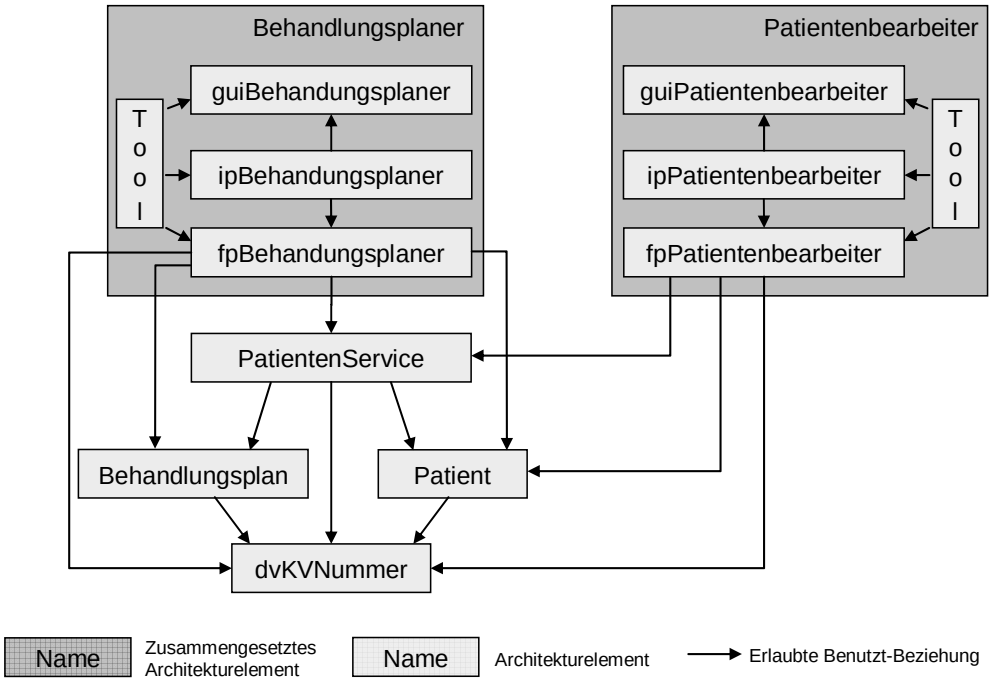

Abbildung 6-5: Beispiel einer WAM-Architektur

Um die Mustertreue auf der Klassen-Ebene zu untersuchen, füge ich im Komplexitätsmodell die folgenden Fragen ein:

- Wie wird die Zugehörigkeit einer Klasse zu einer Elementart der Referenzarchitektur signalisiert?
- Gibt es Abweichungen von den Regeln der Referenzarchitektur?

6.4. Modularität

Eine Architektur ist *modular*, wenn die Zerlegung so vorgenommen wurde, dass auf allen Abstraktionsebenen zusammenhängende Einheiten entstehen, aus denen sich Wissenseinheiten – Chunks – bilden lassen. Diese Aussage lässt sich mit den Erkenntnissen aus der kognitiven Psychologie begründen;

es schließt sich aber direkt die Frage an: Was macht eine modulare Einheit aus?

Der IEEE Standard Glossary of Software Engineering Terminology (IEEE 1990) enthält dazu die folgende Definition:

Definition 6-1: Modularität nach IEEE Standard Glossary

> "Modularity is the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components."

In der Softwaretechnik gibt es eine Reihe von relativ alten Entwurfsprinzipien, die diese Forderung nach Änderungen mit minimalen Auswirkungen einlösen wollen: Kapselung, das Geheimnisprinzip, Trennung von Schnittstelle und Implementierung (vgl. Parnas 1972), Kohäsion und lose Kopplung (Myers 1978, S. 29ff, 41ff).

Diese Entwurfsprinzipien habe ich in drei Kriterien zum Faktor Modularität geordnet (s. Abbildung 6-6):

- *Zusammenhalt*: Bildet ein Architekturelement in seinem Inneren ein zusammenhängendes Ganzes, so dass es als eine Einheit hinreichend abgeschlossen ist? (Geheimnisprinzip, Kohäsion, Kapselung)
- *Abhängigkeit*: Wie stark ist ein Architekturelement mit anderen gekoppelt, so dass es nur verstanden werden kann, wenn seine abhängigen Architekturelemente mit betrachtet werden? (lose Kopplung)
- *Schnittstellenumfang*: Sind die Schnittstellen der Architekturelemente so gestaltet, dass die Entwicklerinnen und Entwickler die Dienstleistungen des Architekturelements verstehen und nutzen können, ohne die Implementierung zu kennen? (Trennung von Schnittstelle und Implementierung)

Ein Verstoß gegen diese Kriterien trägt zu hoher Architekturkomplexität bei, wenn beispielsweise Architekturelemente willkürlich gebildet werden, mit einer beliebigen Schnittstelle ausgestattet sind und viele Beziehungen zu anderen Architekturelementen haben.

Neben diesen drei Kriterien führe ich noch ein weiteres Kriterium ein, das mir in den Fallstudien im Zusammenhang mit Modularität wichtig wurde (s. Abbildung 6-6):

- *Ausgewogenheit*: Stehen die Architekturelemente auf den jeweiligen Abstraktionsebenen in einer angemessenen Größenrelation zueinander, so dass es keine viel zu großen oder viel zu kleinen Architekturelemente gibt?

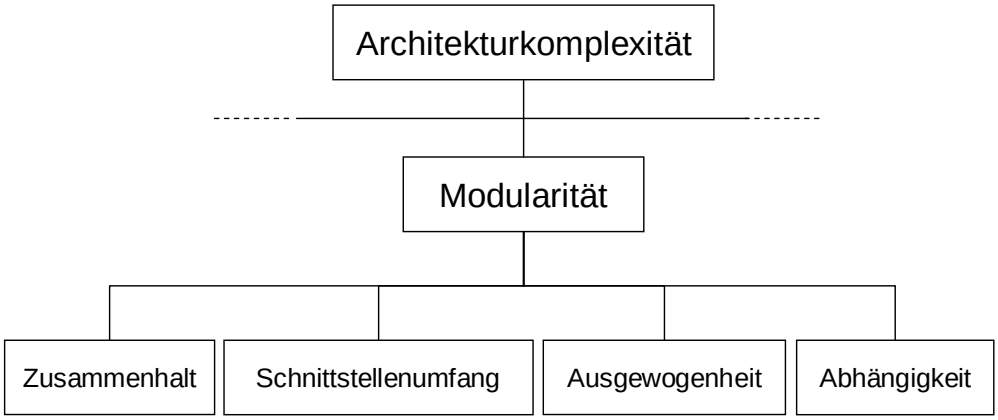

Abbildung 6-6: Kriterien zum Faktor Modularität

Diese vier Kriterien haben ohne Zweifel Wechselwirkungen untereinander. Verschiedene Autoren (Darcy 2001, S. 9ff; Darcy und Slaughter 2005, S. 985; Simon 2001, S. 55, 130ff) weisen daraufhin, dass Kohäsion (Zusammenhalt) und Kopplung (Abhängigkeit) zusammenhängende Konzepte sind. Simon macht deutlich, dass die Kohäsion von Architekturelementen auf einer Ebene von der Kopplung der in ihnen enthaltenen Elemente abhängig ist. Beispielsweise bestimmt die Kopplung zwischen den Klassen eines Packages die Kohäsion in diesem Package. Sind die Klassen stark gekoppelt, so hat das Package eine hohe Kohäsion (Simon 2001, S. 148). Ich habe die einzelnen Kriterien trotzdem getrennt, denn zu jedem dieser Kriterien lassen sich spezifische Fragen stellen, die es möglich machen, die Ursachen von Architekturkomplexität aus einer anderen Sicht zu betrachten.

6.4.1. Zusammenhalt

Mit dem Kriterium Zusammenhalt wird die Innensicht von Architekturelementen untersucht. Zu diesem Kriterium gibt es eine Reihe von Entwurfsprinzipien und -heuristiken, die alle das Ziel haben, die Entwicklung abgeschlossener und zusammenhängender Architekturelemente zu unterstützen: das Geheimnisprinzip, Kohäsion, Kapselung, Lokalität und den Entwurf nach Zuständigkeiten. Mit dem Begriff *Zusammenhalt* fasse ich das gemeinsame Anliegen dieser Prinzipien zusammen. Über die verschiedenen Wurzeln gebe ich im Folgenden einen Überblick.

Parnas forderte 1972 als erster, dass ein Modul genau eine Entwurfsentscheidung verbergen soll, und gab diesem Grundsatz den Namen Information Hiding (Geheimnisprinzip) (vgl. Parnas 1972). Mit dem Geheimnisprinzip eng verknüpft sind Begriffe wie Kapselung und Lokalität, die ausdrücken, dass eine Datenstruktur in einem Modul gekapselt wird und nur dort lokal verwendet werden darf (Nagl 1990, S.90ff). In objektorientierten Sprachen wurde dem Geheimnisprinzip mit Klassen Rechnung getragen. Inzwischen werden Kapselung, Geheimnisprinzip und Lokalität auf allen Abstraktionsebenen verwendet, und man spricht davon, dass Klassen in Subsystemen lokal sein sollen, so dass Veränderungen immer alle oder keine Klassen in

einem Subsystem betreffen ((Nagl 1990, S. 127), s. Common-Closure Principle in (Martin 2003, S. 256)). Kapselung wird oberhalb von Klassen als Prinzip verfolgt, wenn beispielsweise mit dem Entwurfsmuster „Fassade“ eine Kapsel um ein Subsystem gelegt wird (Brügge und Dutroit 2004, S. 288; Gamma et al. 1994, S. 185f).

In den 1970er Jahren arbeitete Myers seine Ideen über Entwurf aus und führte das Maß „Module Strength“ (Modulbindung) ein, um die Beziehungen innerhalb von Programmeinheiten zu bewerten (Myers 1978, S. 29ff). Myers beschrieb sieben Stufen der Modulbindung. Später wurde der heute noch gebräuchliche Begriff Kohäsion eingeführt und auf das Modul-Konzept von Parnas ausgeweitet. Coad und Yourdon erweiterten die sieben Stufen der Kohäsion bzw. Modulbildung für objektorientierte Programmiersprachen (Coad und Yourdon 1994, S. 149ff) und fügten als achte Stufe die informationelle Kohäsion hinzu, die mit ihrer Forderung nach der Einheit von Datenstruktur und Operationen das Prinzip der Kapselung integriert.

In die gleiche Richtung wie das Geheimnisprinzip und Kohäsion zielt die Entwurfsheuristik, Klassen nach Zuständigkeiten zu entwerfen: Eine Klasse ist eine Entwurfseinheit, die genau eine Verantwortung erfüllen und damit nur eine Rolle in sich vereinigen sollte. Dieses Prinzip wurde unter dem Namen „Responsibility-driven Design“ veröffentlicht (vgl. Martin 2003; Wirfs-Brock und McKean 2002) und geht auf die von Dijkstra postulierte Forderung „Separation of Concerns“ zurück (vgl. Dijkstra 1976). In der Refactoring-Bewegung werden Klassen mit zu vielen Zuständigkeiten ebenfalls als problematisch betrachtet und als ein „Code Smell“ unter dem Namen „God Class“ geführt (Riel 1996, S. 32f).

In das Komplexitätsmodell nehme ich zu diesem Kriterium die folgenden Fragen auf:

- Wird beim Klassenentwurf auf den Zusammenhalt geachtet?
- Haben Subsysteme einen Zusammenhalt?

Bei der Diskussion mit den Architektinnen und Architekten habe ich zusätzlich die Begriffe: Geheimnisprinzip, Kapselung, Lokalität, Kohäsion und Zuständigkeit verwendet.

6.4.2. Schnittstellenumfang

Das zweite Kriterium zum Faktor Modularität ist der Schnittstellenumfang, d. h. die Außensicht der Architekturelemente. Durch Schnittstellen kann Chunking erheblich unterstützt werden, weil die für Chunking benötigte Wissenseinheit in einer Schnittstelle bereits vorbereitet ist und von der Entwicklerin oder dem Entwickler nicht mehr durch Analysieren des gesamten Architekturelements selbst gebildet werden muss. Sind im Programmtext Schnittstellen vorhanden und sind sie sinnvoll gestaltet, so wird Architekturkomplexität reduziert.

Die Forschung zu Schnittstellen stammt – wie beim Zusammenhalt – bereits aus den 1960/70er Jahren, als über die Schnittstellen von Modulen gesprochen wurde (Myers 1978, S. 148f; Parnas 1972, S. 1056). In der Literatur wird eine Reihe von Vorteilen angegeben, die durch Schnittstellen entstehen:

- Verständlichkeit: Sind Schnittstellen vorhanden, so können komplizierte Implementierungsdetails vernachlässigt werden. (Jacobson 1992, S. 63; Parnas 1972, S. 1054; Siedersleben 2004, S. 16).
- Abstraktion: Schnittstellen gestatten es, die Abhängigkeiten zwischen Architekturelementen in der Schnittstelle zu konzentrieren und jede Abhängigkeit von der Implementierung zu vermeiden (Jacobson 1992, S. 379; Siedersleben 2004, S. 16).
- Wiederverwendung/Austausch: Schnittstellen erleichtern die Wiederverwendung von bewährten und den Austausch von vorhandenen Implementierungen (Bass et al. 2003, S. 54; Jacobson 1992, S. 226; Siedersleben 2004, S. 16).
- Flexibilität: Durch Schnittstellen wird die Kopplung zwischen Architekturelementen reduziert, so dass sie potentiell unabhängig voneinander geändert werden können (Jacobson 1992, S. 63; Parnas 1972, S. 1054; Siedersleben 2004, S. 16).
- Testbarkeit: Für Tests können Teile eines Softwaresystems unterhalb der Schnittstelle durch einen Platzhalter (mock object) ersetzt werden (Bass et al. 2003, S. 120)
- Parallele Entwicklung: Einzelne Entwicklungsteams können unabhängig von anderen gegen die jeweiligen Schnittstellen entwickeln (Jacobson 1992, S. 379; Parnas 1972, S. 1054).

In der Literatur zählen Schnittstellen zum grundlegenden Repertoire für Architektur sowohl auf Klassen- als auch auf Subsystem-Ebene (Bass et al. 2003, S. 211ff; Reussner und Hasselbring 2006, S. 44f; Züllighoven 2005, S. 38). Für diese Arbeit habe ich mich auf die Schnittstellen auf Subsystem-Ebene konzentriert und stelle zu diesem Kriterium die folgende erste Frage:

- Werden Schnittstellen auf Subsystem-Ebene eingesetzt?

Schnittstellen müssen angegeben und dokumentiert werden, aber wie Schnittstellen ausgestaltet werden sollen und wo sie im Programmtext und Package-Baum platziert werden sollten, wird nicht näher erläutert. Einige Autoren geben Forderungen für die Gestaltung von Schnittstellen an und verlangen, dass Schnittstellen *explizit* und *minimal* sein sollten (Horstmann 2006, S. 121; Martin 2003, S. 135f; Nagl 1990, S. 140; Züllighoven 2005, S. 38).

Explizite Schnittstellen werden in Java unterschiedlich unterstützt. Bei Klassen lässt sich die Schnittstelle dadurch modellieren, dass die Operationen der Klassen durch Sichtbarkeitsoperatoren eingeschränkt werden. Zusätzlich kann die Schnittstelle einer Klasse durch Interfaces oder abstrakte Klassen für

verschiedene Verwendungszusammenhänge explizit angegeben werden. Bei Packages kann die Schnittstelle über Sichtbarkeitsoperatoren an Klassen eingeschränkt werden.

Schnittstellen für Subsysteme oder Schichten können in Java nicht ausgedrückt werden. Siedersleben schlägt zu diesem Thema vor (Siedersleben 2004, S. 37), dass die Schnittstelle eines Subsystems im obersten Knoten des Package-Baums liegen soll, der für das Subsystem steht (s. Abschnitt 6.3.1 Mustertreue auf Subsystem-Ebene). Um diesem Aspekt des Kriteriums Schnittstellenumfang auf die Spur zu kommen, verwende ich die folgende Frage:

- Werden Schnittstellen auf der Subsystem-Ebene sichtbar gemacht?

6.4.3. Ausgewogenheit

Als dritter Faktor für Modularität wird die Ausgewogenheit der Architekturelemente untersucht. Besitzt ein Softwaresystem eine ausgewogene Modularität, so sind seine Architekturelemente ähnlich groß und enthalten vergleichbar viele Elemente.

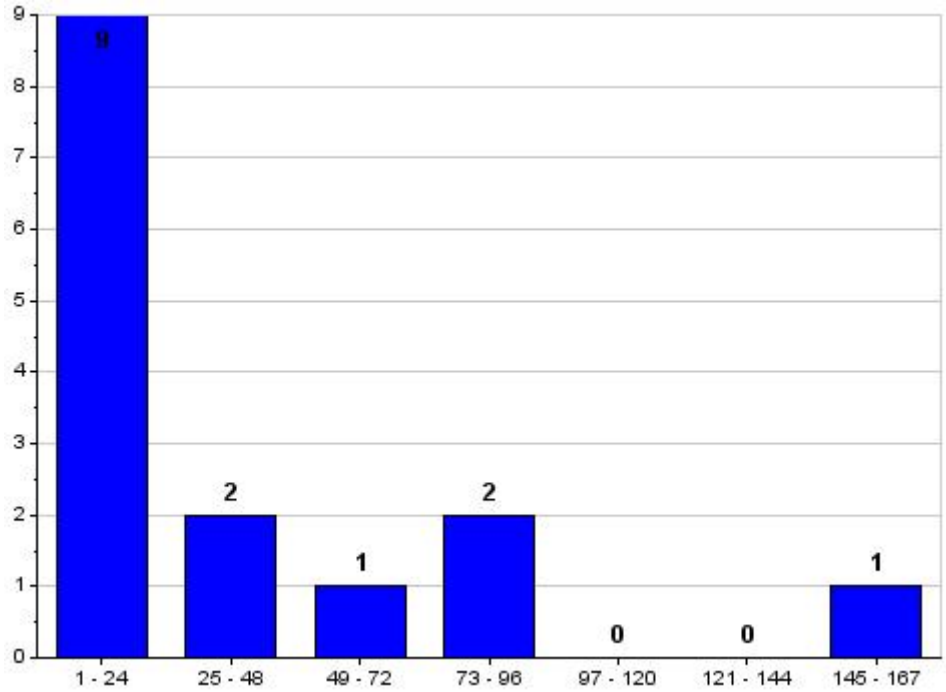

Abbildung 6-7: Anzahl Klassen pro Package

Abbildung 6-7 stellt die Verteilung von Klassen auf Packages in einem Softwaresystem dar. Auf der X-Achse ist die Anzahl der Klassen pro Package notiert und auf der Y-Achse die Anzahl der Packages, die entsprechend viele Klassen enthalten. Insgesamt elf Packages haben weniger als 50 Klassen. Drei Packages sind bereits relativ groß mit 72 und 73 bis 96 Klassen. Ein Package fällt am oberen Rand der Skala gänzlich aus dem Rahmen, weil es insgesamt

167 Klassen enthält. Auch das untere Ende der X-Achse muss genauer untersucht werden, denn 9 Packages haben zwischen 1 und 24 Klassen.

Verschiedene Autoren sprechen davon, dass Klassen und Subsysteme eine handhabbare Größe haben sollten, d.h., weder zu klein noch zu groß ausfallen dürfen. In den 1970er Jahren gab Myers als sinnvolle Modulgröße 40 bis 50 ausführbare Befehlszeilen an und legte fest, dass Ausnahmen in beide Richtungen sorgfältig begründet werden müssen (Myers 1978, S. 57f). Meyer wiederum erklärte, dass Packages zwischen 5 und 40 Klassen enthalten sollten, so dass sie von ein bis vier Personen entwickelt und von einer einzelnen Person verstanden werden können (Meyer 1995, S. 51).

Denkt man an die Randbedingungen beim Chunking zurück, so lassen sich die Forderungen nach Größenbeschränkungen und gleichmäßiger Verteilung nachvollziehen (s. Abschnitt 4.1). Um eine größere Einheit zu verstehen, müssen ihre Einzelelemente als einzelne Wissenseinheiten verarbeitet werden können. Da die Kapazität des Kurzzeitgedächtnisses begrenzt ist, darf eine große Einheit nicht zu viele Elemente enthalten.

Für das Kriterium Ausgewogenheit erweitere ich das Komplexitätsmodell um die folgende Frage:

- Sind die Klassen und Subsysteme alle ähnlich groß?

6.4.4. Abhängigkeit

Das Kriterium *Abhängigkeit* untersucht, wie vernetzt ein Architekturelement mit anderen ist. Separat verstehbare, wiederverwendbare und wartbare Einheiten in einem Softwaresystem unterstützen das Chunking und führen so zu weniger Architekturkomplexität. Um ein Architekturelement auf der Subsystem- oder Klassen-Ebene zu verstehen und ändern zu können, müssen sich die Entwicklerinnen oder Entwickler einen Überblick über die anderen Architekturelemente verschaffen, mit denen es zusammenarbeitet, und sie ggf. zusätzlich in ihren Einzelheiten verstehen (Noak 2007, S. 185). Je mehr Abhängigkeiten es von einem Architekturelement zum anderen gibt, umso schwieriger wird es, die einzelnen Beteiligten mit der begrenzten Kapazität des Kurzzeitgedächtnisses zu analysieren und passende Wissenseinheiten zu bilden.

Dieses Kriterium hat seinen Ursprung in der von Myers geführten Diskussion um das Fan-in und Fan-out von Modulen und die Kopplung zwischen Modulen (Myers 1978, S. 23f). Unter dem Fan-in eines Moduls versteht Myers die Anzahl der Module, die das Modul direkt aufrufen. Das Fan-out steht entsprechend für die Anzahl der direkt gerufenen Module. Parallel dazu führte Myers das Konzept der Kopplung ein und definierte analog zur Kohäsion sieben Stufen der Kopplung (Myers 1978, S. 41ff). Lose Kopplung ist heute ein allgemein anerkanntes Entwurfskriterium (Booch 2004, S. 113; Gamma et al. 1994, S. 24; Riel 1996, S. 18ff; Züllighoven 2005, S. 37). Fan-in/Fan-out und Kopplung haben beide das gleiche Anliegen: Abhängigkeiten in einer Architektur zu identifizieren und, wenn möglich, zu reduzieren.

Zur Abhängigkeit nehme ich die folgende Frage in das Komplexitätsmodell auf:

- Ist die Anzahl der Beziehungen auf Klassen- und Subsystem-Ebene ähnlich groß?

6.5. Geordnetheit

Eine Architektur ist dann geordnet, wenn die aggregierten Benutzt- und Vererbungsbeziehungen zwischen den Architekturelementen einen gerichteten azyklischen Graphen bilden.

In der Softwaretechnik wird ein gerichteter zyklischer Graph als Zyklus oder Zyklengruppe bezeichnet (Bischofberger et al. 2004, S. 6f; Melton und Tempero 2006, S. 1ff; Parnas 1979, S. 273/276ff; Roock und Lippert 2004, S. 40f; Zimmermann und Nagappan 2006, S. 14f). Ein Zyklus stellt genau einen Weg durch einen Graphen dar, und der Begriff Zyklengruppe wird für eine zyklische Struktur aus mehreren Pfaden verwendet. In Abbildung 6-8 sind zwei gerichtete Graphen dargestellt. Die stark umrandeten Knoten bilden eine zyklische Struktur; während die schwach umrandeten Knoten zwar zum Graph, aber nicht zur zyklischen Struktur gehören.

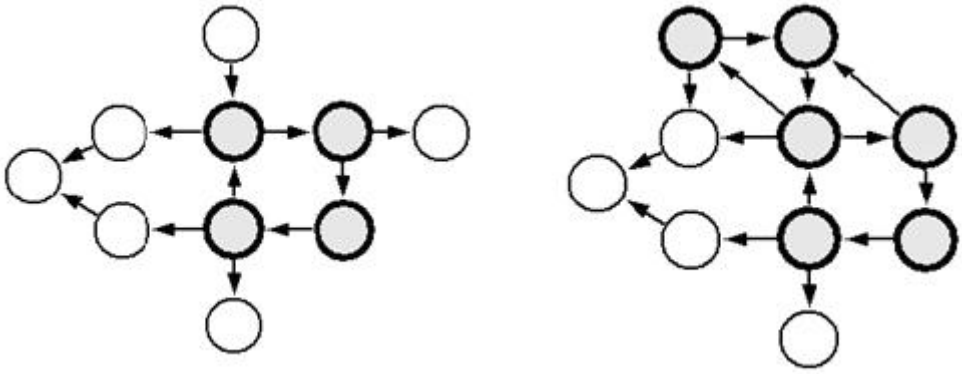

Abbildung 6-8: Zyklen im gerichteten Graphen

Die Struktur in Abbildung 6-8 links enthält einen Zyklus mit vier Knoten. In Abbildung 6-8 rechts hingegen ist eine Zyklengruppe dargestellt, die sich aus drei einzelnen Zyklen zusammensetzt. Der Einfachheit halber verwende ich in dieser Arbeit das Wort Zyklus auch für Zyklengruppen, außer, wenn eine Unterscheidung notwendig ist.

Bereits in den 1960er Jahren wies Dijkstra mit seinem Konzept von geschichteten Softwaresystemen darauf hin, dass geordnete Strukturen auf der Ebene der Benutzung zu klar getrennten Aufgabenbereichen bei der Entwicklung von Softwaresystemen führen (Dijkstra 1968, S. 344). Die Vererbungsbeziehung war zu dieser Zeit noch nicht erfunden, so dass nur die Benutzt-Beziehung betrachtet wurde. Später führte Parnas diese Argumentation fort und machte deutlich, dass Software erweiterbar bleibt, wenn sie keine Zyklen hat (Parnas 1979, S. 277).

Aufbauend auf der Argumentation von Dijkstra und Parnas werden heute eine Reihe von Begründungen angeführt, warum geordnete Strukturen Zyklen vorzuziehen sind. Diese eher technischen Begründungen nehmen die aus der kognitiven Psychologie gewonnenen Argumente gegen zyklische Strukturen zum Teil auf (Verständlichkeit, s. Kapitel 4.2) und erweitern sie (Erweiterbarkeit, Testbarkeit, Evolution, Partitionierung):

- Verständlichkeit: Architekturelemente, die in einem Zyklus sind, lassen sich nur als Ganzes betrachten, d.h., wenn man ein Architekturelement verstehen und ändern will, muss man alle verknüpften Architekturelemente untersuchen (s. Kriterium Abhängigkeit, Abschnitt 6.4.4).
  Häufig spielen an Zyklen beteiligte Architekturelemente mehrere Rollen in einer Architektur. Aus diesen Rollen erwachsen unterschiedliche Zuständigkeiten, die das Verständnis erschweren (s. Kriterium Zusammenhalt, Abschnitt 6.4.1) (Fowler 2001, S. 103f; Martin 2003, S. 256f; Roock und Lippert 2004, S. 40f, 53f, 66f; Savernik 2007, S. 357f). Hinzu kommt, dass Architekturelemente in Zyklen im Verhältnis größer als andere Architekturelemente sind, das macht sie schwerer überschaubar und damit ebenfalls schwerer verständlich (Neumann 2005, S. 94ff).
- Erweiterbarkeit: Zyklische Strukturen sind schwer zu erweitern, weil sie schwer verständlich sind. Eine parallele Weiterentwicklung von Architekturelementen kann durch Zyklen verhindert werden, weil Änderungen an einem Architekturelement direkt zu Änderungen an den mit ihm verknüpften Architekturelementen führen können. (Fowler 2001, S. 103; Lakos 1996, S. 495; Martin 2003, S. 256ff).
- Testbarkeit: Um Fehler in Softwaresystemen zu minimieren, müssen Testklassen geschrieben werden. Zyklisch verknüpfte Architekturelemente können nicht isoliert getestet werden, so dass für den Test eines Architekturelements alle am Zyklus beteiligten Architekturelemente benötigt werden (Binder 1999, S. 348, 983f; Feathers 2004, S. 4; Lakos 1996, S. 161-187).
- Evolution: Im Verhältnis zum Rest des Softwaresystems ändern sich Architekturelemente in Zyklen häufiger und sind fehleranfälliger. Zusätzlich breiten sich zyklische Strukturen aus, weil bei einer Erweiterung neue Klassen an die vorhandenen Zyklen angebaut werden (Neumann 2005, S. 94ff; Zimmermann und Nagappan 2006, S. 14f).
- Partitionierung: Soll ein Softwaresystem in eine Basisversion und Erweiterungen aufgeteilt und diese getrennt ausgeliefert werden, so dürfen ebenfalls keine zyklischen Abhängigkeiten vorhanden sein (Lakos 1996, S. 494).

Obwohl der Faktor Geordnetheit schon relativ alt ist (vgl.Dijkstra 1968), sind zyklenfreie Softwaresysteme heute immer noch eine seltene Ausnahme (Foote

und Yoder 2000, S. 656f; Melton und Tempero 2006, S. 3f). Um die durch Ungeordnetheit entstehende Architekturkomplexität zu untersuchen, verwende ich die in Abbildung 6-9 aufgeführten vier Kriterien.

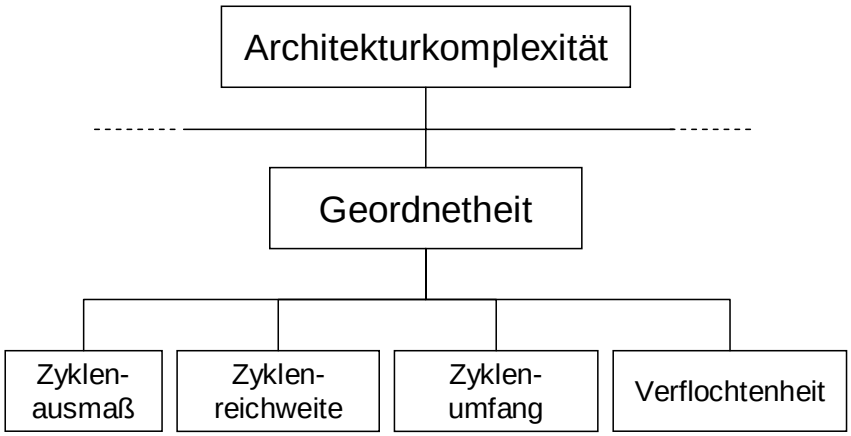

Abbildung 6-9: Kriterien zum Faktor Geordnetheit

6.5.1. Zyklenausmaß

Mit dem Kriterium Zyklenausmaß wird untersucht, wie stark Geordnetheit auf Klassen- und auf Subsystem-Ebene umgesetzt worden ist. Die Zyklen auf der Klassen-Ebene lassen sich direkt aus den Beziehungen zwischen den Klassen ablesen. Um die Zyklen auf der Subsystem-Ebene zu identifizieren, müssen die Beziehungen von der Klassen-Ebene auf die Subsystem-Ebene aggregiert werden (s. Abschnitt 3.2.2). Durch die Aggregation werden Zyklen sichtbar, die auf der Klassen-Ebene nicht vorhanden sind (s. Abbildung 6-10).

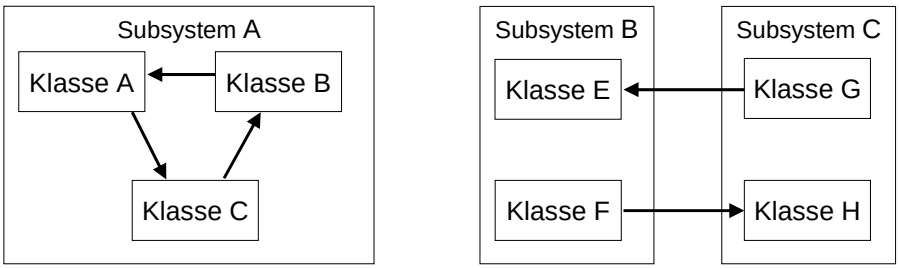

Abbildung 6-10: Klassenzyklen und Subsystemzyklen

Der Klassenzyklus links in Abbildung 6-10 ist auf der Subsystem-Ebene nicht mehr relevant, weil alle beteiligten Klassen zu einem Subsystem gehören. In Abbildung 6-10 rechts ist kein Klassenzyklus vorhanden. Die Beziehungen zwischen den Klassen der beiden Subsysteme B und C führen aber dazu, dass ein Zyklus auf Subsystem-Ebene entsteht.

Um das Ausmaß der Zyklen auf den verschiedenen Ebenen zu überprüfen, verwende ich die folgenden Fragen:

- Wie viele Klassen und Subsysteme sind in Zyklen enthalten?

- Wie viele Beziehungen verursachen die Klassen- und Subsystemzyklen?

6.5.2. Zyklenreichweite

Mit dem Kriterium Zyklenreichweite untersuche ich, ob die Klassenzyklen lokal innerhalb von einzelnen Subsystemen sind oder ob sie über deren Grenzen hinausgehen. Auf Subsystem-Ebene lassen sich Zyklen finden, die erst auf der höheren Ebene durch Aggregation entstehen (s. Abbildung 6-11 links), und im Unterschied dazu Zyklen, die direkt aus Klassenzyklen resultieren (s. Abbildung 6-11 rechts).

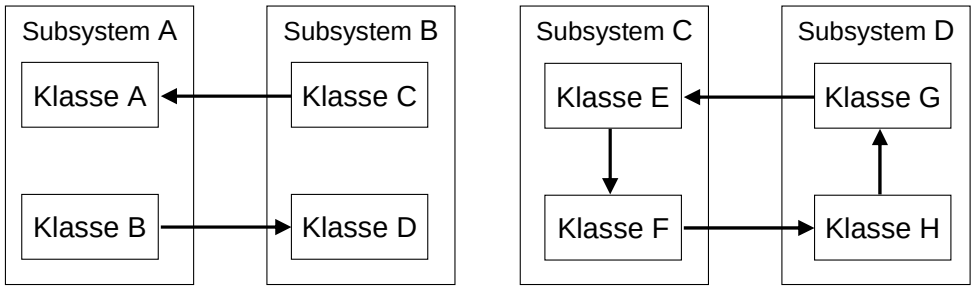

Abbildung 6-11: Zyklen auf Subsystem-Ebene

Zyklen, die nicht aus Klassenzyklen hervorgehen, sondern erst auf Subsystem-Ebene entstehen, deuten in der Regel darauf hin, dass die enthaltenen Klassen oder Subsysteme nicht den richtigen Subsystemen zugeordnet worden sind. Solche Zyklen lassen sich häufig durch Umstrukturierungen auflösen. Beispielsweise könnte man in Abbildung 6-11 links die Klasse D in das Subsystem A verschieben, und der Subsystem-Zyklus wäre verschwunden. Hingegen ist bei einem Zyklus aus Klassen in der Regel ein aufwendigeres Redesign mehrerer Klassen und ihres Zusammenspiels notwendig.

Um Subsystemzyklen erkennen zu können, die aus Klassenzyklen hervorgegangen sind, und um festzustellen, wie aufwendig ein Redesign werden wird, setze ich die folgenden Fragen ein:

- Gibt es Zyklen auf Subsystem-Ebene, die aus Klassenzyklen entstehen?
- Wie stark sind die Auswirkungen der Klassenzyklen auf der Subsystem-Ebene?

6.5.3. Zyklenumfang

Haben zwei ähnlich große Softwaresysteme einen gleich großen Anteil von Architekturelementen in Zyklen, so kann es im Extremfall sein, dass in dem einen Softwaresystem viele kleine Zyklen aus jeweils zwei Architekturelementen zu finden sind und in dem anderen ein großer zusammenhängender Zyklus existiert. Für die Entwicklerinnen und Entwickler des Softwaresystems sind viele kleine Zyklen weniger komplex, weil der Umfang der problematischen Struktur geringer ist (Binder 1999, S. 348; Lakos 1996, S. 164). Sind große

Zyklen vorhanden, so werden die zu Beginn angeführten Probleme wie Verständlichkeit, Erweiterbarkeit, Testbarkeit, Evolution und Partitionierung verstärkt. Um dieses Kriterium zu untersuchen, erweitere ich das Komplexitätsmodell um die folgende Frage:

- Wie groß sind die Zyklen auf Klassen- und Subsystem-Ebene?

6.5.4. Verflochtenheit

Neben Zyklenausmaß, Zyklenreichweite und Zyklenumfang ist mir beim Untersuchen von Softwaresystemen ein weiterer Aspekt aufgefallen. Die Architekturelemente in einem Zyklus können verschieden stark miteinander verknüpft sein, so dass der Aufwand, den Zyklus aufzulösen, stark variiert (Melton und Tempero 2006, S. 3f; Nikitina 2007, S. 15; Roock und Lippert 2004, S. 41f).

Der Zyklus in Abbildung 6-12 besteht aus sechzehn Klassen, die durch eine einzige Beziehung zwischen zwei Klassen zu einem Zyklus werden. Diese Beziehung wird lediglich durch zwei Konstanten verursacht, die in dem gesamten Softwaresystem ansonsten nicht verwendet werden. Durch das Verschieben der Konstanten in die benutzende Klasse kann dieser Zyklus aufgelöst werden.

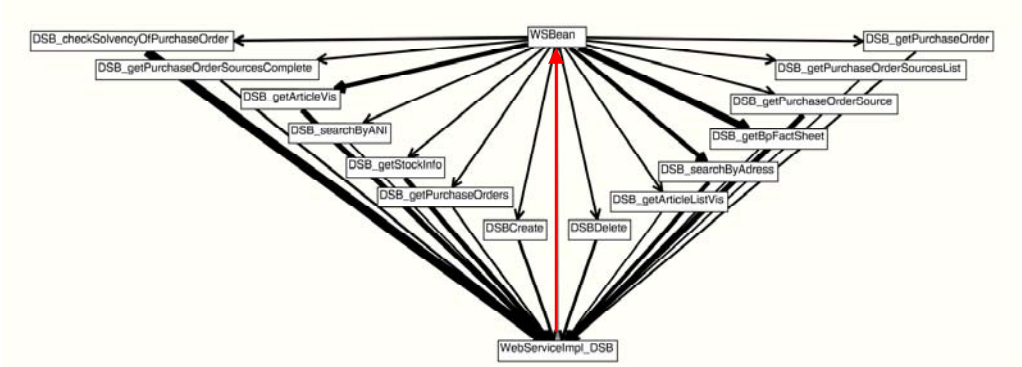

Abbildung 6-12: Wenig verflochtener Zyklus

In Abbildung 6-13 dagegen ist ein stark verflochtener Zyklus zu sehen. Obwohl er mit sieben Klassen weniger als halb so groß ist wie der Zyklus in Abbildung 6-12, hat er wesentlich mehr direkte Zyklen. Bei der Arbeit mit Zyklen habe ich die Erfahrung gemacht, dass solche Zyklen nur durch ein grundlegendes Redesign mehrerer Klassen aufzulösen sind. In manchen Fällen ist es sogar einfacher, die gesamte Funktionalität neu zu implementieren.

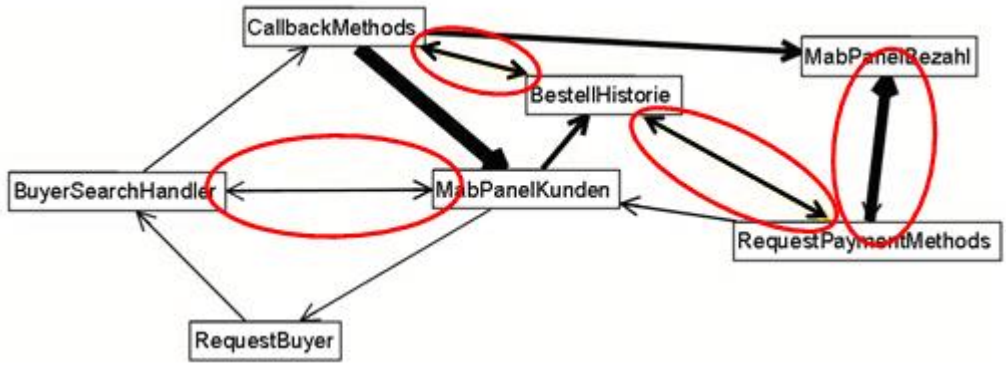

Abbildung 6-13: Stark verflochtener Zyklus

Auffällig an dieser Zyklengruppe sind besonders die vier direkten Zyklen oder auch bidirektionalen Beziehungen, die in Abbildung 6-13 rot markiert sind. Um diese bidirektionalen Beziehungen aufzulösen, müssen die Zuständigkeiten der beteiligten Klassen diskutiert und die insgesamt von diesen Klassen benötigte Funktionalität neu konzipiert werden. Um die Verflochtenheit von Zyklen zu bestimmen, verwende ich die folgende Frage:

- Wie viele bidirektionale Beziehungen sind in den Klassen- und Subsystemzyklen vorhanden?

6.6. Zusammenfassung

Im diesem Kapitel habe ich das Modell für Architekturkomplexität mit seinen drei Faktoren und zehn Kriterien entwickelt. Zu allen Kriterien habe ich Fragen formuliert, die zur Bestimmung von Architekturkomplexität in seinen unterschiedlichen Facetten beitragen. Die Faktoren, Kriterien und Fragen sind in der nachfolgenden Abbildung 6-14 zusammengefasst.

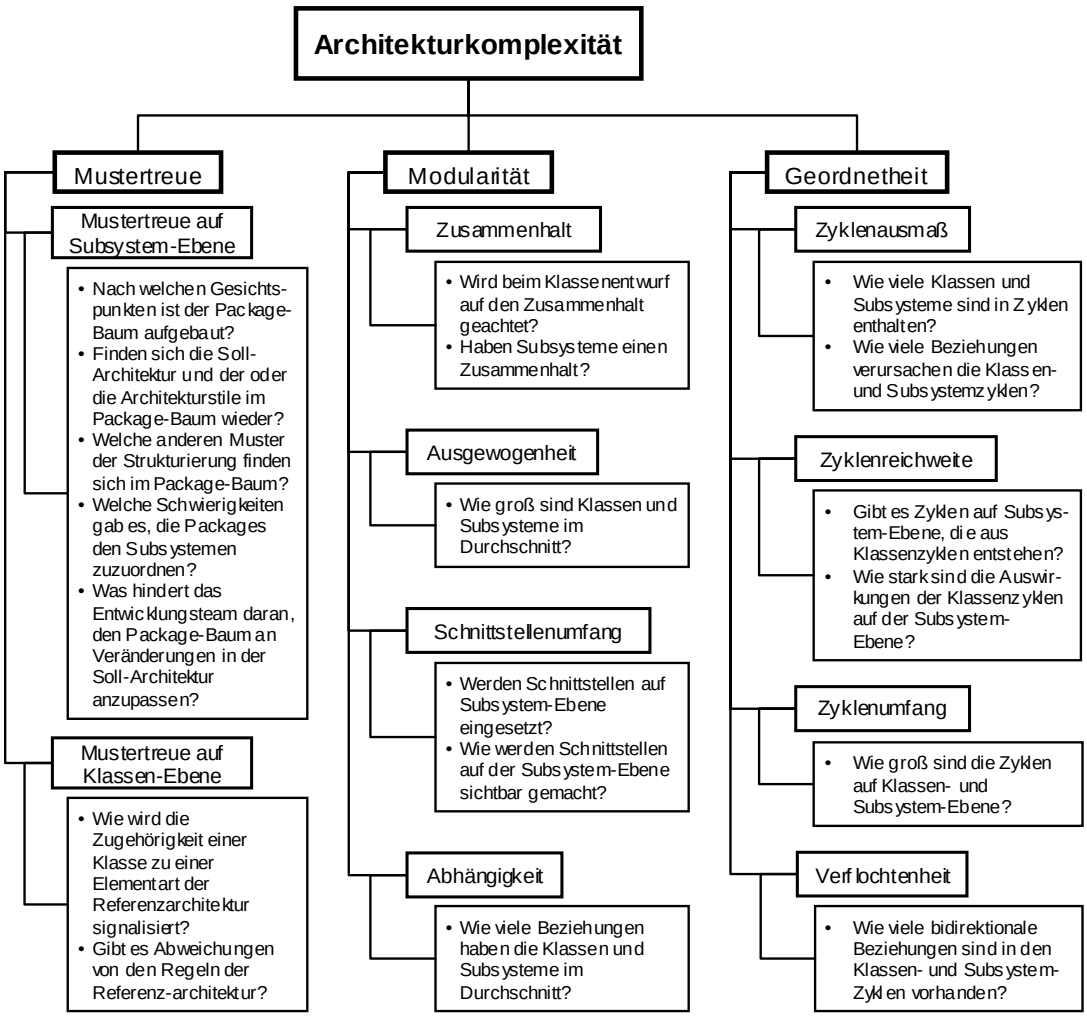

Abbildung 6-14: Modell für Architekturkomplexität

Im nächsten Kapitel werde ich einen Teil der Fragen aus dem Modell zu Maßen weiterentwickeln und schließlich in Kapitel 8 Ergebnisse präsentieren.

Mit dem Modell aus Abbildung 6-14 lässt sich die Architekturkomplexität eines Softwaresystems zu einem bestimmten Zeitpunkt untersuchen. Es entsteht also eine Momentaufnahme der Architekturkomplexität. Architekturzentrierte Softwareentwicklung ist darüber hinaus ein Prozess, der sich kontinuierlich über den gesamten Einsatz eines Softwaresystems erstreckt. Dieser prozessbezogene Aspekt kommt im Kapitel 9 und 10 zum Tragen, wenn verschiedene Stadien der architekturzentrierten Softwareentwicklung und Strategien zur Komplexitätsreduktion vorgestellt werden.

7. Maße zur Geordnetheit

Bei der Ausarbeitung des Komplexitätsmodells hat sich gezeigt, dass die Entwicklung von Maßen für alle Faktoren den Rahmen dieser Arbeit sprengen würde oder alternativ nur eine oberflächliche Bearbeitung aller Faktoren möglich gewesen wäre. Deshalb habe ich bei den Fallstudien nur für den Faktor Geordnetheit Maße eingesetzt und die anderen Faktoren mithilfe der Fragen des Komplexitätsmodells untersucht.

Im Folgenden führe ich zuerst die aus der Messtheorie benötigten Grundlagen ein und zeige, wie Maße definiert und validiert werden. Anschließend definiere ich die Maße zur Geordnetheit und validiere beispielhaft zwei Maße. Zum Abschluss diskutiere ich andere Ansätze, die Komplexität von Architekturen untersucht haben, und bewerte sie auf Basis des Komplexitätsmodells.

7.1. Grundlagen der Messtheorie

Messen ist ein Prozess, bei dem Merkmale realer Objekte auf formale Objekte abgebildet werden. Um ein Maß zu definieren, benötigt man zwei Relationensysteme: das empirische Relationensystem (ERS), das beschreibt, wie reale Objekte in Beziehung zueinander stehen, und das formale Relationensystem (FRS), das festlegt, welche Beziehung die formalen Objekte zueinander haben.

Maße werden benutzt, um Aussagen über den Zustand und um Vorhersagen über einen zu erwartenden Zustand des vermessenen Objekts zu machen. Dabei wird für Maße die sog. Repräsentationsbedingung gefordert, die besagt, dass alle Elemente des ERS auf Elemente des FRS abgebildet werden müssen und dass bei der Abbildung die Relation zwischen den Elemente des ERS für die Elementen des FRS erhalten bleiben. Ist die Repräsentationsbedingung erfüllt, so wird das Tripel aus ERS, FRS und Maß als Skala bezeichnet (Fenton und Pfleegler 1997, S. 31f; Henderson-Sellers 1996, S. 68f).

Es gibt verschiedene Skalentypen, die durch die jeweils auf ihnen zulässigen Transformationen (z.B. Addition, Multiplikation) definiert werden. Abhängig von den zulässigen Transformationen sind nur bestimmte statistische Berechnungsmöglichkeiten anwendbar. (Balzert 1998, S. 228; Bennicke und Rust 2004, S. 14ff; Fenton und Pfleegler 1997, S. 46ff; Henderson-Sellers 1996, S. 65f; Simon 2001, S. 72ff; Stevens 1946, S. 678ff; Zuse 1990, S. 43ff):

Tabelle 7-1: Skalentypen

Skalenniveau		Eigenschaft	Erlaubte Operationen	
			Mathematisch	**Statistisch**
qualitativ	Nominalskala	Reine Kategorisierung von Werten	=, ≠	Modus, Frequenz
	Ordinalskala	Skalenwerte geordnet und vergleichbar	=, ≠ , > , <	Median, Perzentil
quantitativ	Intervallskala	Werte geordnet, Distanzen bestimmbar	=, ≠ , > , <, +, -	Arithmetisches Mittel, Standardabweichung
	Verhältnisskala	Werte geordnet. Skala hat einen absoluten Nullpunkt	=, ≠ , > , <, +, - , *,/,%	Geometrisches Mittel
	Absolutskala	Skalenwerte sind absolute Größen	=, ≠ , > , <, +, - , *,/,%	

Die hier aufgeführten Skalenniveaus bauen aufeinander auf, d. h., jedes Skalenniveau besitzt auch alle Eigenschaften der darüber liegenden niedrigeren Niveaus.

7.1.1. Maßdefinition

In der Literatur hat sich bisher kein einheitliches Format durchgesetzt, wie Maße definiert werden. In dieser Arbeit wird auf eine Vorgabe des Virtuellen Software Engineering Kompetenz Center[18] zurückgegriffen, das eine umfangreiche Veröffentlichung zum Thema Messen zur Verfügung gestellt hat (vgl. Bennicke und Rust 2004). Für die Definition von Maßen werden dort die folgenden Punkte gefordert:

- Name: Bezeichnung, unter dem das Maß bei der Definition, Validierung und tabellarischen Auswertungen im Anhang referenziert wird.
- Produktmodell: Das Modell, auf dem die Auswertungen durchgeführt werden. Beispiele für Produktmodelle sind: Kontrollflussgraphen, Strukturdiagramme, Datenflussgraphen, der Quelltext selbst und objektorientiertes Metamodell.
- Erfassung: Beschreibung, nach welchem Verfahren die Werte aus dem Produktmodell ermittelt werden.
- Typ: Maße können direkt oder abgeleitet sein. Direkte Maße werden anhand eines Messobjekts ermittelt (z. B. Anzahl der

[18] www.software-kompetenz.de

Klassen in einem Paket). Abgeleitete Maße werden aus direkten oder anderen abgeleiteten Maßen berechnet (z. B. durchschnittliche Größe der Klassen eines Pakets).

- Skalenniveau: s. Abschnitt 7.1

Alle in dieser Arbeit verwendeten Maße haben als Produktmodell den Quelltext und die Abbildungsvorschrift der Soll-Architektur. Sie sind direkt und genügen einer Absolutskala. Aus diesem Grund werde ich bei den Maßdefinitionen in diesem Kapitel nur Name und Erfassung angeben.

7.1.2. Validierung

Mit der Maßvalidierung soll sichergestellt werden, dass das Maß eine gültige, numerische Charakterisierung des betrachteten Merkmals darstellt und zuverlässig Messwerte liefern kann.

Die Validierung von Maßen umfasst zwei Schritte: die interne und die externe Validierung (Bennicke und Rust 2004, S. 22f; Fenton und Pfleegler 1997, S. 107ff; Simon 2001, S. 74f; Zuse 1998, S. 71f). Bei der *internen* Validierung wird überprüft, ob die Skala und die innerhalb des FRS stattfindende Interpretation konform zur Messtheorie sind. Bei der *externen* Validierung wird untersucht, inwieweit die dem Messen zugrunde liegende Fragestellung durch die Messwerte und ihre Interpretationen beantwortet wird.

Die interne Validierung umfasst die folgenden Schritte:

1. Prüfen des Skalentyps (Bennicke und Rust 2004, S. 23; Zuse 1998, S. 130f)
   Nach der Messtheorie hat jede Skala einen Skalentyp, der die erlaubten Transformationen und statistischen Operationen festlegt (s. Tabelle 7-1).
2. Überprüfung des Effekts von atomaren Operationen (Bennicke und Rust 2004, S. 23f; Zuse 1998, S. 166ff)
   Atomare Operationen sind primitive Operationen, die auf den empirischen Objekten des ERS definiert sind. Die Reaktionen des Maßes werden bei diesem Schritt beobachtet und müssen die intuitive Vorstellung über die Veränderung der Messwerte im FRS widerspiegeln.
3. Anwendung eines Anforderungskatalogs zur Maßvalidierung (Bennicke und Rust 2004, S. 24f; Briand et al. 1996, S. 68ff; Zuse 1998, S. 399ff)
   Maße müssen Axiomen genügen, die aufgrund des intuitiven Verständnisses über Komplexitätsmaße definiert werden können.

Für die Validierung gegen einen Anforderungskatalog (Schritt 3) werden ausgehend von einem intuitiven Verständnis über das, was ein Maß messen soll, Eigenschaften aufgestellt, die das definierte Maß erfüllen muss. In der Literatur gibt es verschiedene Kataloge, die zur Validierung von Maßen entwickelt wurden (Bennicke und Rust 2004, S. 25; Briand et al. 1996, S. 69;

Zuse 1998, S. 339ff) Für die Validierung gegen einen Anforderungskatalog greife ich auf den von Briand et al. entwickelten Katalog zurück. Dieser Katalog definiert Eigenschaften für verschiedene Arten von Maßen (Briand et al. 1996, S. 71).

Die interne Validierung wird im Anschluss an ihre Definition durchgeführt. Die externe Validierung erfolgt in Kapitel 8 bei der Beschreibung der Ergebnisse, die die Vermessungen und Untersuchungen der verschiedenen Fallstudien ergeben haben.

7.2. Maßdefinitionen zur Geordnetheit

Für die vier Kriterien zur Geordnetheit wurden im letzten Kapitel Fragen formuliert. Zu diesen Fragen definiere ich im Folgenden die passenden Maße. In Abbildung 7-1 sind Fragen und Maße insgesamt dargestellt.

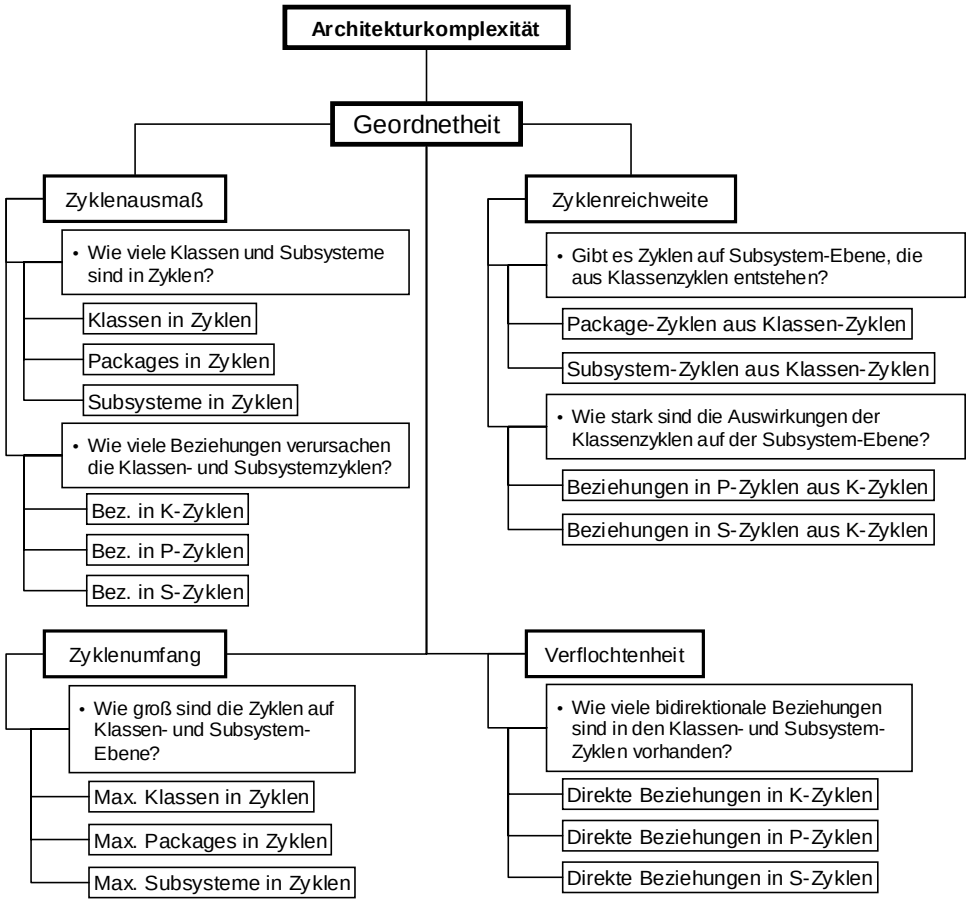

Abbildung 7-1: Maße im Komplexitätsmodell

In Kapitel 3.2.2 habe ich Packages als Java-spezifische Ausprägung der untersten Subsystem-Ebene eingeordnet. Trotzdem unterscheide ich bei den Maßen zwischen Packages und Subsystemen, so dass ich jeweils Maße auf drei Ebenen definiere. Der Grund dafür ist, dass die Softwaresysteme der Fallstudien alle in Java implementiert wurden und es so für die Entwicklungs-

teams ganz normal, war über Packages zu sprechen. Packages als Subsysteme zu bezeichnen, hätte hier eher zu Missverständnissen geführt.

7.2.1. Zyklenausmaß

Um die im Komplexitätsmodell erarbeitete Frage beantworten zu können, wie viele Klassen und Subsysteme sich in Zyklen befinden, werden drei Maße benötigt.

Name: `ClassesInCycles`
Erfassung: Zählen der Anzahl Klassen in Zyklen mithilfe der im Sotographen vorhandenen Metrik

Name: `PckgInCycles`
Erfassung: Zählen der Anzahl Packages in Zyklen mithilfe der im Sotographen vorhandenen Metrik

Name: `SubsysInCycles`
Erfassung: Zählen der Anzahl Subsysteme in Zyklen mithilfe der im Sotographen vorhandenen Metrik

Außerdem wird für das Zyklenausmaß noch überprüft, wie viele Beziehungen jeweils die Klassen-, Package- und Subsystemzyklen verursachen.

Name: `RefClassesInCycles`
Erfassung: Zählen der Anzahl der Beziehungen zwischen Klassen in Zyklen mit einer im Sotographen entwickelten Abfrage

Name: `RefPckgInCycles`
Erfassung: Zählen der Anzahl der Beziehungen zwischen Packages in Zyklen mit einer im Sotographen entwickelten Abfrage

Name: `RefSubsysInCycles`
Erfassung: Zählen der Anzahl der Beziehungen zwischen Subsystemen in Zyklen mit einer im Sotographen entwickelten Abfrage

Mit diesen sechs Maßen lässt sich feststellen, wie viele Architekturelemente in Zyklen enthalten sind und wie viele Beziehungen diese Zyklen verursachen. Durch einen Vergleich wird außerdem sichtbar, ob sich das Ausmaß der Geordnetheit auf Klassen-, Package- und Subsystem-Ebene unterscheidet.

7.2.2. Zyklenreichweite

Für die Zyklenreichweite ist die erste Frage, ob es Subsystemzyklen gibt, die auf Klassenzyklen basieren. Zwei Maße beantworten diese Frage.

Name: `PackageCycleGroupsFromClassCycles`
Erfassung: Zählen der Anzahl Package-Zyklen aus Klassenzyklen mit einer im Sotographen entwickelten Abfrage

Name: `SubsysCycleGroupsFromClassCycles`
Erfassung: Zählen der Anzahl Subsystemzyklen aus Klassenzyklen mit einer im Sotographen entwickelten Abfrage

Um die Auswirkung der Klassenzyklen auf Package- und Subsystem-Ebene zu untersuchen, wird die Anzahl der Beziehungen aus Klassenzyklen bestimmt, die über Package- bzw. Subsystemgrenzen gehen.

Name: `RefsInClassCyclesBetweenPackages`
Erfassung: Zählen der Anzahl Beziehungen zwischen Packages aus Klassenzyklen mit einer im Sotographen entwickelten Abfrage

Name: `RefsInClassCyclesBetweenSubsystems`
Erfassung: Zählen der Anzahl Beziehungen zwischen Subsystemen aus Klassenzyklen mit einer im Sotographen entwickelten Abfrage

Mit diesen Maßen kann die Frage beantwortet werden, welchen Einfluss die Zyklen der tieferen Abstraktionsebene auf die Zyklen höherer Abstraktionsebenen haben. Einerseits ist erkennbar, wie viele der Package- und Subsystemzyklen aus Klassenzyklen entstehen. Andererseits kann anhand der Anzahl der Referenzen festgestellt werden, wie stark die durch die Klassenzyklen hervorgerufene Abhängigkeit ist.

7.2.3. Zyklenumfang

Der Umfang von Zyklen wird anhand des maximalen Werts der Maße `ClassesInCycle`, `PckgInCycle` und `SubsysInCycle` untersucht. Diese Maße wurden bereits für das Kriterium Zyklenausmaß definiert.

7.2.4. Verflochtenheit

Um die Verflochtenheit von Zyklen zu bestimmen, wird die Anzahl der bidirektionalen Beziehungen in Zyklen gemessen.

Name: `DirectRefsInClassCycles`
Erfassung: Zählen aller bidirektionalen Beziehungen zwischen Klassen, die in Klassenzyklen sind, mit einer im Sotographen entwickelten Abfrage

Name: `DirectRefsInPackageCycles`

Erfassung: Zählen aller bidirektionalen Beziehungen zwischen Packages, die in Package-Zyklen sind, mit einer im Sotographen entwickelten Abfrage

Name: `DirectRefsInSubsystemCycles`
Erfassung: Zählen aller bidirektionalen Beziehungen zwischen Subsystemen, die in Subsystemzyklen sind mit einer im Sotographen entwickelten Abfrage

Mit den in diesem Abschnitt definierten Maßen wird es in Kapitel 8 möglich, die in den Fallstudien untersuchten Softwaresysteme aus verschiedenen Blickwinkeln auf ihre Geordnetheit zu beleuchten.

7.3. Validierung zum Kriterium Zyklenausmaß

Die Validierung von Maßen umfasst drei Schritte (s. 7.1.2). Bevor diese Schritte durchgeführt werden können, führe ich einige grundlegende Definitionen ein, die für die Validierung gegen den Anforderungskatalog von Briand et al. benötigt werden (vgl.Briand et al. 1996). Anschließend validiere ich beispielhaft zwei Maße `ClassesInCycles` (*CC*) und `RefClassesInCycles` (*RCC*). Diese beiden Maße habe ich gewählt, damit Beispiele für je ein Maß, das Klassen zählt, und eins, das Beziehungen zählt, vorliegen.

7.3.1. Grundlegende Definitionen

Für die interne Validierung verwende ich eine formale Definition für die Struktur von Softwaresystemen. In Briand et al. wird diese Definition eingeführt (Briand et al. 1996, S. 70, Definition 1).

Definition 7-1: Repräsentation von Systemen und Modulen

Ein System *S* wird repräsentiert als ein Paar <*E*,*R*>, wobei *E* für die Menge aller Elemente von *S* steht und *R* eine binäre Relation auf *E* bildet, die die Beziehungen zwischen den Elementen von *S* abbildet ($R \in E \times E$).

Zu einem gegebenen System *S*=<*E*,*R*> ist ein Teilsystem *m*=<E_m,R_m> ein Modul von S, wenn und nur wenn gilt
$E_m \subseteq E,\ R_m \subseteq E_m \times E_m, R_m \subseteq R$.

Die Elemente des Moduls sind mit den restlichen Elementen des Systems durch eingehende und ausgehende Beziehungen verbunden. Die Menge *InputR(m)* beinhaltet alle Beziehungen von Elemente außerhalb des Moduls *m*=<E_m,R_m> zu Elementen innerhalb des Moduls *m* und ist definiert als

$$InputR(m) = \left\{ < e_1, e_2 > \in R \,\middle|\, e_2 \in E_m \text{ und } e_1 \in E - E_m \right\}$$

Die Menge *OutputR(m)* beinhaltet alle Beziehungen von Elementen in dem Modul $m=<E_m,R_m>$ zu Elementen des restlichen Systems und ist definiert als

$$OutputR(m) = \{< e_1, e_2 > \in R \mid e_1 \in E_m \text{ und } e_2 \in E - E_m\}.$$

Um weitere Definitionen zu vereinfachen und Eigenschaften von Maßen leichter untersuchen zu können, definieren Birand et al. einige Operatoren auf Modulen (Briand et al. 1996, S. 70):

Definition 7-2: Operatoren auf Modulen

Für ein System $S=<E,R>$ und zwei Module von S $m_i =< E_{mi}, R_{mi} >$ und $mj =< E_{mj}, R_{mj} >$ gilt:

Ein Modul $m_i =< E_{mi}, R_{mi} >$ inkludiert ein Modul $mj =< E_{mj}, R_{mj} >$, wenn $E_{mi} \subseteq E_{mj}$ und $R_{mi} \subseteq R_{mj}$. Die Schreibweise für *Inklusion* ist $m_i \subseteq m_j$.

Die *Vereinigung* $(m_i \cup m_j)$ der Module $m_i =< E_{mi}, R_{mi} >$ und $mj =< E_{mj}, R_{mj} >$ ist das Modul $< E_{mi} \cup E_{mj}, R_{mi} \cup R_{mj} >$.

Ein Modul ist *leer*, wenn seine Elementmenge und seine Beziehungsmenge leer sind. $m = \langle \varnothing, \varnothing \rangle = \varnothing$.

Modul $m_i =< E_{mi}, R_{mi} >$ und Modul $mj =< E_{mj}, R_{mj} >$ sind *disjunkt*, wenn $m_i \cap m_j = \varnothing$.

Schließlich stellen Briand et al. eine weitere Definition zur Verfügung, mit der sich die Subsystem-Ebene der Ist-Architektur abbilden lässt (Briand et al. 1996, S. 70, Definition 2)

Definition 7-3: Repräsentation eines Modularen Systems

Das 3-Tupel $MS=<E,R,M>$ ist ein modulares System, wenn $S=<E,R>$ ein System gemäß Definition 7-1 ist und M eine Sammlung von Moduln, so dass gilt

$\forall e \in E \left(\exists m \in M \left(m =< E_m, R_m > \text{ und } e \in E_m \right)\right)$ und

$\forall m_i, m_j \in M \left(m_i =< E_{mi}, R_{mi} > \text{und } m_j =< E_{mj}, R_{mj} > \right.$

$\left. \text{und } E_{mi} \cap E_{mj} = \varnothing \right)$.

Diese Definitionen sind die Basis der folgenden Validierung und werden im nächsten Abschnitt auf objektorientierte Architekturen und Zyklen übertragen.

7.3.2. Objektorientierte Architekturen und Zyklen

Im Prinzip gibt es zwei Möglichkeiten, die Definitionen für Systeme und modulare Systeme auf objektorientierte Architekturen zu übertragen. Beide Übertragungen werden in diesem Abschnitt verwendet, um den von Briand et

al. zur Verfügung gestellten Anforderungskatalog einsetzen zu können. Briand et al. führen fünf verschiedene Arten von Maßen ein: Größenmaße, Längenmaße, Komplexitätsmaße, Kohäsionsmaße und Kopplungsmaße. Die ersten drei können auf Systemen (s. Definition 7-1) validiert werden, die letzten beiden nur auf modularen Systemen (s. Definition 7-3). Das eine im Folgenden zu validierende Maß *CC* ist ein Größenmaß, und das andere *RCC* ist ein Kopplungsmaß.

Die Übertragung von objektorientierten Architekturen auf Systeme *S*=<*E*,*R*> erfolgt, indem die Elemente $e_1 \ldots e_n \in E$ Klassen und die Beziehungen $r_1 \ldots r_n \in R$ als Benutzt- und Vererbungsbeziehungen zwischen Klassen betrachtet werden. Alternativ kann man die Module eines modularen Systems *MS*=<*E*,*R*,*M*> als Klassen reinterpretieren, denn Klassen sind genau wie die Module eines modularen Systems disjunkt. Die Beziehungen zwischen Klassen (Modulen) lassen sich mithilfe der Mengen *InputR(m)* und *OutputR(m)* bestimmen.

Damit sind alle Architekturelemente der Klassen-Ebene aus Abschnitt 3.2.1 abgebildet, die für die Validierung der Maße *CC* und *RCC* wichtig sind: Klassen und Beziehungen. Für Zyklen uninteressant und daher an dieser Stelle verzichtbar ist das dritte Architekturelement, die Schnittstelle. Für die Validierung von Maßen, die sich mit Schnittstellen beschäftigen, wie beispielsweise das Kriterium Schnittstellenumfang (s. Abschnitt 6.3), müssten Schnittstellen zusätzlich modelliert werden. Um die Maße für die Subsystem-Ebene zu validieren, können mithilfe der Inklusion (s. Definition 7-2) Module als Subsysteme reinterpretiert werden.

Die zu validierenden Maße zur Geordnetheit beziehen sich alle auf Zyklen. Eine Zyklengruppe wird in der Graphentheorie als starke Zusammenhangskomponente (strongly connected component, SCC) bezeichnet und ist wie folgt definiert (Clark und Holton 1994, S. 255f):

Definition 7-4: Starke Zusammenhangskomponente

> Sei $G =< N, E >$ ein gerichteter Graph mit Knoten N (Node) und Kanten $E \subseteq N \times N$. Weiterhin sei $G' =< N', E' >$ ein Teilgraph von G mit $N' \subseteq N$ und $E' \subseteq E$. *G'* heißt *stark zusammenhängend*, wenn es für jedes Knotenpaar $(a,b) \in N' \times N'$ einen Pfad von *a* nach *b* gibt. In G' ist somit jeder Knoten von jedem anderen Knoten aus erreichbar.

Diese Definition für starke Zusammenhangskomponenten lässt sich auf Systeme und modulare Systeme übertragen.

Definition 7-5: Zyklen bei Elementen und Modulen

> Die Menge aller Elemente in Zyklen $Z_E =< E_Z, R_{ZE} >$ ist ein Teilsystem eines gegebenen Systems $S =< E, R >$ gemäß Definition 7-1, für die gilt, $E_Z \subseteq E$ enthält alle Knoten aller starken Zusammen-

hangskomponenten in E und $R_{ZE} \subseteq R$ enthält alle Beziehungen aus R, die zwischen den Knoten in E_Z bestehen.

Die Menge aller Module in Zyklen $Z_M =< M_Z, R_{ZM} >$ ist eine Teilmenge der Menge der Module eines gegebenen modularen Systems $MS =< E, R, M >$ gemäß Definition 7-3, für die gilt: $M_Z \subseteq M$ enthält alle Module aller starken Zusammenhangskomponenten in M und $R_{ZM} \subseteq R$ enthält alle Beziehungen aus R, die zwischen den Modulen in E_{MZ} bestehen.

Je nachdem ob man *CC* und *RCC* für Systeme oder für modulare Systeme festlegen will, ergibt sich $CC(S) = \left| E_Z \right|$ oder $CC(MS) = \left| M_Z \right|$ und $RCC(MS) = \left| R_{ZE} \right|$ oder $RCC(MS) = \left| R_{ZM} \right|$.

Es folgen nun die drei Schritte der internen Maßvalidierung.

7.3.3. Prüfung des Skalentyps

Als Skalentyp für die Maße *CC* und *RCC* kann nur die Absolutskala angenommen werden, weil Elemente (Klassen und Beziehungen) mithilfe der natürlichen Zahlen gezählt werden.

7.3.4. Effekt von atomaren Operationen

Atomare Operationen sind die kleinstmöglichen Veränderungen in dem System, das der Berechnung des Maßes zugrunde liegt. Das Ergebnis der atomaren Operationen muss sich in das ERS als eine zulässige Aussage zurückübersetzen lassen. D. h. ihr Effekt im FRS muss mit begründeten und gewollten Effekten im ERS übereinstimmen. Ist das nicht der Fall, so muss das Maß zurückgewiesen werden. Um diesen Effekt für die Maße *CC* und *RCC* zu überprüfen, werden die folgenden atomaren Operationen definiert:

Definition 7-6: Atomare Operationen

A1: Hinzufügen einer Klasse zur Menge aller Klassen, die keine Beziehung zu einer anderen Klasse des Systems hat.

A2: Hinzufügen einer Beziehung zwischen zwei Klassen, die schon in derselben oder jeweils in einer eigenen Zyklengruppen sind.

A3: Hinzufügen einer Beziehung zwischen zwei Klassen, die noch nicht in einer Zyklengruppe sind.

A4: Hinzufügen einer Beziehung zwischen zwei Klassen, von denen eine bereits in einer Zyklengruppe ist.

A1-A4 sind atomar, weil ihr Ergebnis nicht durch eine Komposition anderer feingranularerer Operationen erreicht werden kann. Analog zu **A1-A4** lassen sich vier weitere atomare Operationen angeben, die das Entfernen von Klassen und Beziehungen behandeln. Die entfernenden Operationen werden

hier nicht weiter betrachtet, weil sie als das Inverse von **A1-A4** automatisch gültig sind, wenn die Veränderungen des Maßes durch **A1-A4** zu zulässigen Aussagen des FRS führen.

A1 erhöht die Anzahl der Elemente des Systems *S* oder der Module des modularen Systems *MS*. Dadurch entsteht weder eine Veränderung an E_Z noch an M_Z, und somit werden die Maße *CC* und *RCC* nicht beeinflusst. Dasselbe würde man für ein reales Softwaresystem erwarten.

A2 erhöht die Anzahl der Beziehungen *R* und die Anzahl der Beziehungen in Zyklen R_{ZE} oder R_{ZM}. Es kann sogar sein, dass durch A2 zwei vorhandene Zyklen zu einem größeren verknüpft werden. Trotzdem bleiben sowohl die Anzahl der Elemente im (modularen) System als auch die der Elemente in Zyklen gleich, so dass sich weder E_Z noch M_Z verändern, und somit werden die Maße *CC* und *RCC* nicht beeinflusst. Dasselbe würde man für ein reales Softwaresystem erwarten.

Um den Effekt von **A3** zu untersuchen, müssen zwei Fälle unterschieden werden. **A3** kann bewirken, dass eine neue Zyklengruppe entsteht oder dass zwei Elemente in Beziehung treten, ohne einen neuen Zyklus zu erzeugen. Auf *CC* und *RCC* hat nur der erste Fall Auswirkungen, denn der zweite Fall bewirkt lediglich eine Erhöhung von *R*, aber nicht von *E* oder *M*. Entsteht eine neue Zyklengruppe, so kann der Effekt auf *CC* und *RCC* unterschiedlich ausfallen. Besteht die neue Zyklengruppe lediglich aus diesen beiden Elementen, so erhält die Menge E_Z oder M_Z zwei neue Elemente, und sowohl *CC* als auch *RCC* erhöht sich um zwei. Der andere Extremfall ist, dass durch das Hinzufügen der einen Beziehung, alle Elemente des (modularen) Systems zu einer Zyklengruppe zusammengeschlossen werden. In diesem Fall wird $E_Z = E$ oder $M_Z = M$, und *CC* ergibt die Gesamtzahl aller Elemente bzw. *RCC* die Gesamtzahl aller Beziehungen. Zwischen diesen beiden Extremen kann sich die Anzahl der Elemente und Beziehungen in Zyklen beliebig erhöhen. **A3** führt also entweder dazu, dass *CC* und *RCC* gleich bleiben oder um 2 bis *n* erhöht wird, wenn *n* die Anzahl der Elemente oder Beziehungen ist, die vor **A3** nicht in Zyklen waren. Dasselbe würde man für ein reales Softwaresystem erwarten.

A4 kann ebenfalls verschiedene Effekte auf *CC* und *RCC* haben: *CC* und *RCC* bleiben gleich oder werden um 1 bis *n* erhöht, wenn *n* die Anzahl der Klassen ist, die vor **A4** nicht in Zyklen waren. Angenommen durch **A4** wird eine neue Beziehung von e_1 nach e_2 eingeführt. Dann wird die Zyklengruppe, zu der e_1 gehört, anschließend größer, falls e_1 vorher bereits von e_2 aus erreichbar war. Eine Erhöhung um 1 kommt dann zustande, wenn e_1 vorher direkt von e_2 erreicht werden konnte. In diesem Fall wird nur das Element e_2 an die Zyklengruppe von e_1 angeschlossen. Verliefen die Pfade von e_1 nach e_2 über verschiedene andere Elemente, so wird *CC* abhängig von der Länge dieser Pfade entsprechend größer. Dasselbe würde man für ein reales Softwaresys-

tem erwarten. Entsprechendes gilt für *RCC*, denn eine analoge Betrachtung lässt sich für Module anstellen.

Alle hier auf der Basis des FRS beschriebenen Veränderungen passen zu den erwarteten Effekten im ERS. Das bedeutet, dass die atomaren Operationen auf dem FRS sich in akzeptable Aussagen im ERS zurückübersetzen lassen; demzufolge hat das Maß *CC* diesen Test bestanden.

7.3.5. Validierung von Größenmaßen

Briand et al. definierten die Größenmaße eines Systems *S* als eine Funktion *Size(S)*, die einen Nullwert hat, nicht negativ, additiv für disjunkte Module, modular monoton und nie größer als die Größe seiner Teilsysteme ist (Briand et al. 1996, S. 71f). Im Folgenden werden diese Eigenschaften im Einzelnen vorgestellt, und das Maß *CC* wird gegen die Eigenschaften validiert.

Eigenschaft 1 (Nichtnegativität): Die Größe eines Systems *S*=<*E*,*R*> ist nicht negativ: $Size(S) \geq 0$.
Validierung: CC wird durch das Zählen von vorhandenen Elementen gewonnen und kann daher nicht negativ sein $\Rightarrow CC(S) \geq 0$.

Eigenschaft 2 (Nullwert): Die Größe eines Systems *S*=<*E*,*R*> ist Null, wenn gilt: $E = \varnothing \Rightarrow Size(S) = 0$.
Validierung: Ist $E = \varnothing$, dann enthält die objektorientierte Architektur keine Klassen, also auch keine Klassen in Zyklen. Für *CC* gilt: $E = \varnothing \Rightarrow CC(S) = 0$.

Eigenschaft 3 (Additivität von disjunkten Modulen)[19]: Die Größe eines Systems *S*=<*E*,*R*> entspricht der Größe zweier in ihm enthaltener Module, für die gilt:
$(m_1 \subseteq S \text{ und } m_2 \subseteq S \text{ und } E = E_{m1} \cup E_{m2} \text{ und } E_{m1} \cap E_{m2} = \varnothing)$
$\Rightarrow Size(S) = Size(m_1) + Size(m_2)$.
Validierung: Wird eine objektorientierte Architektur in zwei Teile geteilt, die keine gemeinsamen Klassen enthalten, so ist die Anzahl der Klassen in Zyklen in der gesamten Architektur dieselbe wie die der beiden Teilsysteme. Es gilt also $CC(S) = CC(m_1) + CC(m_2)$.

Eigenschaft 4 (Modulmonotonie): Die Größe eines Systems *S*=<*E*,*R*> wird nicht kleiner, wenn weitere Elemente hinzugefügt werden:
$(S' =< E', R' > \text{ und } S'' => E'', R'' > \text{ und } E' \subseteq E'')$
$\Rightarrow Size(S') \leq Size(S'')$.
Validierung: Wird eine Klasse zu einer objektorientierten Architektur hinzugefügt, so wird die Anzahl der Klassen in Zyklen dadurch niemals kleiner. Es gilt also $CC(S') \leq CC(S'')$.

Eigenschaft 5 (Gesamtanzahl): Die Größe eines Systems *S*=<*E*,*R*> ist nie größer als die Summe der Größe beliebiger Paare von Modulen, für die gilt:

[19] Diese Eigenschaft ist abweichend zur Definition in Briand et al. (Briand et al. 1996, S. 74f) nach Poels et al. definiert (Poels und Dedene 1997, S. 193f), die in ihrem Papier aufzeigen, dass die Definition für Additivität bei Briand et al. Inkonsistenzen aufweist.

$(m_1 \subseteq S \text{ und } m_2 \subseteq S \text{ und } E = E_{m1} \cup E_{m2})$
$\Rightarrow Size(S) \leq Size(m_1) + Size(m_2)$.

Validierung: Wird eine objektorientierte Architektur in beliebige, nicht disjunkte Teilsysteme aufgeteilt, so kann es sein, dass Klassen in beiden Teilsystemen und in Zyklen sind. Die Anzahl der Klassen in Zyklen aus den beiden Teilsystemen kann somit größer sein als die des Gesamtsystems. Gibt es keine Klassen in Zyklen, die zu beiden Teilsystemen gehören, wird *CC* denselben Wert ergeben. Kleiner kann *CC* niemals werden. Es gilt also $CC(S) \leq CC(m_1) + CC(m_2)$.

7.3.6. Validierung von Kopplungsmaßen

Briand et al. definierten die Kopplungsmaße eines modularen Systems *MS* als eine Funktion *Coupling(MS)*, die einen Nullwert hat, nicht negativ, modular monoton, zusammenlegbar und additiv für disjunkte Module ist (Briand et al. 1996, S. 78f). Im Folgenden werden diese Eigenschaften im Einzelnen vorgestellt und das Maß *RCC* gegen die Eigenschaften validiert.

Eigenschaft 1 (Nichtnegativität): Die Kopplung in einem modularen System *MS*=<*E*,*R*,*M*> ist nicht negativ: $Coupling(MS) \geq 0$.

Validierung: *RCC* wird durch das Zählen von vorhandenen Beziehungen gewonnen und kann daher nicht negativ sein $\Rightarrow RCC(MS) \geq 0$.

Eigenschaft 2 (Nullwert): Die Kopplung in einem modularen System *MS*=<*E*,*R*,*M*> ist Null, wenn gilt: $R - IR = \varnothing \Rightarrow Coupling(MS) = 0$.

Validierung: Gilt $R - IR = \varnothing$, dann sind nur Beziehungen innerhalb der Klassen einer objektorientierten Architektur vorhanden. In diesem Fall gibt es keine Beziehungen zwischen Klassen, und somit existieren keine Zyklen. Es gilt also $RCC(MS) = 0$.

Eigenschaft 3 (Modulmonotonie): Gegeben seien zwei modulare Systeme $MS' =< E, R', M' >$ und $MS'' =< E, R'', M'' >$ mit der gleichen Menge *E* an Elementen, und es existieren zwei Module $m' \in M', m'' \in M''$, so dass gilt $R'\text{–}OuterR(m') = R''\text{–}OuterR(m'')$ und $OuterR(m') \subseteq OuterR(m'')$. Dann muss gelten $Coupling(MS') \leq Coupling(MS'')$.

Validierung: Wird eine zusätzliche Beziehung zwischen zwei Klassen hinzugefügt, so wird die Anzahl der Beziehungen in Zyklen niemals kleiner, sondern nur gleich oder größer. Es gilt also $RCC(MS') \leq RCC(MS'')$.

Eigenschaft 4 (Zusammenlegbarkeit): Gegeben seien zwei modulare Systeme $MS' =< E', R', M' >$ und $MS'' =< E'', R'', M'' >$ mit $E' = E'', R' = R''$, $M'' = M' - \{m'_1, m'_2\}$, wo $m'_1 =< E_{m'1}, R_{m'1} >, m'_2 =< E_{m'2}, R_{m'2} >$ und $m'' =< E_{m''}, R_{m''} >$ mit $m'_1 \in M', m'_2 \in M', m'' \notin M', E_{m''} = E_{m'1} \cup E_{m'2}$ und $R_{m''} = R_{m'1} \cup R_{m'2}$. Dann muss gelten $Coupling(MS') \geq Coupling(MS'')$.

Validierung: Werden zwei Klassen zu einer zusammengelegt, so kann die Anzahl der Beziehungen in Zyklen nicht steigen, sondern nur gleich bleiben oder weniger werden. Entweder waren beide Klassen nicht im Zyklus, dann hat die Zusammenlegung keine Auswirkungen. Wenn eine der beiden in

einem Zyklus ist, dann ist die sich aus der Zusammenlegung ergebende Klasse hinterher auch im Zyklus. Der Fall, dass die Anzahl der Beziehungen in Zyklen sinkt, tritt ein, wenn beide in Zyklen waren. Es gilt also $RCC(MS') \geq RCC(MS'')$.

Eigenschaft 5 (Additivität von disjunkten Modulen): Gegeben seien zwei modulare Systeme $MS'=<E,R,M'>$ und $MS''=<E,R,M''>$ mit dem gleichen zugrunde liegenden System $S=<E,R>$, so dass $M''=M'-\{m'_1, m'_2\}\cup\{m''\}$ mit $m'_1 \in M', m'_2 \in M', m'' \notin M', m''=m'_1 \cup m'_2$. Wenn keine Beziehung zwischen den Elementen in m'_q und m'_2 existiert, für die gilt

$Input(m'_1) \cap Output(m'_2) = \emptyset \; und \; Input(m'_2) \cap Output(m'_1) = \emptyset$

$\Rightarrow Coupling(MS') = Coupling(MS'')$.

Validierung: Werden zwei Klassen, die nicht miteinander in Beziehung sind, zusammengelegt, so hat das keine Auswirkungen auf die Anzahl der Beziehungen. Es gilt also $RCC(MS') = RCC(MS'')$.

Durch die Anwendung des Katalogs von Briand et al. habe ich beispielhaft gezeigt, wie Maße validiert werden können.

7.4. Vergleichbare Untersuchungen

Die in der Literatur vorhandenen Untersuchungen, die mit dieser Arbeit in einem Zusammenhang stehen, lassen sich in verschiedene thematische Kategorien aufteilen:

- Definition und Vermessung von Komplexität anhand von Entwurfsdokumenten
- Definition und Vermessung von Komplexität anhand des Programmtextes
- Vermessung von einzelnen Kriterien aus dem hier vorgestellten Komplexitätsmodell ohne Fokus auf das Thema Komplexität

Diese Arbeit unterscheidet sich von allen anderen dadurch, dass ein Komplexitätsmodell aufgestellt wird, anhand dessen Maße und Fragestellungen ausgewählt wurden. Ein solch umfassendes Modell ist in der Literatur nicht zu finden.

7.4.1. Komplexität von Entwurfsdokumenten

McCabe et al. stellen in ihrem Papier drei Maße vor, mit denen die Komplexität eines Designs gemessen werden kann: Modul-Design-Komplexität, Design-Komplexität und Integrationskomplexität (McCabe und Butler 1989). Die Modul-Design-Komplexität berechnet sich aus der zyklomatischen Komplexität des Benutzungsgraphen des Moduls, der auf die Beziehungen zwischen diesem Modul und den von ihm benutzten Modulen reduziert ist. Zyklomatische Komplexität, die innerhalb des Moduls entsteht, wird weggelassen. Die Design-Komplexität eines Moduls ergibt sich aus der Summe der Modul-Design-Komplexitätswerte aller seiner benutzten Module und seiner eigenen

Modul-Design-Komplexität. Die Integrationskomplexität eines Moduls errechnet sich aus der Design-Komplexität minus der Anzahl Module plus 1. Dieser Wert gibt gleichzeitig an, wie viele Tests für dieses Modul geschrieben werden müssen. Die Autoren dieses Papiers geben keine Untersuchungsergebnisse bekannt, sondern schlagen nur die neuen Maße vor. Eine Validierung der Maße wird nicht vorgenommen.

Zhao schlägt drei neue Maße für die Komplexität eines Softwaredesigns vor (Zhao 1998). Das Produktmodell für das Vermessen ist ein Graph der architekturellen Abhängigkeiten im Softwaresystem, der auf der Basis einer formalen Spezifikation in ADL aufgestellt werden konnte. In dem Graphen sind die Komponenten die Knoten, die jeweils eingehende und ausgehende Ports haben. Zwischen den Ports der einzelnen Komponenten sind gerichtete Beziehungen vorhanden. Die Komplexitätsmaße setzen sich zusammen aus der Summe der Beziehungen zwischen den Komponenten und der inneren Komplexität der Komponenten. Zhao schlägt die Maße ebenfalls nur vor, ohne sie zu validieren oder sie an einem Beispiel anzuwenden.

Prnjat und Sacks wenden sieben Maße auf ein 150 Seiten langes Designdokument aus Text und UML Diagrammen an, das ein noch zu entwickelndes Telekommunikationssystem beschreibt (Prnjat und Sacks 2001). Alle Maße beziehen sich auf die Klassen-Ebene und werden nicht validiert.

AlSahrif et al. stellen in ihrem Artikel Maße vor, mit denen die Komplexität eines Softwaresystems zu einem möglichst frühen Punkt bei der Softwareentwicklung bestimmt werden können soll (AlSharif et al. 2004). Komplexität wird auf der Basis von Dokumenten berechnet, die die Zerlegung des Softwaresystems in Komponenten und ihre Interaktion beschreiben. Für jede Komponente muss die intra-component complexity für alle Aufrufe und Berechnungen, die innerhalb einer Komponente ausgeführt werden, und die inter-component complexity als ein Maß für das Interface der Komponente zu anderen Komponenten berechnet werden. Die Autoren probieren ihre Maße an dem KWIC-Problem (Key Word In Context) aus, das ursprünglich von Parnas als Beispiel verwendet und dann von Shaw und Garlan ausgebaut wurde (Garlan und Shaw 1994; Parnas 1972). Die Autoren stellen fest, dass der gewählte Architekturstil Auswirkungen auf die durch die Maße bestimmten Schnittstellenwerte hat. Problematisch an diesem Papier ist, dass das gewählte Beispiel sehr klein ist, so dass die berechneten Komplexitätswerte wenig Aussagekraft haben. Die Autoren sind sich dieser Problematik allerdings bewusst und verweisen auf weitere Untersuchungen. Auch in dieser Veröffentlichung werden die Maße nicht validiert.

7.4.2. Komplexität des Programmtextes

Zuse nennt sein erstes Buch „Software Complexity: Measures and Methods“ und führt eine große Anzahl von Maßen auf, die er alle als Komplexitätsmaße bezeichnet (Zuse 1990). Das Ziel dieser Sammlung ist, die einzelnen Maße entsprechend der Messtheorie zu beschreiben und zu validieren. Zuse hat nicht den Anspruch, eine Definition oder Suite für die Komplexität von

Softwaresystemen anzugeben. Vielmehr ist dieses erste Buch von Zuse eine umfassende Sammlung von nicht objektorientierten Maßen. Zuse arbeitete diese Sammlung weiter aus und ergänzte sie in seinem zweiten Buch um objektorientierte Maße (Zuse 1998). Der Begriff Komplexität taucht bei diesem Buch nicht mehr im Titel auf, wird von Zuse aber als Oberbegriff für alle von ihm beschriebenen Maße verwendet (Zuse 1998, S. 19).

Das zweite Buch, das in seinem Titel von Komplexität spricht, ist von Henderson-Sellers und heißt: „Object-oriented metrics: measures of complexity". Henderson-Sellers klassifiziert den Begriff Komplexität (s. Kapitel 2) und führt Modulmaße und Intermodulmaße ein, mit denen strukturelle Komplexität gemessen werden kann. Henderson-Sellers Buch befasst sich grundlegend mit Messtheorie, Maßen für objektorientierte Programmierung und kognitiver Psychologie als ihre Fundierung. Außerdem gibt er Empfehlungen, wie Messverfahren in Organisationen eingeführt werden können. Für diese Dissertation habe ich mich in einigen Punkten auf Henderson-Sellers abgestützt (s. Kapitel 1). Der Schwerpunkt dieser Dissertation liegt anders als der von Henderson-Sellers, da ich die Maße in einem Komplexitätsmodell anordne, an einer Reihe von Fallstudien überprüfe und schließlich Strategien für ein architekturzentriertes Vorgehen ausarbeite.

Ebert misst Komplexität sowohl in den Entwurfsdokumenten als auch im Sourcecode und stellt dafür einen Komplexitätsvektor aus sechzehn Maßen zusammen, die anhand von Diskussionen auf einem Workshop identifiziert wurden (Ebert 1995). Die sechzehn Maße werden neun Komplexitätsfaktoren zugeordnet, für die Ebert in seinem Text keine nähere Erklärung gibt. Für vier verschiedene Softwaresysteme werden Ergebnisse präsentiert. Die sechzehn Maße werden über den gesamten Entwicklungsprozess erhoben, indem die Analyse- und die Designdokumente sowie der Sourcecode vermessen werden. Damit stellt Ebert anders als diese Arbeit neben dem Sourcecode auch noch alle im Entwicklungsprozess erstellten Dokumente in den Mittelpunkt der Untersuchung. Ebert macht die interessante Feststellung, dass die beiden Projekte, bei denen es in den Dokumenten des detaillierten Designs höhere Werte für Kopplung und Größe gegeben hat als im Sourcecode, ein exzellentes Ergebnis erzielt haben. Die beiden anderen untersuchten Projekte hatten diesen Peak nicht und wurden als weniger erfolgreich eingestuft. Diese Arbeit geht über Eberts Ansatz hinaus, weil die Wahl der Maße mithilfe des Komplexitätsmodells begründet werden kann und die Anzahl der Fallstudien größer ist.

7.4.3. Aspekte von Komplexität

Chidamber und Kemerer stellten 1994 eine viel beachtete Suite aus sechs Maßen vor (Chidamber und Kemerer 1994), mit denen Design-Fehler und Bereiche, die überarbeitet werden müssen, identifiziert werden sollen. Viele spätere Untersuchungen beziehen sich auf diese erste Veröffentlichung zum Vermessen von objektorientierten Softwaresystemen. Chidamber und Kemerer validieren ihre Maße gegen die Axiome von Weyuker (vgl. Weyuker

1988) und überprüfen sie an drei Beispielen. Der entscheidende Unterschied zwischen Chidamber und Kemerers Untersuchung und dem in dieser Arbeit gewählten Ansatz ist, dass die sechs Maße lediglich die Klassen- und nicht die Subsystem-Ebene von Softwaresystemen vermessen.

Simon erarbeitet in seiner Dissertation neue Ansätze zur messwertbasierten Qualitätssicherung für praxisrelevante Systemgrößen (Simon 2001, S. 24f). Der besondere Fokus liegt neben der Auseinandersetzung mit dem Begriff Qualität auf Prozessen zur Qualitätssicherung, auf Werkzeugunterstützung, auf der Identifikation von Problemen bei der Anwendung vieler bisheriger Softwaremaße und auf der Entwicklung eines generischen Distanzmaßes. Mithilfe dieses Maßes können Softwaresysteme visualisiert werden und auf diese Weise Zusammenhänge zwischen Klassen oder Subsystemen sichtbar gemacht werden. Die Effizienz und Effektivität dieses Ansatzes wird durch empirische Arbeit belegt. Simon leistet eine umfassende Arbeit auf dem Gebiet der Kohäsions- und Kopplungsmaße, die in dieser Arbeit unter dem Faktor Modularität ihren Platz haben.

Darcy untersucht in seiner Dissertation Komplexität anhand von Kohäsion und Kopplung und begründet diese Wahl mit dem Aspekt des Chunkings aus der kognitiven Psychologie (Darcy 2001). Er vertritt die Meinung, dass Programmtexte, wenn die Kopplung erhöht wird, sehr viel schneller unverständlich werden, als wenn die Kohäsion reduziert wird. (Darcy 2001, S. 64f). Er begründet diese Behauptung damit, dass bei hoher Kopplung viele Chunks gespeichert werden müssen und das Kurzzeitgedächtnis schneller an seine Kapazitätsgrenzen stößt. Bei reduzierter Kohäsion kann es zwar sein, dass anstelle von einem Chunk mehrere Chunks gespeichert werden müssen, aber der Effekt von hoher Kopplung auf das Chunking ist nach Darcys Meinung größer. Diese Hypothese kann Darcy in seiner Arbeit anhand der von ihm erhobenen geringen Datenmenge[20] allerdings nicht belegen. Darcy nimmt keine Validierung vor und hat nur eine sehr kleine Menge an Daten zur Verfügung.

Nagappan et al. stellen in ihrem Artikel die Frage, ob sich anhand eines Satzes von Komplexitätsmaßen Vorhersagen machen lassen, welche Softwareeinheiten in einem Softwaresystem fehleranfälliger sind als andere (Nagappan et al. 2006). In der Untersuchung werden die Fehlerdatenbanken von fünf Projekten bei Microsoft analysiert und die Fehler bestimmten Softwareeinheiten zugeordnet. Das Ergebnis ist, dass auf der Basis von Komplexitätsmaßen zwar Vorhersagen über Fehler gemacht werden können, aber kein einheitliches Set von Maßen für alle Projekte verwendet werden kann. Jedes Projekt muss aus seiner Fehlerhistorie die richtige Sammlung an Maßen zusammenstellen. In

[20] Darcy beobachtet fünfzehn Studenten, die jeweils ein prozedurales (C) und ein objektorientiertes (C++) Programm bekommen, das sie erweitern müssen. Es gibt von jedem Programm vier Varianten jeweils aus einer Kombination von hoher und niedriger Kopplung und hoher und niedriger Kohäsion. Das prozedurale Programm besteht aus vier Seiten Source Code, das OO Programm aus sechs Seiten Source Code. Die Programme sind jeweils in fünf bis sechs Klassen oder Prozeduren aufgesplittet und so programmiert, dass sie unterschiedliche Kohäsion und Kopplung haben. Bei den Tests wurden verschiedene Daten erhoben: Dauer für die Veränderung, Alter der Teilnehmer, Erfahrung mit den Programmiersprachen und die Mausklicks der einzelnen Versuchspersonen wurden protokolliert.

dieser Arbeit wird Architekturkomplexität nicht in Bezug auf Fehler und eine entsprechende Vorhersage untersucht, sondern die Komplexität von verschiedenen Softwaresystemen wird in ihrem gerade aktuellen Zustand analysiert. Ein Vergleich mit Fehlerdatenbanken wäre ein weiterer Schritt.

Zimmermann und Nagappan untersuchen in ihrem Beitrag die Architektur und die Abhängigkeiten im Windows Server 2003 (28,3 Mio. LOC) daraufhin, ob die Komplexität des Abhängigkeitsgraphs zwischen den Subsystemen eine statistisch signifikante Korrelation mit der Anzahl der Fehler in diesen Subsystemen hat, die nach der Auslieferung auftraten (Zimmermann und Nagappan 2006). Die Autoren stellen Hypothesen auf, verwenden historische Daten aus den Fehlerdatenbanken von Microsoft und überprüfen ihre Hypothesen durch eine Korrelations- und eine Regressionsanalyse. Auch dieser Artikel befasst sich wie der vorherige mit Fehlervorhersage, untersucht aber ganz speziell Zyklen. In dieser Arbeit analysiere ich mit dem Komplexitätsmodell Zyklen sowohl auf Klassen- als auch auf Subsystem-Ebene, wie Zimmermann und Nagappan. Zimmermann und Nagappan kommen zu dem Ergebnis, dass große Zyklen zu mehr Fehlern im Windows Server 2003 führen.

Die beiden Artikel von Nagappan et al. sind beachtenswert, weil sie ihre Aussagen mit großen Datenmengen statistisch belegen können. Diesen Schritt habe ich nicht machen können, weil die Datenmengen aus vierundzwanzig Fallstudien für eine statistische Relevanz nicht aussagekräftig genug sind.

Melton und Tempero (Melton und Tempero 2006) stellen eine empirische zu Zyklen auf Klassen-Ebene in 85 Java-Systemen vor. Dabei messen die Autoren Zyklenausmaß, Zyklenumfang und die Verflochtenheit von Klassenzyklen. Dabei kommen sie zu zwei interessanten Ergebnissen:

- Die Autoren berechnen zur Größe der Zyklen die minimale Menge an Beziehungen (MMB), die entfernt werden müssten, um den jeweiligen Zyklus auflösen zu können. Die Berechnung der MMB ist eine weitere Art, die Verflochtenheit von Zyklen zu bestimmen. Melton und Tempero nehmen diese Berechnung vor, weil sie die Vision haben, ein Werkzeug für automatische Refactorings von Zyklen zu entwickeln. Die Diskussionen, die ich mit Architektinnen und Architekten über die Auflösung von Zyklen geführt habe, deuteten nicht darauf hin, dass es möglich ist, Zyklen anhand der MMB aufzulösen. Die MMB kann in den Diskussionen lediglich als Hinweis verwendet werden, welche Beziehungen problematisch waren.
- Die Autoren unterscheiden zwischen Zyklen, die durch Beziehungen in der Schnittstelle entstehen, und Zyklen, die sich bilden, weil eine Klasse in ihrem Inneren, ohne dass es an ihrer Schnittstelle sichtbar wird, andere Klassen verwendet. Die Ergebnisse zeigen, dass Klassen, die nur im privaten Teil von anderen Klassen vorkommen, die hauptsächlichen Verursacher von großen Zyklen sind. Melton und Tempero vermuten, dass das

> Bemühen darum, interne Implementierungsdetails aus der Schnittstelle herauszuhalten, zu zyklischen Beziehungen führen kann. In diesem Zusammenhang ist das in den letzten zwei Jahren aufgekommene Prinzip „Dependency Injection" ein entscheidender Fortschritt, weil es diese Abhängigkeiten an der Schnittstelle deutlich macht.

Als die Arbeit von Melton und Tempero erschien, waren die Auswertungen für diese Arbeit bereits abgeschlossen. Zyklen, die nur innerhalb einer Klasse und an ihrer Schnittstelle sichtbar sind, werde ich in weiteren Auswertungen des Datenmaterials vornehmen.

7.5. Zusammenfassung

Für den Faktor Geordnetheit wurden in diesem Abschnitt dreizehn Maße definiert, die die vier Kriterien messbar machen. Zwei dieser Maße habe ich beispielhaft validiert, um sicher zu gehen, dass die durch sie stattfindende Interpretation konform zu Messtheorie ist. Die für Definition und Validierung herauskristallisierte Arbeitsweise wurde so beschrieben, dass sie in späteren Arbeiten auf die anderen Maße zur Geordnetheit und auf weitere Maße für die anderen Faktoren übertragen werden kann.

Im Anschluss an Definition und Validierung habe ich einen Überblick über andere Arbeiten zu Komplexität gegeben. In diesem Gebiet wird zum einen über die Komplexität von Entwurfsdokumenten und zum anderen über die Komplexität von Programmtexten geforscht. In keiner Arbeit ist ein ähnliches Komplexitätsmodell zu finden, es gibt aber eine Reihe interessanter Ergebnisse, die sich in das Komplexitätsmodell einordnen lassen.

Im nun folgenden Kapitel werden die Maße eingesetzt, um Architekturkomplexität von Softwaresystemen zu untersuchen. Im Laufe der Auswertungen wird sich zeigen, ob die dreizehn Maße tatsächlich etwas Sinnvolles messen; denn die hier durchgeführten Schritte der Validierung sind aus Sicht der Messtheorie nur eine notwendige, aber keine hinreichende Bedingung dafür, dass ein Maß verwendbar ist (Briand et al. 1996, S. 84).

8. Komplexität in der Praxis

In diesem Kapitel werden die Ergebnisse aus der Untersuchung zur Architekturkomplexität vorgestellt. Die Präsentation der Ergebnisse ist entlang der drei Faktoren des Komplexitätsmodells aufgebaut und kann mithilfe der herausnehmbaren Pappe gut verfolgt werden.

8.1. Mustertreue

Mit dem Faktor Mustertreue wird untersucht, ob ein Softwaresystem getreu der in der Soll-Architektur und dem Architekturstil festgelegten Mustern implementiert wurde. Mustertreue auf der Klassen-Ebene konnte ich nur bei Referenzarchitekturen untersuchen, denn allein bei diesem Architekturstil werden durchgängig Muster auf der Klassen-Ebene vorgegeben. Mustertreue auf der Subsystem-Ebene war bei allen Fallstudien analysierbar.

8.1.1. Mustertreue auf der Subsystem-Ebene

Wie mustergetreu die Soll-Architektur auf den Package-Baum übertragen wurde, konnte ich feststellen, wenn wir den Package-Baum im Sotographen in Subsysteme und Schichten aufgeteilt haben (s. Abschnitt 5.4.1). Im Folgenden gehe ich zuerst auf die Gestaltung des Package-Baums ein. Anschließend beschreibe ich die Schwierigkeiten bei der Zuordnung, und zum Abschluss gebe ich einen Überblick über die Probleme der Entwicklungsteams mit Änderungen im Package-Baum.

Grundsätzlich waren die ersten zwei bis vier Knoten des Package-Baums nach einem der zwei folgenden Schemata aufgebaut: `org.systemname` oder `land.firma.(abteilung.)systemname`. Die Package-Bäume wiesen bei den kleineren Softwaresystemen eine Tiefe von sechs verschachtelten Packages auf. Bei den größten Softwaresystemen wurde eine maximale Tiefe von zehn verschachtelten Packages erreicht. Insgesamt wuchs der Package-Baum bei den größeren Fallstudien eher in die Breite als in die Tiefe. Bei Fallstudie 22 hatte der Package-Baum mehr als 1.000 Blätter, bei Fallstudie 23 ca. 800 Blätter.

Die Struktur unterhalb der einleitenden Packages wurde deutlich vom eingesetzten Architekturstil beeinflusst. Im Folgenden stelle ich die für die einzelnen Architekturstile typischen Package-Bäume vor.

Bei den Schichtenarchitekturen gab es zwei Varianten, die in Abbildung 8-1 vereinfacht dargestellt sind. Entweder kamen zuerst Package-Knoten für Schichten und darunter fachlich zusammengehörende Subsysteme, oder die umgekehrte Sortierung war gewählt worden.

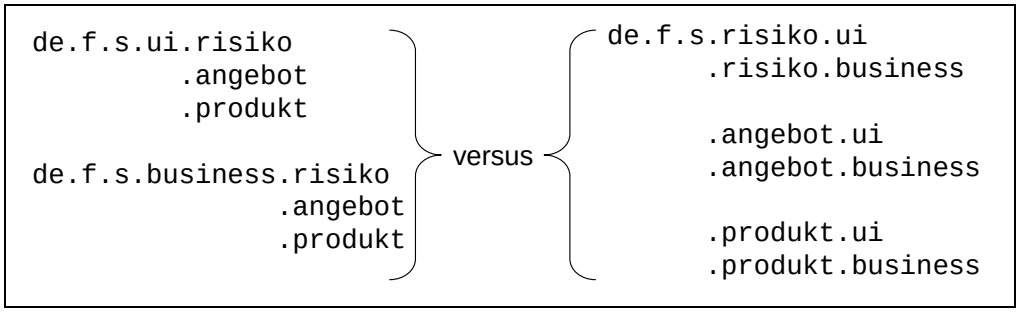

Abbildung 8-1: Package-Baum bei Schichtenarchitekturen[21]

Fallstudie 13, 14 und 16 folgten dem ersten Muster und Fallstudie 6, 11, 15, 18, 19 und 23 dem zweiten. Mehrere Architektinnen und Architekten sprachen auf der einen Seite von technischen Schichten, wie User-Interface-Schicht, Business-Schicht, Datenbankzugriffsschicht etc., und auf der anderen Seite von fachlichen Subsystemen, wie zum Beispiel Risikoberechnung, Angebotserstellung, Produktdefinition etc. Offensichtlich betrachten diese Architektinnen und Architekten die Schichtung als eine grundsätzliche Struktur, die unabhängig vom Gegenstandsbereich ist. Die fachlichen Subsysteme hingegen bilden sie auf Basis des Gegenstandsbereichs.

Der Package-Baum der Referenzarchitekturen war bei 11 von 12 Fallstudien an den Elementarten, wie Werkzeug, Material, Service etc. orientiert (Fallstudie 1, 2, 3, 5, 7, 9, 10, 12, 17, 22). Unterhalb dieser Package-Knoten waren direkt fachliche Subsysteme zu finden (s. Abbildung 8-2).

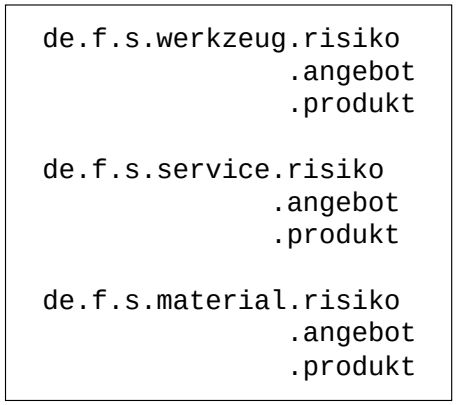

Abbildung 8-2: Package-Baum bei Referenzarchitekturen

Bei zwei Fallstudien mit Referenzarchitekturen (Fallstudie 4 und 22) handelt es sich um Softwaresysteme mit einer erweiterten Schichtenarchitektur (s. Abschnitt 3.3.3 und Tabelle A-1). In ihrem Package-Baum waren zusätzlich die fachlichen Schnitte zu erkennen. Bei Fallstudie 22 wurde die obere Variante aus Abbildung 8-3 gewählt, bei Fallstudie 4 die untere.

[21] In der Darstellung des Package-Baums steht f für den Namen der Firma, und s für den Namen des Softwaresystems.

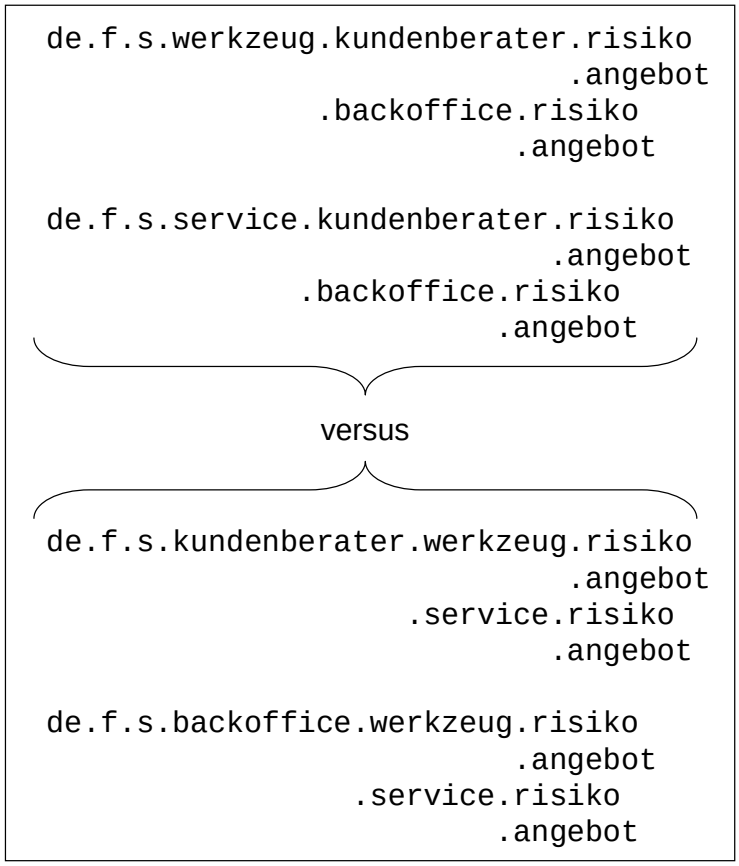

Abbildung 8-3: Package-Baum mit fachlichen Schnitten

Neben den zwei Referenzarchitekturen aus den Fallstudien 4 und 22 hat Fallstudie 24 eine erweiterte Schichtenarchitektur. Für dieses Softwaresystem ist der Package-Baum analog zu der unteren Variante aus Abbildung 8-3 aufgebaut, wobei anstelle von Package-Knoten, wie „werkzeug“ oder „service“, die Schichten zur Strukturierung verwendet werden.

Die Fallstudien mit Subsystemarchitekturen hatten einen Package-Baum, der als Knoten entweder fachliche Subsysteme (Fallstudie 8 und 20) oder zuerst eine Client-Server-Trennung und dann fachliche Subsysteme enthielt (Fallstudie 21, s. Abbildung 8-4).

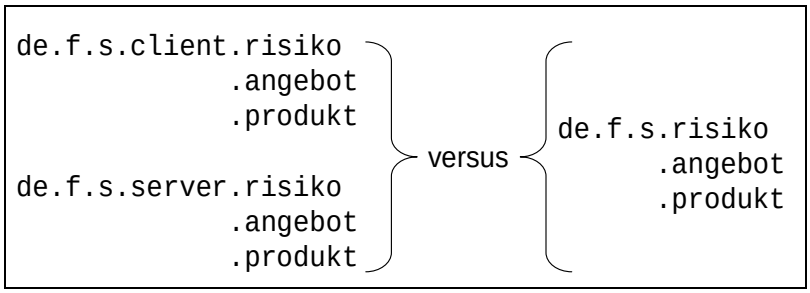

Abbildung 8-4: Package-Baum bei Subsystemarchitekturen

Bei der Hälfte aller Fallstudien entsprach der Package-Baum in weiten Teilen nicht der geplanten Strukturierung. Die Subsysteme setzten sich aus verschie-

denen Teilbäumen zusammen, die zum Teil auf verschiedenen Ebenen des Package-Baums lagen. Oder es gab Package-Knoten, die nach Aussage der Architekten Klassen aus verschiedenen Subsystemen enthielten. Abbildung 8-5 zeigt einen Package-Baum, bei dem zwei Subsysteme (lila und orange) auf verschiedenen Ebenen im Package-Baum zu finden sind. Als zusätzliche Schwierigkeit sind sie zum Teil unterhalb von Package-Knoten angesiedelt, die den Wurzel-Knoten anderer Subsysteme (blau und grün) bilden.

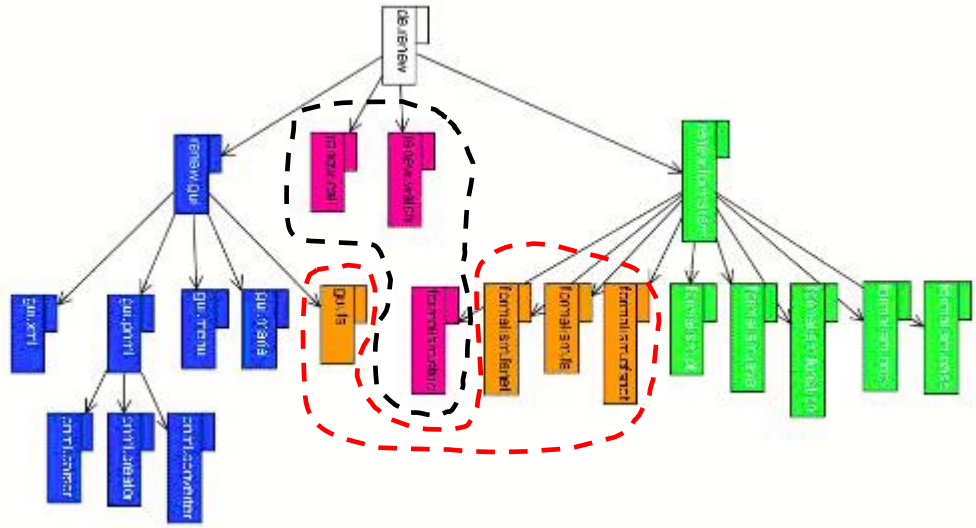

Abbildung 8-5: Subsysteme im Package-Baum[22]

In zwei Fällen (5, 11) waren auf der obersten Knoten-Ebene gleichzeitig fachliche Subsysteme und technische Schichten zu finden. Die Architekten dieser Softwaresysteme sahen die Ursache darin, dass einzelne Teammitglieder, die für einen Teilbereich des Softwaresystems zuständig sind, persönliche Vorstellungen über den Aufbau des Package-Baums hatten.

Auffällig war, dass besonders die großen Softwaresysteme ab Fallstudie 17 gut strukturierte Package-Bäume hatten. Bei ihnen war die Abbildung der Soll-Architektur auf den Package-Baum einfacher als bei einem Großteil der kleineren Fallstudien. Die Architektinnen und Architekten gaben an, dass ihr Team früh feste Regeln aufgestellt hat, wie der Package-Baum gestaltet sein soll, und die Einhaltung der Regeln ständig überprüft.

Wenn bei der Architekturanalyse deutlich wurde, dass der Package-Baum nicht der geplanten Strukturierung entsprach, konnte die letzte Frage zur Mustertreue auf der Subsystem-Ebene diskutiert werden: Was hindert das Entwicklungsteam daran, den Package-Baum an Veränderungen in der Soll-Architektur anzupassen?

Die Architektinnen und Architekten gaben hierfür eine technische und eine organisatorische Begründung. Technisch war ein Restrukturieren des Package-Baums bei den Teams schwierig, die ein restriktives Sourcecode-

[22] Die Kästen stehen für Package-Knoten und die Pfeile stellen die Enthaltenseins-Beziehung zwischen den Packages dar. Packages, die zu einem Subsystem gehören, haben die gleiche Farbe.

Management-System einsetzten, das Klassen im Package-Baum verschiebt, indem es sie am Ursprungsort löscht und am Zielort als neue Klasse wieder hinzufügt. Durch das Löschen und Neuanlegen geht die Historie der verschobenen Klasse mit allen Änderungen verloren. Aus diesem Grund wurden in einigen Fallstudien ungern Klassen im Package-Baum verschoben und ein Architekt ging sogar so weit zu sagen, dass der einmal entstandene Package-Baum nicht geändert werden dürfe.

Als zweiten Grund führten die Architektinnen und Architekten organisatorische Schwierigkeiten an. Aus ihrer Sicht ist die Restrukturierung des Package-Baums ein großes Refactoring, das mit einer Reihe von Komponenten-, Integrations- und Anwender-Tests abgesichert werden müsste. Für eine solch umfassende Veränderung, die die Funktionalität des Softwaresystems nicht erweitert, sondern „nur" seine Struktur verbessert, waren keine Zeit und kein Geld vorhanden.

Für die Mustertreue auf der Subsystem-Ebene bleibt festzuhalten, dass in allen Fallstudien der Package-Baum verwendet wurde, um die Soll-Architektur und den Architekturstil sichtbar zu machen. Allerdings ließ sich die geplante Architektur in der Hälfte der Fallstudien nur teilweise im Package-Baum wiederfinden. Gerade die großen Fallstudien waren mustergetreuer als die kleinen. Außerdem gab es für die Abbildung keine einheitlichen Muster, so dass die einmal gebildeten Schemata sich kaum zwischen Projekten übertragen lassen.

Diese Diversität lässt darauf schließen, dass das richtige Ausdrucksmittel für die Abbildung der Soll-Architektur auf den Programmtext fehlt. Im Java-Umfeld werden seit einiger Zeit Versuche unternommen, Subsysteme als Superpackages[23] oberhalb von Packages verfügbar zu machen.

8.1.2. Mustertreue auf der Klassen-Ebene

Die Mustertreue auf Klassen-Ebene war den Architektinnen und Architekten von Referenzarchitekturen ein wichtiges Anliegen. In den Fallstudien konnte ich drei verschiedene Ansätze beobachten, wie für Klassen im Programmtext die Elementart sichtbar gemacht wird: Namenskonvention, Strukturkonvention und Typisierung. In den meisten Teams wurden diese drei Ansätze in Kombination verwendet, so dass jeweils einige Elementarten an Typisierung und Namenskonvention und andere an Struktur- und Namenskonvention erkennbar waren.

Setzt ein Team *Namenskonvention* ein, so haben die Klassen Namen, aus denen sich schließen lässt, zu welcher Art sie gehören. In den WAM-Architekturen hatten alle Klassen mit Ausnahme der Materialien einen Namenszusatz, der auf ihre Elementart hinweist: dv – Fachwert (domainvalue), fp - Funktionskomponente (function part), ip – Interaktionskomponente (interaction part), tool – Werkzeug, gui – UserInterface-Klasse. In der Quasar-Architektur wurde ein ähnlich reichhaltiges Repertoire an Kürzeln verwendet: Type – Datenty-

[23] Java Specification Request 294 auf jcp.org

pen, Bo – Entitätstypen (BusinessObject), Bo{*}Mgr – Verwalter (BusinessObjectManager), Uc – Anwendungsfälle (UseCase) und I für Schnittstellen. Bei zwei projekteigenen Referenzarchitekturen gab es einige Namenskonventionen, wie: Manager-, UseCase- und Model-Klassen. Die Referenzarchitekturen aus Fallstudie 17 und 22 haben gänzlich auf Namenskonventionen verzichtet, sondern mit Strukturkonventionen gearbeitet.

Bei *Strukturkonventionen* werden alle Klassen einer Elementart in gemeinsamen Packages abgelegt, so dass z. B. in WAM-Architekturen alle Materialien im Package `material` untergebracht sind. Wie sich schon bei der Diskussion im letzten Abschnitt zur Mustertreue auf Subsystem-Ebene gezeigt hat, sind Strukturkonventionen in allen Referenzarchitekturen vorgekommen. Am ausgeprägtesten wurden sie in den Fällen (17, 22) eingesetzt, die keine Namenskonventionen hatten.

Als dritte Möglichkeit wurde in allen Fallstudien *Typisierung* verwendet. Bei Typisierung setzen die Teams Interfaces und Oberklassen ein, um Zugehörigkeiten zu signalisieren. Bei den projekteigenen Referenzarchitekturen und bei der Quasar-Architektur wurden im Projekt selbst Oberklassen geschrieben. Bei den WAM-Architekturen wurde das JWAM-Rahmenwerk verwendet. Durch die Einführung von Oberklassen und Interfaces wollten die Architektinnen und Architekten einerseits sicherstellen, dass Klassen eindeutig einer Elementart zugeordnet werden können und müssen. Auf der anderen Seite wurde in den Oberklassen Standardfunktionalität der Elementarten zur Verfügung gestellt.

Abweichungen von den Regeln der Referenzarchitektur wurden von den Architektinnen und Architekten unterschiedlich beurteilt. Handelte es sich um Verbots- und Elementregeln, so wurden die Abweichungen als Verletzungen betrachtet. Gab es beispielsweise eine verbotene Benutzt-Beziehung oder war die Schnittstelle einer Klasse anders gestaltet, als es ihre Elementart vorschrieb, so musste diese Verletzung aufgelöst werden. Bei Gebotsregeln waren Abweichungen weniger problematisch, denn die Gebotsregeln legen eine typische Zusammenarbeit zwischen Klassen fest, die in manchen Fällen überflüssig sein kann.

Abschließend lässt sich sagen, dass die Mustertreue auf der Klassen-Ebene allen Entwicklungsteams mit Referenzarchitektur ein wichtiges Anliegen war. Die Architektinnen und Architekten betonten, dass es für sie ein architektonisches Problem darstellen würde, wenn das Entwicklungsteam Klassen programmiert, die nicht den gewählten Konventionen entsprechen.

Die Vielzahl der Konventionen lässt darauf schließen, dass den Entwicklungsteams ein einheitliches und eindeutiges Ausdrucksmittel fehlt, mit dem sie die Zugehörigkeit zu einer Elementart signalisieren können.

8.2. Modularität

Der Faktor Modularität überprüft, ob in sich zusammenhängende, abgeschlossene, angemessen große und lose gekoppelte Klassen und Subsysteme

implementiert worden sind. Die dazu entwickelten vier Kriterien Zusammenhalt, Ausgewogenheit, Schnittstellenumfang und Abhängigkeit konnte ich in allen Fallstudien untersuchen.

8.2.1. Zusammenhalt

Der Zusammenhalt von Klassen, Packages und Subsystemen wurde von den Architektinnen und Architekten der Fallstudien unterschiedlich ernst genommen. Manchen war es sehr wichtig, dass die Klassen nach Zuständigkeiten entwickelt werden, andere dachten besonders auf der Ebene der Subsysteme über Zuständigkeiten nach.

Bei den Referenzarchitekturen waren die Architektinnen und Architekten der Meinung, dass die Elementarten den Entwicklerinnen und Entwicklern eine direkte Orientierung geben, welche Zuständigkeit eine Klasse haben sollte. Beispielsweise hat ein Werkzeug in einer WAM-Architektur die Aufgaben, ein Material an der Oberfläche zu präsentieren und seine Bearbeitung zu unterstützen. Diese Ausrichtung wurde von den Architektinnen und Architekten als ein wichtiger Vorteil von Referenzarchitekturen betont. Dementsprechend interessant waren die restlichen Klassen, für die es in den Referenzarchitekturen keine Elementarten gab. Diese Klassen hatten häufig einen geringen Zusammenhalt, was sich daran festmachte, dass sie mehrere Zuständigkeiten gleichzeitig zu erfüllen hatten. Bei fast allen Referenzarchitekturen gab es beispielsweise eine Klasse, die für die Initialisierung beim Systemstart zuständig war. In drei Fällen enthielt diese Klasse zusätzlich einige systemweit benötigte Konstanten und bot Standardfunktionen an, die an mehreren Stellen benötigt wurden. Diese eine Klasse war mit mindestens zwei, wenn nicht mehr, Zuständigkeiten versehen. Bei Architektur-Reviews von Referenzarchitekturen muss daher insbesondere auf Klassen geachtet werden, die keiner Elementart angehören.

Bei Schichten- und Subsystemarchitekturen haben die meisten Architektinnen und Architekten darauf Wert gelegt, dass ihre Entwicklungsteams auf die Zuständigkeiten von Klassen achten sollen. Explizite Regeln zu diesem Punkt wurden in den Projekten allerdings nicht aufgestellt. Besonders aus dem Rahmen fiel Fallstudie 11, deren Architekt auf diese Frage mit Verwunderung reagierte. Die anschließende Diskussion machte deutlich, dass für diesen Architekten Klassen lediglich eine Sammlung von Methoden sind. Besonders augenfällig wurde dies bei der Diskussion darüber, wie Zyklen aufgelöst werden könnten. Der Architekt ging diese Aufgabe so an, dass er Methoden zwischen den Klassen hin und her schob, unabhängig davon, ob sie zu den anderen Methoden der Zielklasse und einer daraus möglicherweise ableitbaren Aufgabe passten oder nicht.

Neben dem Zusammenhalt auf Klassen-Ebene wurde der Zusammenhalt von Subsystemen mit den Architekten der Fallstudien diskutiert. Hier konnte ich zwei Beobachtungen machen. Bei den Subsystemarchitekturen war es den Architektinnen und Architekten ein wichtiges Anliegen, dass ihre Subsysteme jeweils einen Zuständigkeitsbereich haben. Die Subsysteme waren als Dienst

konzipiert, der von außen als eine Einheit wahrgenommen werden sollte. Schließlich fielen auch hier wieder die großen Softwaresysteme ab Fallstudie 17 auf. Bei ihnen waren Subsystemverantwortliche festgelegt, die in textueller Form dokumentiert hatten, welche Aufgabe ihr Subsystem im Gesamtsystem übernehmen sollte. Keine der kleineren Fallstudien hatte hier Vergleichbares zu bieten.

In der Literatur gibt es eine Reihe von Untersuchungen, die den Zusammenhalt von Klassen und zum Teil auch von Subsystemen mit Maßen bestimmen (vgl. Bonja und Kidanmariam 2006; Darcy und Slaughter 2005; Etzkorn et al. 1998; Gui und Scott 2006; Simon 2001). Eine entsprechende Vermessung der Fallstudien war im Rahmen dieser Arbeit nicht zu leisten, ist aber ein wichtiger Ansatzpunkt für weitere Arbeiten mit dem Komplexitätsmodell.

8.2.2. Schnittstellenumfang

Schnittstellen auf Subsystem-Ebene hatten bei einigen Teams einen sehr hohen und bei anderen einen niedrigen Stellenwert. Alle Softwaresysteme oberhalb von 250.000 RLOC waren durchgängig mit Subsystem-Schnittstellen ausgestattet, und den Architektinnen und Architekten war die Pflege der Schnittstellen ein Anliegen (Fallstudie 19 bis 24).

Im Gegensatz dazu waren die kleineren Schichten- und Referenzarchitekturen (1, 4, 6, 11, 13 bis 18) zwar aus Subsystemen aufgebaut. Schnittstellen wurden aber bei der Architekturanalyse nicht erwähnt. Auf Nachfrage zeigte sich, dass den Architektinnen und Architekten Schnittstellen im Prinzip durchaus wichtig waren, sie die Definition von Schnittstellen in ihrem Fall aber nicht für notwendig oder sinnvoll hielten. Die Architektinnen und Architekten sagten meistens sinngemäß: „Die Subsysteme sind noch zu klein, da ist alles in der Schnittstelle “.

Ein weiterer Grund sprach in Fallstudien (13, 14 und 16) gegen Schnittstellen. Diese drei Fallstudien stammen aus derselben Organisation, bei der ein selbst entwickeltes Komponentenmodell eingesetzt wird. Die Definition von Subsystemen und ihren Schnittstellen müssen in xml-Dateien abgelegt werden, und ein Zugriff an der Schnittstelle vorbei wurde in der Entwicklungsumgebung nicht zugelassen. Die Architekten dieser Fallstudien machten grundsätzlich alle Packages eines Subsystems zu seiner Schnittstelle. Ihre Begründung für dieses Vorgehen war, dass sie nicht sicher sein können, was andere Entwicklerinnen und Entwickler aus ihrem Subsystem benötigen würden, daher stellen sie lieber mehr als weniger zur Verfügung.

Ab einer Größe von knapp 50.000 RLOC (Fallstudie 10) waren bei den Referenz- und Schichtenarchitekturen erste Ansätze von Schnittstellen zu erkennen. I. d. R. bildete sich als erstes die Schnittstelle zwischen Client und Server heraus. Später kamen weitere Schnittstellen für Subsysteme und Schichten hinzu.

Für die Architektinnen und Architekten der Subsystemarchitekturen (Fallstudie 8, 20 und 21) waren Schnittstellen im Gegensatz dazu das entscheidende

Mittel, um die Architektur ihres Softwaresystems stabil zu halten. Diese Architektinnen und Architekten waren ganz besonders daran interessiert, mit dem Sotographen zu überprüfen, ob die Schnittstellen ihrer Subsysteme eingehalten werden. Alle drei Teams berichteten, dass sie von Beginn ihrer Entwicklung an über die Schnittstellen der Subsysteme diskutiert und sie festgelegt haben. Das Team aus Fallstudie 20 hatte sehr früh damit begonnen, alle Schnittstellen im firmen-internen Intranet abzulegen. Diese Dokumentation bildet das Kernstück der Architekturbeschreibung und wird an alle Veränderungen angepasst. Subsysteme dürfen nur über die dort dokumentierten Klassen und Operationen benutzt werden. Nach Meinung seiner Architektin war das Softwaresystem nur deshalb erweiterbar und wartbar geblieben, weil das Entwicklungsteam sich dieser Restriktion unterworfen hat.

Nicht nur über Dokumentation versuchen die Teams in den Fallstudien deutlich zu machen, aus welchen Klassen und Packages die Schnittstelle ihrer Subsysteme besteht. In den Fallstudien mit Schnittstellen haben die Entwicklungsteams den Package-Baum genutzt, um Schnittstellen von den Subsystem-internen Packages abzugrenzen. Drei Mittel wurden dabei einzeln oder in Kombination verwendet:

- Der oberste Package-Knoten eines Subsystems enthält die Klassen, die die Schnittstelle bilden. Alle tiefer liegenden Packages sind nicht Teil der Schnittstelle und dürfen von außen nicht verwendet werden (Fallstudie 11, 14, 16, 20, 21, 22).
- Direkt unter dem obersten Package-Knoten eines Subsystems existiert ein Package namens `api`, das die Schnittstelle enthält (Fallstudie 13, 24).
- Im Package-Baum eines Subsystems gibt es ein oder mehrere Package-Teilbäume, die `intern` oder `impl` heißen. Alle anderen Packages bilden die Schnittstelle (Fallstudie 13, 14, 16, 21, 22, 23, 24).

Die Art, wie die Entwicklungsteams die Schnittstellen im Package-Baum abbildeten, war in den einzelnen Fallstudien selbst einheitlich. In diesem Punkt haben die Teams ihren Package-Baum also mustertreu im Sinne des Komplexitätsmodells aufgebaut (s. Abschnitt 6.3).

Insgesamt bleibt festzuhalten, dass Schnittstellen bei Schichten- und Referenzarchitekturen erst mit zunehmender Größe zum Einsatz kommen. Subsystemarchitekturen hingegen bauen ihre Architektur von Anfang an mit Schnittstellen auf. Alle Teams versuchen, die Schnittstellen im Package-Baum über Konventionen sichtbar zu machen.

8.2.3. Ausgewogenheit

Das Kriterium Ausgewogenheit befasst sich mit der Frage, ob Klassen und Subsysteme ähnlich groß sind. In Abbildung 8-6 ist die durchschnittliche Anzahl von Programmzeilen pro Klasse im Verhältnis zur Gesamtanzahl der Programmzeilen (RLOC) abgebildet. Abbildung 8-7 stellt zusätzlich die

Anzahl der Klassen im Verhältnis zur Größe der Fallstudien dar. Die beiden Abbildungen zeigen, dass größere Softwaresysteme zwar mehr Klassen haben, die Klassen im Durchschnitt aber nicht größer werden. Die typische Größe für Klassen lag bei den Fallstudien unterhalb von 100 ausführbaren Programmzeilen. Die Messwerte zu den einzelnen Fallstudien lassen sich aus Tabelle C-1 im Anhang entnehmen.

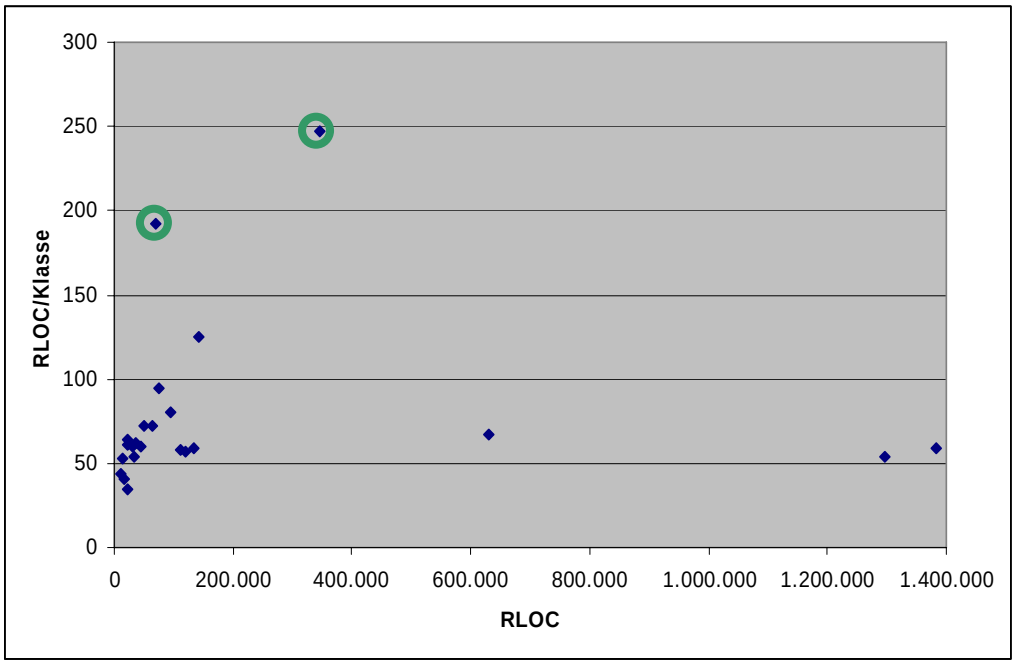

Abbildung 8-6: Anzahl von Programmzeilen pro Klasse

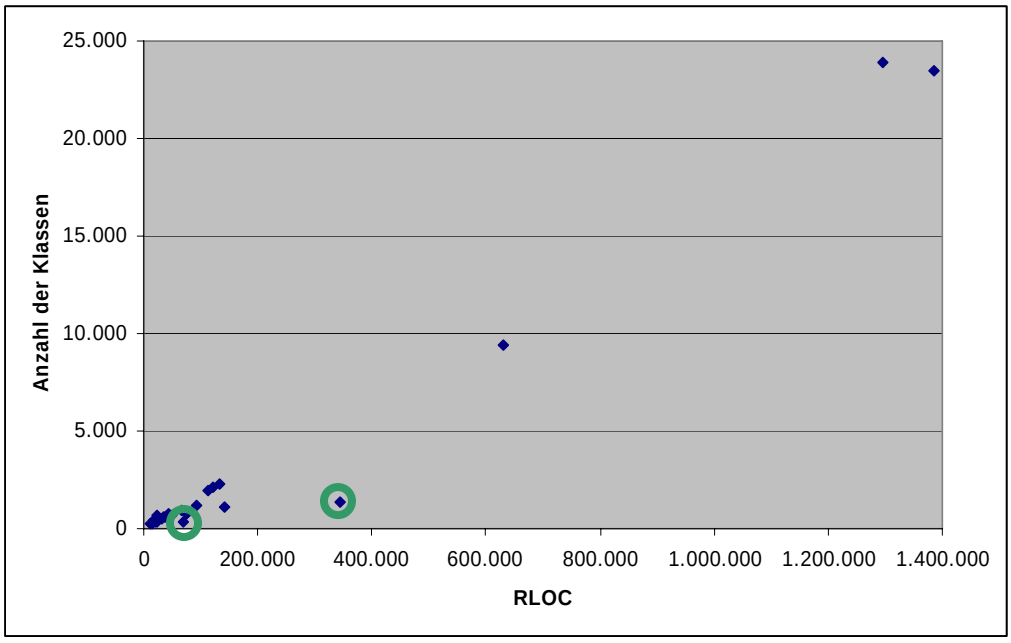

Abbildung 8-7: Klassenanzahl nach Größe der Softwaresysteme

Zwei Fallstudien (12 und 20) besaßen im Vergleich zu den anderen größere Klassen und damit auch verhältnismäßig weniger Klassen insgesamt (grüner Rand). Bei Fallstudie 12 wurden die gesamte Logik der Datenbankanbindung und die Steuerung der Oberfläche in einigen sehr großen Klassen implementiert. Bei Fallstudie 20 waren die Entwicklerinnen und Entwickler der Meinung, dass große Klassen kein Problem darstellen und sie mehr Probleme damit hätten, wenn sie viele kleinere Klassen hätten. Von diesen zwei Ausnahmen abgesehen, lässt sich aus Abbildung 8-7 und Abbildung 8-6 folgern, dass die Entwicklerinnen und Entwickler der Fallstudien eine ähnliche Vorstellung davon haben, wie groß Klassen sein sollen, so dass bei größeren Softwaresystemen nicht die Größe, sondern die Anzahl der Klassen steigt. Im Anhang findet sich der Vollständigkeit halber eine Auswertung der größten Klassen aus den Fallstudien (s. Abbildung A-1).

Für Subsysteme zeichnen die Fallstudien ein anderes Bild als für die Klassen. Abbildung 8-8 macht deutlich, dass die Entwicklungsteams ihre Subsysteme unterschiedlich groß anlegen. Die beiden großen Fallstudien sind hierfür das Paradebeispiel. Fallstudie 23 hat dreiundzwanzig Subsysteme, und Fallstudie 22 besteht aus 210 Subsystemen. Ein entsprechendes Diagramm mit der Anzahl Programmzeilen pro Subsystem findet sich im Anhang (s. Abbildung A-2).

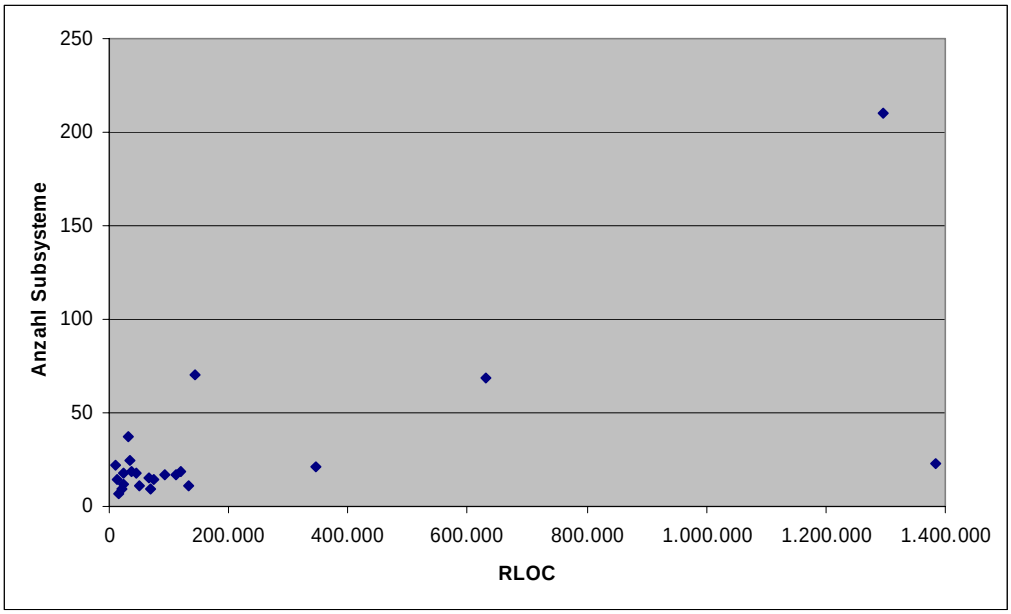

Abbildung 8-8: Anzahl der Subsysteme zu Größe der Softwaresysteme

Die uneinheitliche Größe von Subsystemen lässt sich daraus erklären, dass ich in den Untersuchungen keine Vorgaben gemacht habe, was unter einem Subsystem zu verstehen sei. Vielmehr habe ich den Entwicklungsteams die Möglichkeit gegeben, ihr Verständnis von Subsystemen abzubilden. In Abbildung 8-8 werden deshalb zwei unterschiedliche Vorstellungen von Subsystemen sichtbar. Die einen Teams betrachten die Schnitte einer erweiterten Schichtenarchitektur als Subsysteme (s. Abbildung 8-9 Subsystem A, B, C).

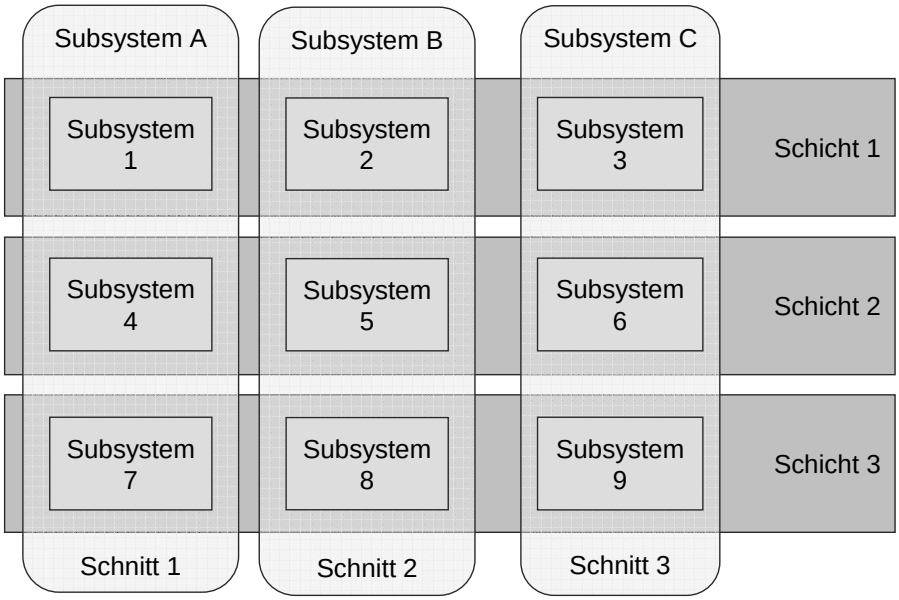

Abbildung 8-9: Bildung von Subsystemen

Die anderen Teams gingen von einer Matrix-Aufteilung ihres Softwaresystems in Schnitte und Schichten aus und bildeten die Subsysteme in den Zellen der Matrix (s. Abbildung 8-9 Subsystem 1 bis 9). Die einen Teams verstanden unter Subsystemen eigenständige Einheiten (Subsystem A-C), die sowohl eine Oberfläche als auch Fachlogik und Datenhaltung beinhalten (Fallstudie 6, 12, 15, 17, 18, 20, 23). Andere Teams schnitten ihre Subsysteme kleiner, so dass sie neun statt drei Subsysteme erhielten (Fallstudie 1, 2, 3, 5, 7, 9, 10, 13, 14, 16, 22).

Die in Abschnitt 8.1.1 zur Mustertreue auf der Subsystem-Ebene beschriebenen Varianten für den Aufbau des Package-Baums machen diese unterschiedlichen Präferenzen bei der Größe von Subsystem ebenfalls deutlich. Werden Subsysteme nach Schnitten gebildet, so hat der Package-Baum fachliche Einheiten als oberste Knoten der Subsysteme. Sind die obersten Knoten Schichten, so kann man davon ausgehen, dass die Subsysteme nach der Matrix-Sicht gebildet wurden.

Für das Kriterium Ausgewogenheit lässt sich festhalten, dass die durchschnittliche Größe der Klassen in den meisten Fallstudien ähnlich war, so dass größere Softwaresysteme nicht zu größeren Klassen führen. Auf der Subsystem-Ebene war zu beobachten, dass die Größe von Subsystemen uneinheitlich gesehen wird.

8.2.4. Abhängigkeit

Das Kriterium Abhängigkeit befasst sich mit der Frage, wie viele Beziehungen Klassen und Subsysteme im Durchschnitt haben. Aus Abbildung 8-10 lässt sich entnehmen, dass, bis auf zwei, alle Fallstudien im Durchschnitt unter elf Beziehungen pro Klasse haben. Die Messwerte zu den einzelnen Fallstudien lassen sich aus Tabelle C-1 im Anhang entnehmen.

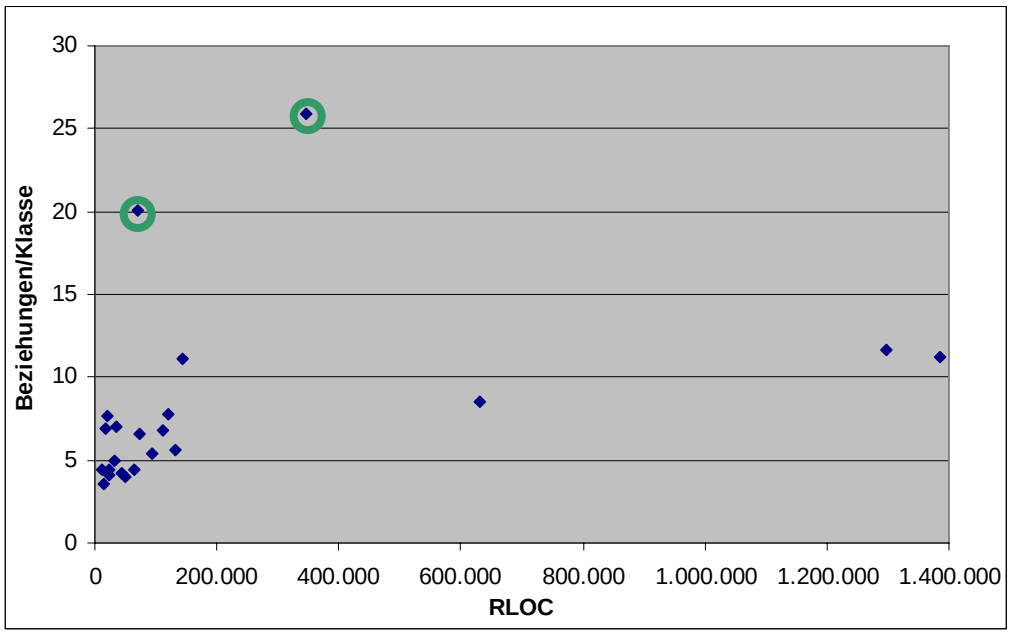

Abbildung 8-10: Anzahl der Beziehungen pro Klasse

Die beiden Softwaresysteme, die oberhalb dieser Grenzen liegen, sind Fallstudie 12 (20 Beziehungen) und 20 (26 Beziehungen). Beide Fallstudien sind bereits beim Kriterium Ausgewogenheit aufgefallen (s. Abschnitt 8.2.3), weil ihre Klassen besonders groß sind. In Fallstudie 12 steuerten die sehr großen Klassen die Abläufe in weiten Teilen des Softwaresystems und stellten Standardfunktionalität zur Verfügung, so dass sie viele Klassen kennen und von vielen gekannt werden mussten. Ihre Namen deuteten auf eine solche umfassende Zuständigkeit hin: `{*}Manager`. Die Refactoring-Bewegung bezeichnet diese Klassen als „God Class“ (s. auch Abschnitt 6.4.1) und geht davon aus, dass ein solcher Entwurf nicht objektorientiert ist (Riel 1996, S. 32f). In Fallstudie 20 sind ebenfalls große steuernde Klassen mit vielen Beziehungen enthalten. Zusätzlich wurden die meisten fachlichen Klassen, die die grundlegenden Produkte des Gegenstandsbereichs abbildeten, von 100 bis über 500 anderen Klassen benutzt.

Auf die Beziehungen zwischen Subsystemen gehe ich hier nicht weiter ein, weil die Größe der Subsysteme in den Fallstudien zu uneinheitlich war (s. dazu Abschnitt 8.2.3 zur Ausgewogenheit). Im Anhang findet sich der Vollständigkeit halber eine Auswertung der Beziehungen auf Subsystem-Ebene (s. Abbildung A-3).

8.3. Geordnetheit

Die Kriterien Zyklenausmaß, -reichweite, -umfang und Verflochtenheit untersuchen, wie stark die Geordnetheit in einem Softwaresystem durch Zyklen auf den verschiedenen Ebenen verletzt wird. In Kapitel 7 wurden dreizehn Maße definiert, so dass für diesen Faktor eine Reihe von Messwerten vorliegen (s. Tabelle C-2 bis Tabelle C-6 im Anhang).

Die Ergebnisse werden in diesem Abschnitt nicht entlang der Kriterien vorgestellt, sondern zuerst auf Klassen-Ebene, dann auf Package-Ebene und abschließend auf Subsystem-Ebene. Diese Reihenfolge habe ich gewählt, weil ich so die Auswirkungen der Architekturstile auf eine Ebene kompakt diskutieren kann.

8.3.1. Geordnetheit auf Klassen-Ebene

Unter den zweiundzwanzig vermessenen Fallstudien waren lediglich zwei, die keine Zyklen auf der Klassen-Ebene hatten. In den anderen Fallstudien habe ich zwischen 0,4 und 76% Klassen in Zyklen gemessen (Maß `Classes-InCycles`, s. Tabelle C-2 im Anhang).

Bei Auswertungen von Messwerten wird in der Literatur darauf hingewiesen (vgl. El-Emam et al. 2001), dass die Größe des jeweiligen Softwaresystems einen entscheidenden Einfluss auf seine Messwerte haben kann. Um diesen Zusammenhang auszuschließen, habe ich in Abbildung 8-11 die Anzahl der Klassen in Zyklen zu der Gesamtanzahl der Klassen in Beziehung gesetzt.

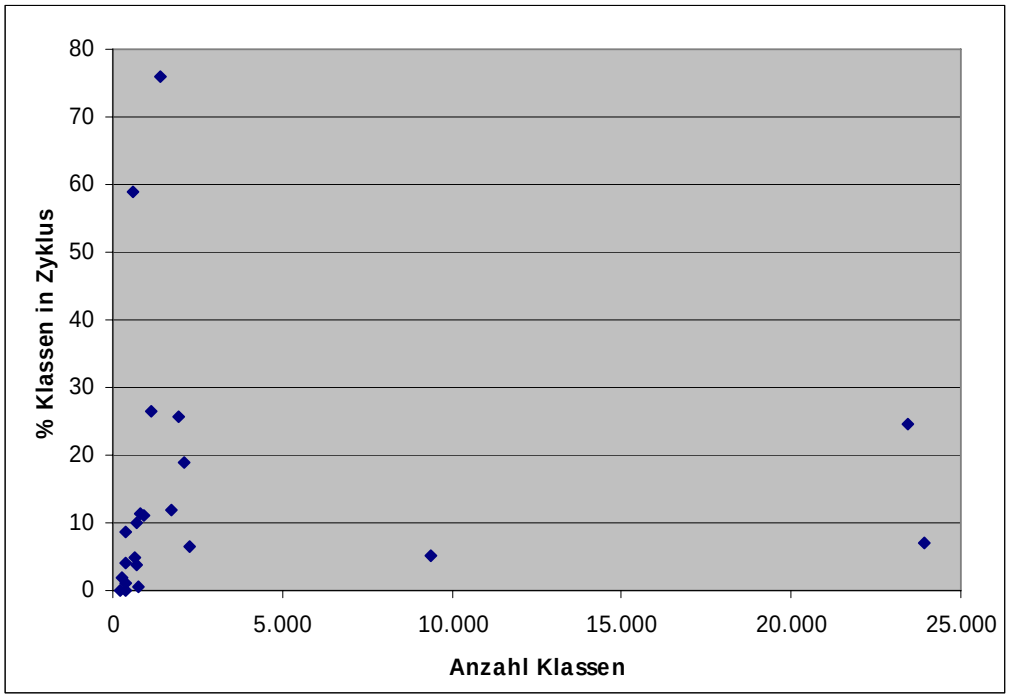

Abbildung 8-11: Zyklen und Klassen

Die Anzahl der Klassen hat demnach keinen Einfluss auf den Zyklenumfang. Trägt man das Ausmaß der Zyklen gegen die relevanten Programmzeilen (RLOC) auf, so ergibt sich ein vergleichbares Bild (s. Abbildung A-4 im Anhang). Eine Erklärung für die gemessenen Ergebnisse zum Faktor Geordnetheit lässt sich stattdessen finden, wenn man die eingesetzten Architekturstile betrachtet.

Architekturstile und Zyklen auf Klassen-Ebene

In Abbildung 8-12 und Abbildung 8-13 wird das Zyklenausmaß auf die Architekturstile abgebildet (Maße `ClassesInCycles` und `RefClasses-InCycles`).

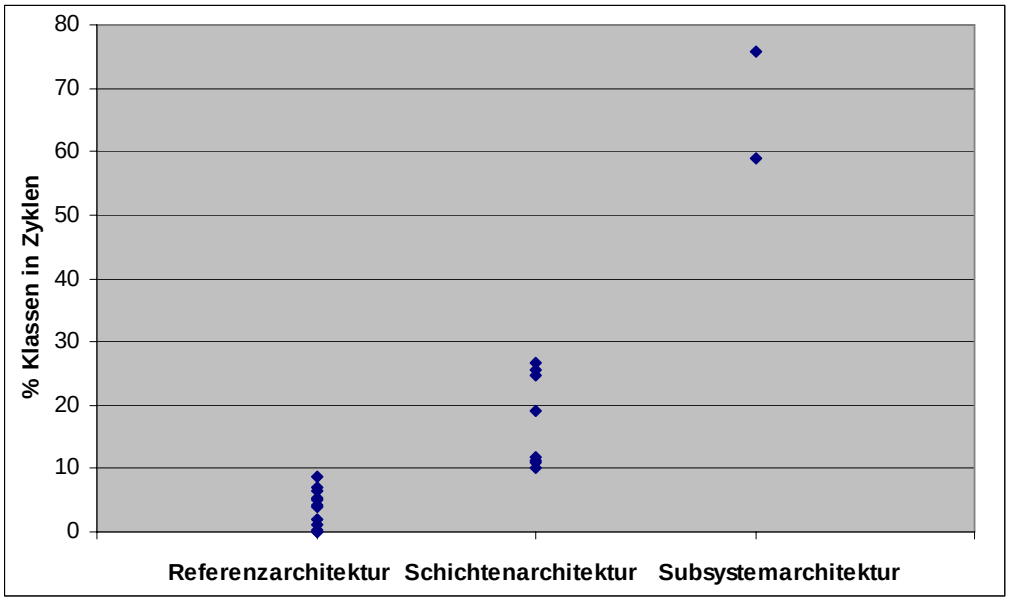

Abbildung 8-12: Architekturstile mit Klassen in Zyklen

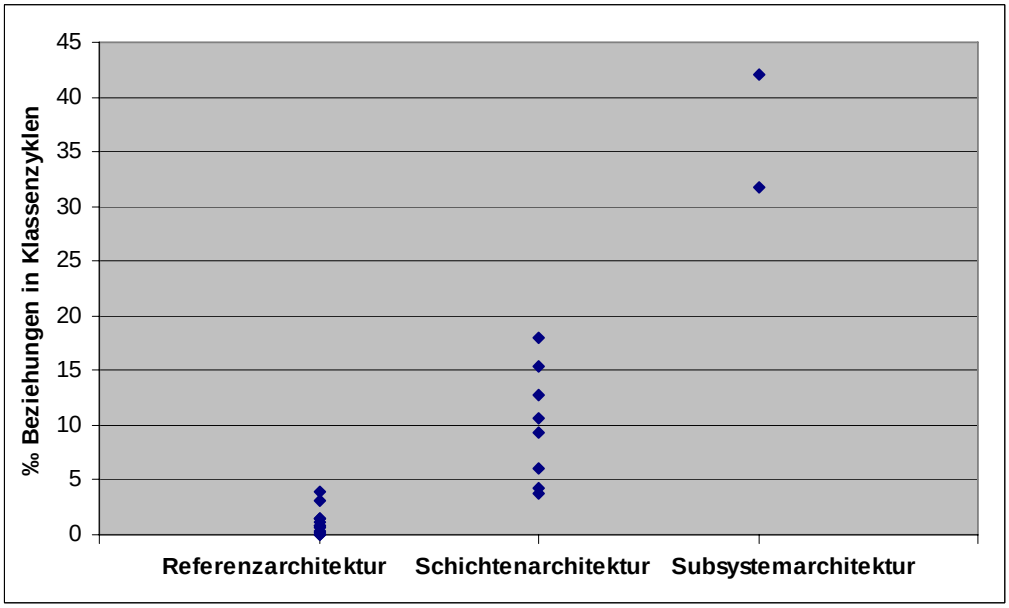

Abbildung 8-13 Architekturstil mit Beziehungen in Zyklen

Alle Fallstudien, die eine Kombination aus einer Referenzarchitektur und einem anderen Architekturstil haben, werden in diesem Abschnitt als Referenzarchitekturen betrachtet, weil dieser Architekturstil auf der Klassen-Ebene angesiedelt ist. Die beiden Abbildungen zeigen, dass Referenzarchitek-

tur in der Regel ein geringeres Zyklenausmaß haben als Schichten- oder Subsystemarchitektur.

Bei den Referenzarchitekturen nimmt Fallstudie 5 mit 8,7% und 3,9% bei beiden Maßen den letzten Platz ein. Dieser hohen Werte lassen sich aus ihrer Entwicklungsgeschichte erklären. Die Architekturanalyse zu diesem Softwaresystem habe ich mit dem zuständigen Wartungsentwickler durchgeführt. Er erzählte, dass das Softwaresystem mit einer WAM-Architektur entwickelt worden war. Anschließend wurde es von einem Wartungsteam übernommen, das keine Erfahrung mit der verwendeten Referenzarchitektur hatte. Als zusätzliche Schwierigkeit kam hinzu, dass die ursprüngliche Entwicklung von externen Mitarbeitern geleistet worden war, die nach der Übergabe an das Wartungsteam nicht mehr zur Verfügung standen. Nach zwei Jahren Wartung durch wechselnde Gruppen von Wartungsentwicklern war das Softwaresystem für den jetzt zuständigen Wartungsentwickler unverständlich und schwer beherrschbar. Um Abhilfe zu schaffen, wurde dem Wartungsentwickler einer der ursprünglichen Entwickler zur Seite gestellt, der feststellen musste, dass bei den Erweiterungen der letzten zwei Jahre verschiedene Vorgaben der Referenzarchitektur missachtet worden waren und so die Struktur des Softwaresystems degeneriert war.

Die höchsten Werte bei den Schichtenarchitekturen (18,0% und 15,3%) stammen von Fallstudie 14 und 15, die beide von ihren Architekten als problematisch in der Wartung eingestuft wurden.

Die meisten Zyklen auf Klassen-Ebene lassen sich in den beiden Softwaresystemen finden, die als reine Subsystemarchitektur entwickelt wurden. Dieses Ergebnis lässt sich mit Hilfe von Aussagen aus den Interviews und Architekturdiskussionen erklären. Die Architekten der Subsystemarchitektur haben nicht darauf geachtet, Zyklen zu vermeiden. Im Interview zu Fallstudie 20 sagte eine Architektin explizit: „Zyklen sind für uns kein Kriterium gewesen!“

Nicht nur das Ausmaß an Zyklen ist abhängig von der Art des Architekturstils, sondern auch die maximale Größe und die Verflochtenheit der Zyklen werden durch den Architekturstil beeinflusst. Abbildung 8-14 stellt diesen Zusammenhang zwischen dem Zyklenumfang und Referenzarchitekturen und Schichtenarchitekturen dar (Maß Maximaler Wert von `ClassesInCycles`). Die beiden Werte für die Subsystemarchitekturen (45,8 und 75,2%) habe ich weggelassen, um den Unterschied zwischen Referenz- und Schichtenarchitekturen deutlich sichtbar zu machen.

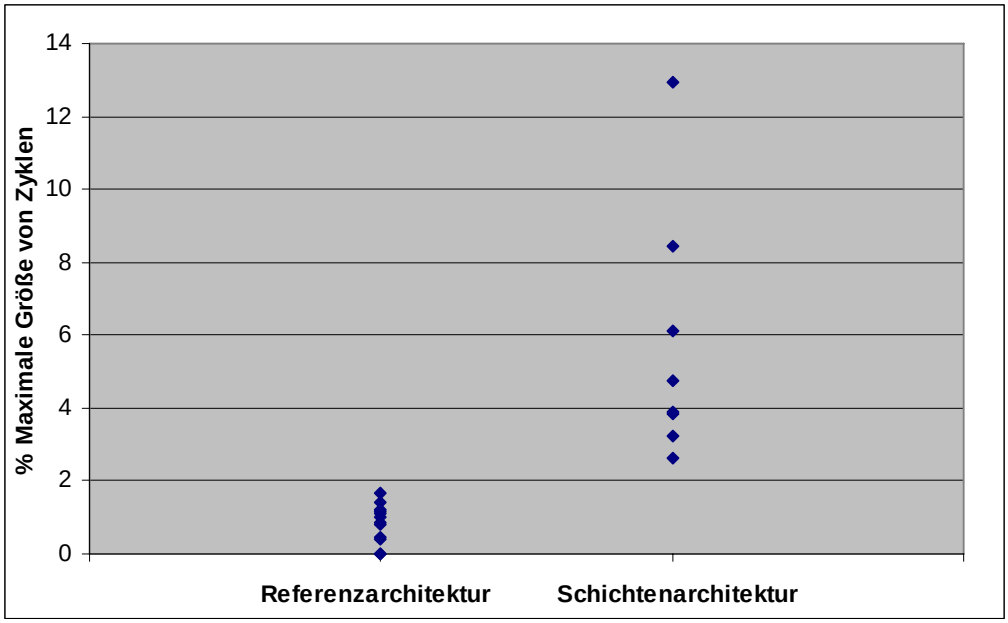

Abbildung 8-14: Architekturstile und Zyklenumfang

In Abbildung 8-14 wird sichtbar, dass die relative Größe der Zyklen bei Referenzarchitekturen deutlich geringer ist als bei Schichtenarchitekturen. Für Referenzarchitekturen liegen die zwölf Werte für die maximale Größe zwischen 0 und 1,7%. Bei den Schichtenarchitekturen reichen die Werte von 2,6 bis 13%.

Abbildung 8-15 stellt den Zusammenhang zwischen Architekturstil und der Verflochtenheit der Zyklen dar. Hier wird sichtbar, dass Referenzarchitekturen Zyklen mit weniger bidirektionalen Beziehungen enthalten.

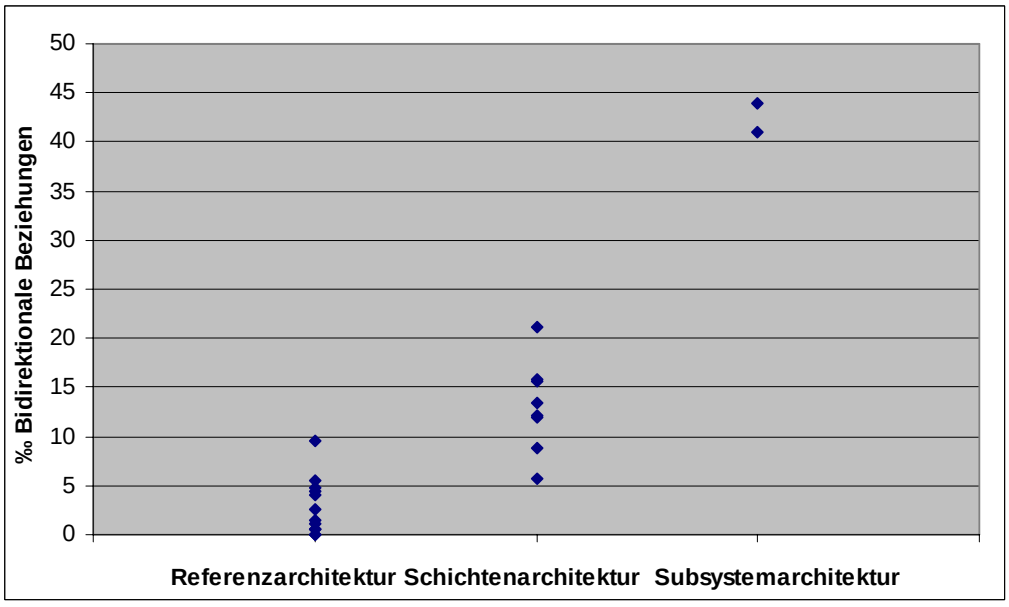

Abbildung 8-15: Architekturstil und Verflochtenheit

Der höchste Wert (9,6%) bei den Referenzarchitekturen stammt wiederum aus Fallstudie 5, die bereits beim Zyklenausmaß aufgefallen ist.

Neben dem bis hierher diskutierten Zusammenhang zwischen der Geordnetheit auf der Klassen-Ebene und dem Architekturstil konnte noch eine weitere Beobachtung gemacht werden, die eine Erkenntnis zum Kriterium Zusammenhalt wieder aufnimmt (s. Abschnitt 8.2.1). Bei Referenzarchitekturen wurden Zyklen in der Regel durch die Klassen verursacht, die keiner Elementart der Referenzarchitektur zugeordnet werden konnten. Typische Vertreter sind Klassen, die Teile des Softwaresystems starten, Klassen für die Ausnahmebehandlung und Klassen, die Daten zur Benutzeranmeldung und/oder den Datenbankzugang verwalten. Bei solchen Klassen fehlt den Entwicklerinnen und Entwicklern eine Anleitung durch Regeln der Referenzarchitektur, die festlegt, welche Rolle die Klasse im Softwaresystem spielen soll. Ohne eine solche Anleitung kommt es in Referenzarchitekturen vermehrt zu Zyklen.

Programmzeilen in Zyklen

Neben den im Komplexitätsmodell untersuchten Maßen zur Geordnetheit auf Klassen-Ebene konnten Messwerte zu einem weiteren Effekt erhoben werden, der im Zusammenhang mit Zyklen in früheren Arbeiten untersucht wurde (Neumann 2005, S. 94f).

Am Beispiel des Projekts `Eclipse`[24] konnte Neumann statistisch belegen, dass Klassen, die in Zyklen sind, potentiell größer sind als Klassen, die nicht zu Zyklen gehören (Neumann 2005, Hypothese 7, S. 94f). Diese nur für das Eclipse-Projekt belegte Hypothese konnte auf Basis der Fallstudien bestätigt werden. Bei den hier untersuchten Softwaresystemen war der Anteil der relevanten Programmzeilen aller Klassen in Zyklen um den Faktor 1,5 bis 3 größer als der Anteil der Klassen in Zyklen selbst (s. Tabelle 8-1). Die Daten in Tabelle 8-1 sind nach dem Zyklenausmaß sortiert, und der jeweilige Architekturstil ist anhand des Grautons erkennbar (Referenzarchitekturen – weiß, Schichtenarchitekturen – hellgrau, Subsystemarchitekturen – dunkelgrau).

Tabelle 8-1: Programmzeilen in Klassenzyklen

Nr	RLOC	% Klassen in Zyklen	% RLOC in Zyklen
1	10.679	0,0	0,0
4	21.591	0,0	0,0
9	43.969	0,4	2,0
3	16.314	1,0	3,5
2	13.219	2,0	2,8
10	49.428	3,8	9,9
12	70.266	4,1	5,2
7	33.651	5,0	20,7
21	629.674	5,2	9,6

[24] `www.eclipse.org`

17	133.095	6,5	17,2
22	1.296.455	7,1	15,3
5	22.957	8,7	16,9
6	23.478	10,1	23,4
11	65.284	11,0	23,8
13	84.480	11,3	26,1
18	143.101	11,9	17,2
16	120.964	19,0	40,7
23	1.384.260	24,6	59,2
15	112.675	25,6	35,9
14	93.593	26,6	40,3
8	36.671	59,0	66,4
20	344.911	75,9	90,4

In Tabelle 8-1 stellt der Wert für Fallstudie 7 von 20,72% RLOC in Zyklen eine auffällige Abweichung dar. Hier wird ein Effekt sichtbar, der für alle in dieser Arbeit untersuchten Softwaresysteme als Tendenz zu beobachten war: Klassen, die für den Zugriff auf die Datenbank- oder den Import und Export externer Dateien zuständig sind, sind im Verhältnis zwei- bis dreimal so groß wie andere Klassen. In Klassen zum Speichern, Importieren oder Exportieren wird häufig der komplette Ablauf prozedural implementiert, oder es wird eine Hauptklasse entwickelt, die den gesamten Vorgang steuert und Grundfunktionalität zur Verfügung stellt. So entstehen große Klassen, die, wenn sie in Zyklen eingebunden sind, zu extremen Werten führen. Bei Fallstudie 7 sind neun der 31 an Zyklen beteiligten Klassen für die Speicherung von Objekten in die Datenbank oder den Import von externen Dateien zuständig.

Für die Geordnetheit auf Klassen-Ebene lässt sich festhalten, dass der jeweils gewählte Architekturstil in den Fallstudien starken Einfluss hatte. Referenzarchitekturen verhindern durch ihre Regeln Klassenzyklen, sind aber keine Garantie dafür, dass Klassen, für die es keine Regeln gibt, ohne Zyklen gebaut werden. Schichten- und Subsystemarchitekturen haben keinen vergleichbaren Effekt auf der Klassen-Ebene.

8.3.2. Geordnetheit auf Package-Ebene

Auf der Package-Ebene wurden in allen vermessenen Softwaresystemen Zyklen gefunden. Die tabellarischen Ergebnisse der Vermessung sind im Anhang in Tabelle C-3 und Tabelle C-4 zu finden. Genau wie bei den Zyklen auf Klassen-Ebene hat auf der Package-Ebene die Größe der Softwaresysteme keinen Einfluss auf Zyklenausmaß, –reichweite, –umfang und auf Verflochtenheit (s. Abbildung A-5 im Anhang).

Untersucht man den Zusammenhang zwischen Zyklenausmaß und Architekturstil, so ergibt sich die in Abbildung 8-16 dargestellte Verteilung (Maß `PckgInCycles`). In Abbildung 8-16 habe ich die Softwaresysteme, die sowohl eine Referenzarchitektur als auch eine Schichtenarchitektur haben, separat

aufgeführt, um festzustellen, ob diese Kombination auf der Package-Ebene Auswirkungen hat. Für die Beziehungen in Package-Zyklen ergibt sich ein ähnliches Bild (s. Abbildung A-6 im Anhang zu Maß `RefPckgInCycles`).

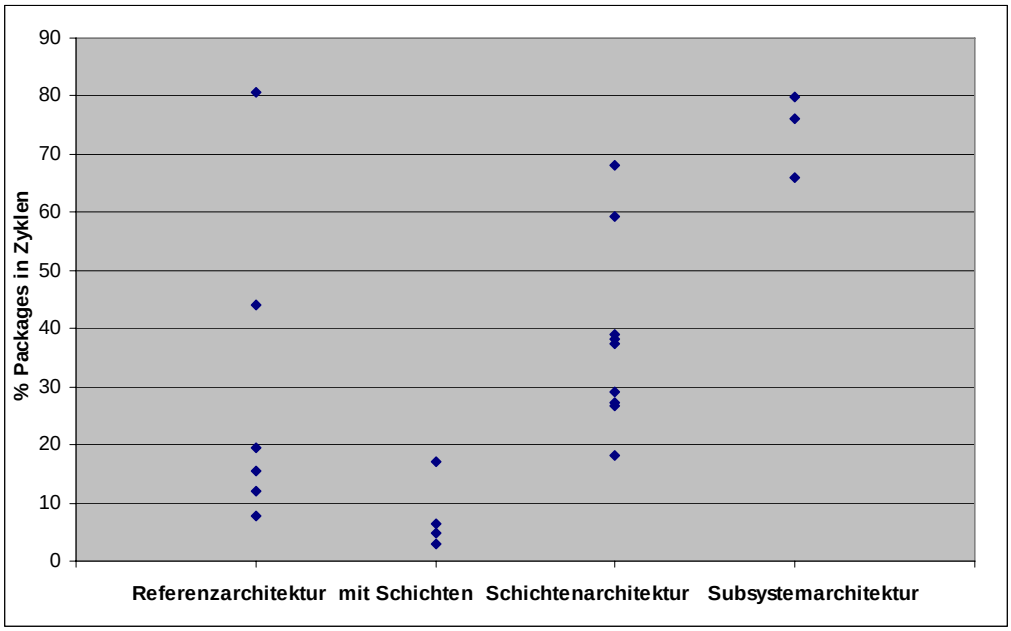

Abbildung 8-16: Architekturstil mit Ausmaß an Package-Zyklen

Das Ergebnis in Abbildung 8-16 lässt keinen allgemeinen Zusammenhang zwischen Architekturstil und dem Ausmaß an Package-Zyklen erkennen. Wie auf der Klassen-Ebene haben die untersuchten Subsystemarchitekturen ein hohes Ausmaß an Package-Zyklen. Da bei Subsystemarchitekten die Vermeidung von Zyklen auf keiner Ebene ein Ziel ist, war dieses Ergebnis zu erwarten. Die vier Fallstudien (1, 4, 17, 21), bei denen eine Referenzarchitektur mit einer Schichtenarchitektur kombiniert ist, haben die wenigsten Package-Zyklen. Diese Werte geben die Haltung eines Architekten wieder, der in der Architekturdiskussion sagte: „Wenn wir schon auf Zyklen achten, dann überall!“.

Im Komplexitätsmodell habe ich vorgeschlagen, dass neben dem Zyklenausmaß auf Package-Ebene auch die Zyklenreichweite untersucht werden sollte, um festzustellen, welchen Anteil die Klassenzyklen an den Package-Zyklen haben (Maße `PackageCycleGroupsFromClassCycles` und `RefsInClassCyclesBetweenPackages`). Die Messergebnisse zu diesen beiden Maßen lassen sich aus Tabelle C-3 im Anhang entnehmen. Abbildung 8-17 stellt den Zusammenhang zwischen dem Architekturstil und der Anzahl der Referenzen in Package-Zyklen dar, deren Ursache Referenzen in Klassenzyklen sind.

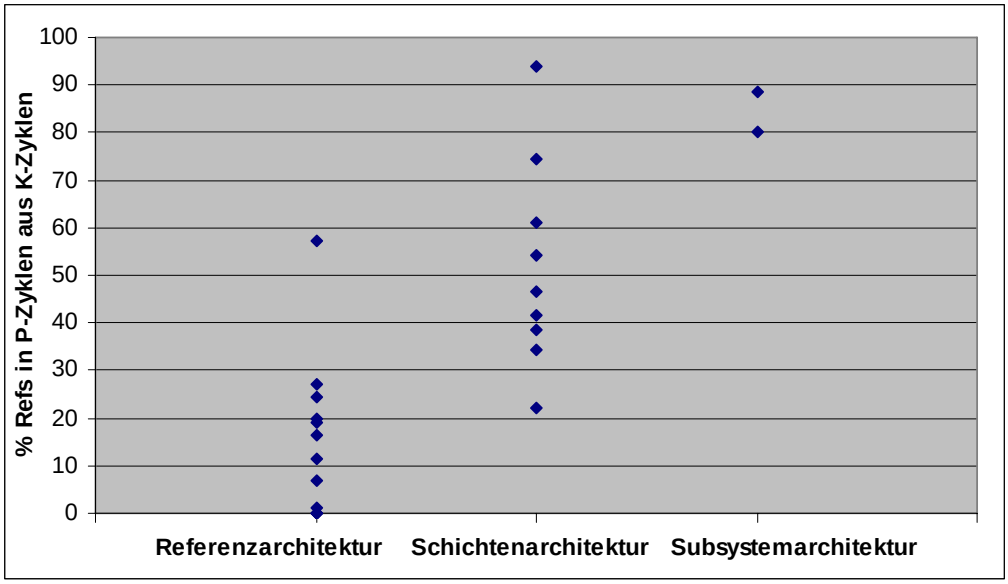

Abbildung 8-17: Zyklenreichweite auf Package-Ebene

Der höchste Wert (57,1%) bei den Referenzarchitekturen wird, genau wie schon bei der Geordnetheit auf Klassen-Ebene, durch Fallstudie 5 verursacht, die sich in einem schlechten Zustand befindet. Die drei höchsten Werte bei den Schichtenarchitekturen (94,1%, 74,3% und 61,2%) stammen von den Fallstudien 11, 14 und 15; sie wurden alle von ihren Architekten als problematisch eingestuft.

Insgesamt macht Abbildung 8-17 deutlich, dass die untersuchten Referenzarchitekturen im Verhältnis weniger Package-Zyklen besitzen, denen Klassenzyklen zugrunde liegen. Dieser Effekt ist die logische Folge der Ergebnisse aus dem letzten Abschnitt, nach denen Referenzarchitekturen weniger Klassenzyklen verursachen als andere Architekturstile (s. Abschnitt 8.3.1).

Package-Zyklen haben in Referenzarchitekturen ihre Ursache häufig darin, dass Klassen einem falschen Package zugeordnet worden waren. Wie schon in Abschnitt 8.3.1 im Zusammenhang mit Klassenzyklen diskutiert wurde, sorgen bei Referenzarchitekturen besonders solche Klassen für Probleme, die keiner der vorgegebenen Elementarten zugeordnet werden konnten. Bei diesen Klassen ist dem Entwicklungsteam nicht klar, wo sie einsortiert werden sollen, und es wird eine adhoc-Entscheidung getroffen. In der Regel sorgte eine Restrukturierung dieser Klassen dafür, dass die Package-Zyklen aufgelöst wurden.

Für die Kriterien Zyklenumfang und Verflochtenheit konnten aus Tabelle C-4 keine weiteren Erkenntnisse gewonnen werden (Maße `MaxPckgInCycle` und `DirectRefsInPackageCycles`).

Insgesamt lässt sich für Geordnetheit auf Package-Ebene festhalten, dass man zu den besten Ergebnissen kommt, wenn Referenzarchitekturen mit Schichtenarchitekturen kombiniert werden.

8.3.3. Geordnetheit auf Subsystem-Ebene

Auf der Subsystem-Ebene wurden in neun der zweiundzwanzig vermessenen Fallstudien keine Zyklen gefunden[25]. Alle neun Fallstudien ohne Subsystemzyklen haben eine Schichtenarchitektur. Die gemessenen Werte finden sich im Anhang in Tabelle C-5 und Tabelle C-6. Genau wie bei den Zyklen auf Klassen- und Package-Ebene hat auch auf der Subsystem-Ebene die Größe der Softwaresysteme keinen sichtbaren Einfluss auf die Geordnetheit (s. Abbildung A-7 im Anhang).

Architekturstil und Zyklen auf Subsystem-Ebene

Abbildung 8-18 stellt den Anteil der Subsysteme in Zyklen im Verhältnis zum Architekturstil dar (Maß `SubsysInCycles`). Für diese Darstellung wurde ein gewichtetes Streudiagramm gewählt, um sichtbar zu machen, dass fast alle der zwölf untersuchten Schichtenarchitekturen keine Zyklen zwischen Subsystemen haben. Die Auswertung für die Beziehungen in Subsystemzyklen findet sich im Anhang (s. Abbildung A-8 zu Maß `RefSubsysInCycles`).

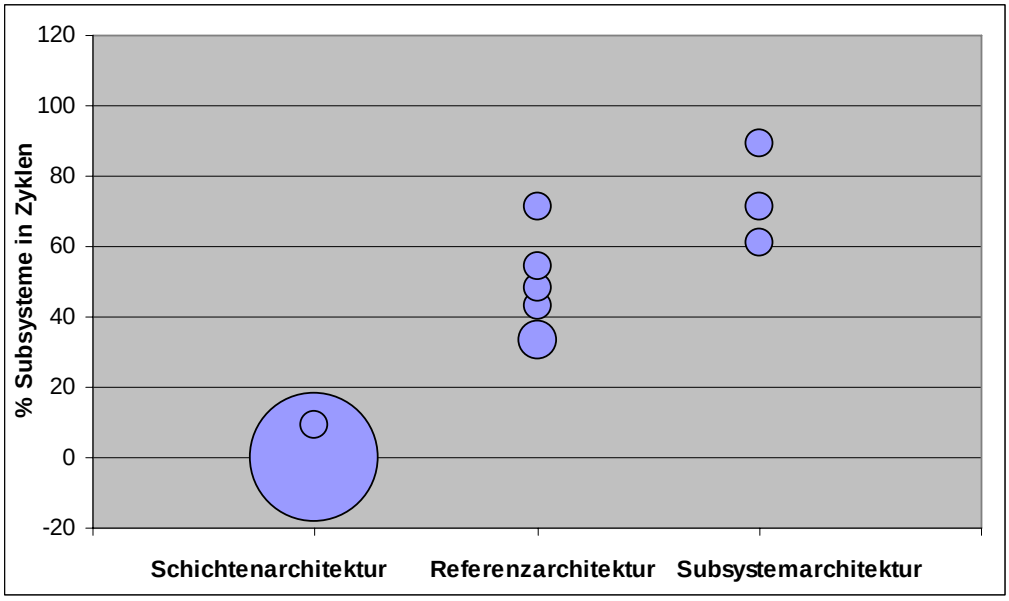

Abbildung 8-18: Architekturstile mit Ausmaß an Subsystemzyklen

In Abbildung 8-18 sieht man, dass lediglich eine Schichtenarchitektur Zyklen auf Subsystem-Ebene hat. Dieser Wert (9,1%) stammt von Fallstudie 1, bei der zwei Subsysteme durch einen Aufruf in einem Zyklus sind, so dass dieses Softwaresystem bei 14 Subsystemen einen relativen Zyklenanteil von 9% hat. Weggelassen habe ich den Wert der kombinierten Referenz- und Schichtenarchitektur aus Fallstudie 22 (s. Tabelle C-5 im Anhang). Bei dieser Fallstudie war es in der Architekturanalyse nicht möglich, die Zyklen zwischen den

[25] Diese niedrigen Werte mögen im ersten Moment erstaunlich wirken. Sie wurden von den Teams erreicht, weil sie in ihren Entwicklungsumgebungen durch ein Komponentenmodell einen Kontrollmechanismus eingeführt hatten, der Zyklen zwischen Subsystemen verbietet. Dieser Mechanismus wird im Rahmen der Strategien in Abschnitt 10.2 genauer beschrieben.

Subsystemen im Detail mit dem Architekten zu besprechen. Da dieses Softwaresystem aus insgesamt 210 Subsystemen bestand, war im Rahmen der engen zeitlichen Restriktionen seines Architekten nur eine grobe Aufteilung der Subsysteme möglich. Um an dieser Stelle einen wirklich aussagekräftigen Wert zu erhalten, müsste die Aufteilung in Subsysteme noch einmal zusammen mit dem Architekten überprüft werden.

Für den Umfang der Zyklen im Verhältnis zum Architekturstil ergibt sich ein vergleichbares Bild wie für das Ausmaß. Diese starke Übereinstimmung kommt dadurch zustande, dass auf Subsystem-Ebene nur noch relativ wenige getrennte Subsystemzyklen vorhanden sind und damit der größte Subsystemzyklus häufig der Anzahl der Subsysteme in Zyklen entspricht. (s. Tabelle C-5 im Anhang). Für die Verflochtenheit der Subsystemzyklen und der Reichweite der Zyklen aus Klassenzyklen konnten keine weiteren Erkenntnisse gewonnen werden.

Die Verteilung in Abbildung 8-18 legt für die in dieser Arbeit untersuchten Softwaresysteme den Schluss nahe, dass Schichtenarchitekturen im Vergleich zu Modell- und Subsystemarchitekturen zu einer stärkeren Geordnetheit auf der Subsystem-Ebene führen.

Geordnetheit und Modularität bei Subsystemen

Die hohe Anzahl an Fallstudien (neun von elf), bei denen auf Subsystem-Ebene kein einziger Zyklus gefunden werden konnte, erweckt den Eindruck, als wäre es bei einer Schichtenarchitektur für die Entwicklungsteams einfach, Subsysteme zyklenfrei zu halten. In den Architekturdiskussionen zeigte sich jedoch ein anderes Bild.

Alle Teams berichteten in den Interviews von ihren Schwierigkeiten, die Subsysteme, die ihnen als modulare Einheiten natürlich erschienen, tatsächlich zu bilden, weil dabei eine Reihe von Zyklen auftreten würde. Drei Teams baten darum, die Aufteilung in Subsysteme bei der Architekturanalyse versuchsweise anders vorzunehmen. Abbildung 8-19 zeigt eins der dabei entstandenen Ergebnisse. Die einzelnen farbigen Rechtecke mit Reitern stellen die bei der Analyse mit dem Sotographen definierten Subsysteme dar, die von oben nach unten in fünf Schichten aus gleichfarbigen Subsystemen angeordnet sind. Das Subsystem „Util“ in der untersten Schicht wird von allen übergeordneten Subsystemen benutzt, darum sind die Beziehungen zum Subsystem „Util“ zur besseren Lesbarkeit weggelassen.

In der Soll-Architektur dieser Fallstudie waren lediglich drei Schichten mit insgesamt neun Subsystemen vorgesehen. Eins dieser neun Subsysteme enthielt ca. 80% des gesamten Programmtextes. In Abbildung 8-19 ist dieses große Subsystem als ein grüner gestrichelter Kasten angedeutet. Zwischen den neun in ihm enthaltenen Subsystemen, die ich auf Wunsch der Architekten in der Analyse getrennt habe, waren viele Zyklen zu finden, die zum großen Teil aus Klassenzyklen stammten.

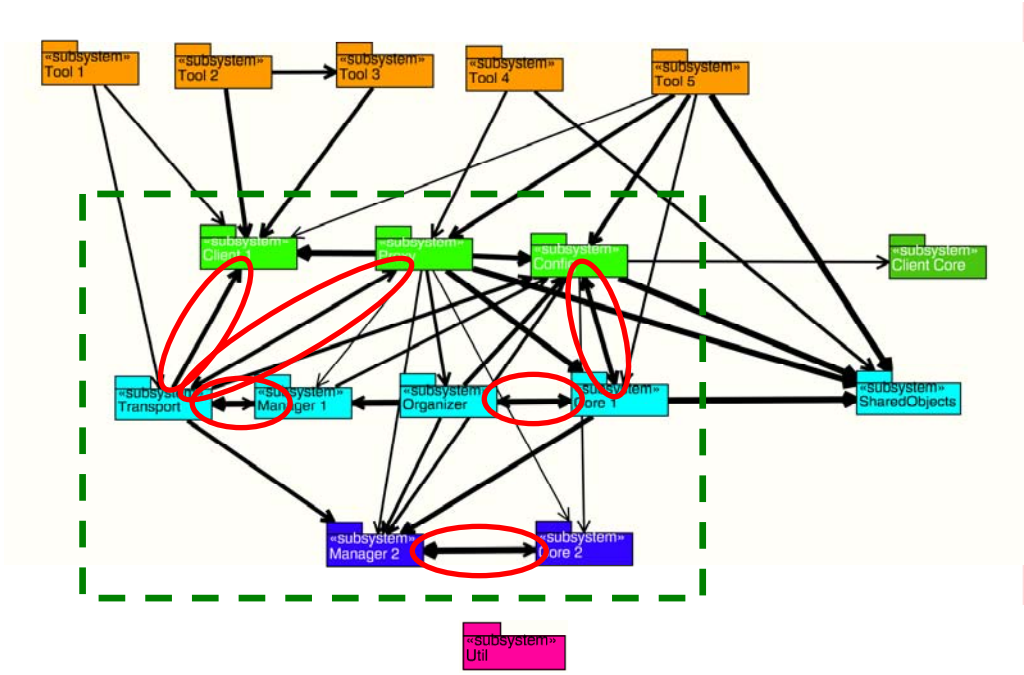

Abbildung 8-19: Subsystembildung und Schichten

Der Unterschied zwischen der gewählten Subsystemaufteilung und der von den Entwicklungsteams gewünschten Aufteilung macht deutlich, dass die Entwicklungsteams gern kleinere und besser zusammenhängende Chunks gebildet hätten. Eine entsprechende Aufteilung der Subsysteme wurde aber nicht zu Beginn der Entwicklung geplant, so dass ein späterer kleinerer Schnitt der Subsysteme ein großes Refactoring mit sich gebracht hätte, um die so sichtbar werdenden Zyklen aufzulösen. Zu diesem Schritt war die Projektleitung nicht bereit gewesen.

8.4. Zusammenfassung

Insgesamt hat sich bei den Untersuchungen mit dem Komplexitätsmodell gezeigt, dass der Architekturstil, den ein Entwicklungsteam verwendet, einen deutlichen Einfluss auf die Architekturkomplexität hat. Referenzarchitekturen sorgen für geringere Architekturkomplexität auf der Klassen-Ebene, speziell für die Faktoren Mustertreue, Geordnetheit und das Kriterium Zusammenhalt. Schichtenarchitektur unterstützt Mustertreue und Geordnetheit auf der Subsystem-Ebene, und Subsystemarchitekturen tragen zum Zusammenhalt und dem Schnittstellenumfang auf der Subsystem-Ebene bei. Einige Teams setzen diese Architekturstile in Kombination ein und erreichen so geringe Architekturkomplexität auf allen Ebenen.

Bei den Untersuchungen zur *Mustertreue* wurde offensichtlich, dass in Java Möglichkeiten fehlen, die Muster der Subsystem- und der Klassen-Ebene aus der Soll-Architektur und dem Architekturstil auf den Programmtext zu übertragen. Architektinnen und Architekten behelfen sich mit Konventionen bei der Strukturierung des Package-Baums und mit Namenszusätzen für die Zuordnung zu Elementarten.

Bei *Modularität* wurde deutlich, dass über die Größe von Subsystemen keine einheitliche Vorstellung existiert; vielmehr betrachteten einige Teams ganze Schnitte als Subsysteme, während andere Teams pro Schicht und Schnitt ein Subsystem bildeten. Auf der Subsystem-Ebene entwickeln sich Schnittstellen in Schichten- und Referenzarchitekturen erst, wenn das Softwaresystem eine Größe von 250.000 RLOC erreicht hat. Subsystemarchitekturen legen auf Schnittstellen besonderen Wert und führen sie von Beginn der Entwicklung an ein.

Für den Faktor *Geordnetheit* waren bei Schichten- und Subsystemarchitekturen auf Klassen-Ebene erheblich größere und verflochtenere Zyklen vorhanden als bei Referenzarchitekturen. Diese Klassenzyklen wirken sich auf der Package- und Subsystem-Ebene aus, weil die meisten von ihnen nicht lokal sind. Bei Referenzarchitekturen fielen besonders die Klassen als komplex auf, für die in der Referenzarchitektur keine Regeln vorhanden waren.

Restriktive Mittel, mit denen die Entwicklerinnen und Entwickler eingeschränkt werden, um Zyklenfreiheit zwischen Subsystemen oder Angaben von Schnittstellen in xml-Dateien zu erzwingen, führten nicht dazu, dass die Entwicklerinnen und Entwickler weniger komplexe Architekturen bauten. Vielmehr entsteht neue Komplexität, weil die Entwickler unter Zeitdruck Workarounds suchen, damit sie trotz der Beschränkungen das konstruieren können, was ihnen notwendig erscheint.

Was Entwicklungsteams tatsächlich unterstützt und wie sie im Entwicklungsprojekt vorgehen können, um geringe Architekturkomplexität zu erreichen, wird in den folgenden beiden Kapiteln untersucht.

9. Architekturzentrierte Softwareentwicklung

In meiner Untersuchung habe ich Softwaresysteme in einer Momentaufnahme analysiert und zum Teil auch vermessen. Mit dem Begriff *Architekturzentrierte Softwareentwicklung* gehe ich einen Schritt weiter und rücke den prozessbezogenen Aspekt von Softwareentwicklung ins Zentrum der Betrachtung.

Zuerst führe ich mein Verständnis von architekturzentrierter Softwareentwicklung ein, das sich auf die beiden Vorgehensmodelle RUP/UP und STEPS abstützt. Anschließend schlage ich drei Stadien der architekturzentrierten Softwareentwicklung vor, um verschiedene Phasen im „Leben" einer Softwarearchitektur zu benennen. Am Ende des Kapitels werden die Fallstudien diesen Stadien zugeordnet, so dass einige Ergebnisse aus Kapitel 8 reinterpretiert werden können.

9.1. Vorgehensmodelle

In den 1970er Jahren ging man davon aus, dass ein Softwaresystem und seine Architektur einmal am Anfang der Entwicklung entworfen werden (Entwurfsphase) und dann nicht mehr verändert werden. Selbst als man in den 1980er Jahren für den Entwicklungsprozess iterative und inkrementelle Vorgehensmodelle einzusetzen begann, wurde eine Übertragung auf Architektur in der wissenschaftlichen Diskussion nicht vorgenommen (vgl. Floyd et al. 1989; Nagl 1990). Um das Jahr 1995 herum änderte sich diese Sicht dahingehend, dass von einer Weiterentwicklung der Architektur in der Zeit gesprochen wurde (Kruchten 1995, S. 50). Die Architektur wurde nicht mehr am Anfang für das gesamte Projekt festgelegt, sondern konnte im Laufe des Entwicklungsprozesses angepasst werden.

9.1.1. Das generische Vorgehensmodell RUP/UP

Diese Art des Vorgehens wurde mit dem Begriff *architecture-centric* bezeichnet und in das damals entstandene allgemeine Vorgehensmodell, den Unified Process (UP) bzw. den Rational Unified Process (RUP), als eine seiner Eigenschaften integriert (Jacobson et al. 1999, S. 59). Der RUP ist eine spezielle Ausprägung des generischen Prozesses UP und wurde in der Firma Rational entwickelt (Kruchten 1999, S. XVI).

Das Vorgehensmodell RUP/UP beschreibt Entwicklungsprojekte als einen sich zyklisch wiederholenden Ablauf von vier Phasen: Inception, Elaboration, Construction und Transition (Jacobson et al. 1999, S. 318). Jede der vier Phasen hat wiederum eine oder mehrere Iterationen, in denen Tätigkeiten (sog. „Workflows") durchgeführt werden. Die von RUP/UP vorgegebenen Workflows sind: Requirements, Analysis, Design, Implementation und Test. In Abbildung 9-1 habe ich die vier Ebenen des RUP/UP in Anlehnung an Züllighoven schematisch dargestellt (Züllighoven 2005, S. 452).

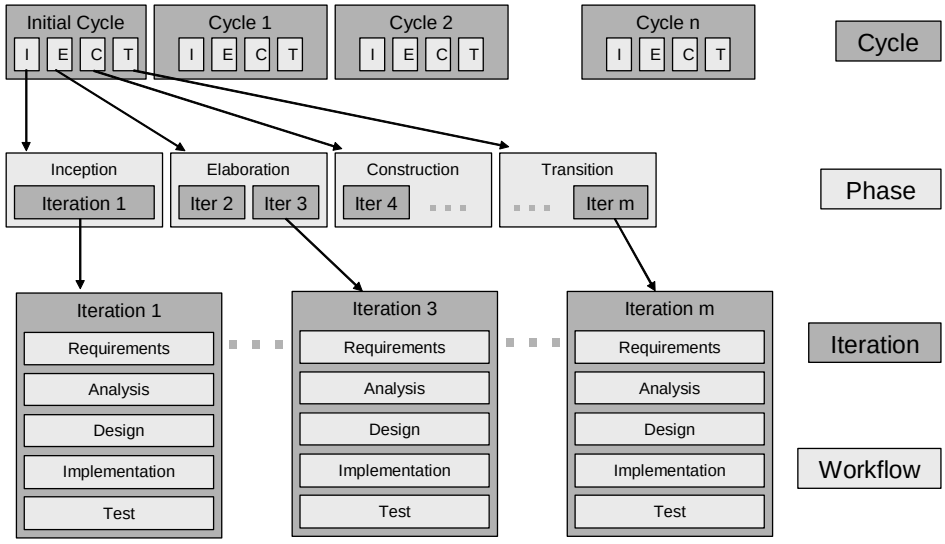

Abbildung 9-1: RUP/UP-Vorgehensmodell

Das besondere an RUP/UP war zu dieser Zeit, dass in jeder Iteration, egal in welcher Phase, alle Workflows ausgeführt werden können. Es wird beispielsweise bereits in der einleitenden Phase des Projekts (Inception) in geringem Umfang implementiert und getestet.

Abhängig von der Phase, in der sich ein Projekt gerade befindet, variiert der Aufwand, der für die einzelnen Workflows geplant wird. Eine mögliche Verteilung des Aufwands stellt Abbildung 9-2 dar, die ich von Jacobson et al. übernommen habe (Jacobson et al. 1999, S. 104). In der Elaboration-Phase wird beispielsweise viel Aufwand für das Design der Software veranschlagt, während in der Construction-Phase weniger Design vorgesehen ist.

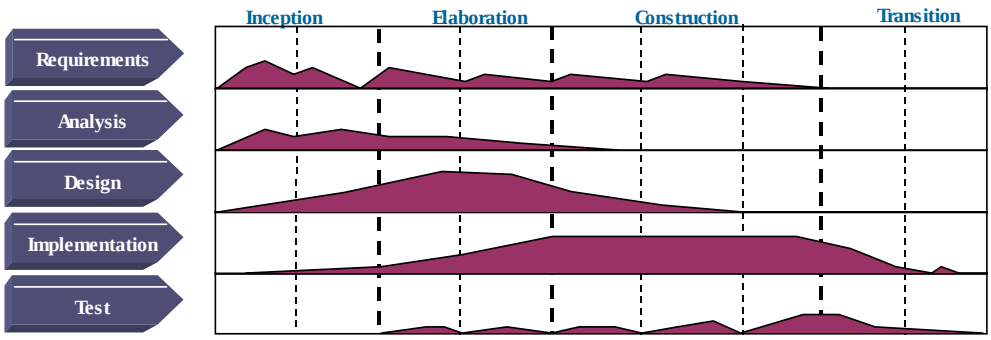

Abbildung 9-2: Aufwand der Tätigkeiten in den Phasen bei RUP/UP

RUP/UP wird von seinen Autoren charakterisiert als ein anwendungsfall-getriebener, iterativer, inkrementeller und architekturzentrierter („architecture-centric“) Prozess (Jacobson et al. 1999, S. 6, 59ff). *Anwendungsfall-getrieben* bedeutet, dass die Anforderungen der Benutzerinnen und Benutzer als sog. Anwendungsfälle (Use Cases) beschrieben werden, die einen kompletten von der Software zu unterstützenden Ablauf beinhalten. Diese Anwendungsfälle werden vom Entwicklungsteam nacheinander in einzelnen Iterationen

umgesetzt. Jede dieser Iterationen führt zu einer lauffähigen Version des Softwaresystems, die getestet und bewertet werden kann. Im Entwicklungsprozess werden regelmäßig neue Inkremente des Softwaresystems zur Verfügung gestellt, so dass das gesamte Vorgehen als *inkrementell* bezeichnet wird. Am Ende eines Zyklus wird schließlich eine einsatzfähige Version des Softwaresystems ausgeliefert. Der *iterative* Charakter des RUP/UP entsteht dadurch, dass die vier Phasen der Entwicklung immer wieder durchlaufen werden, wenn das Softwaresystem erweitert werden soll. Schließlich nennen die Autoren als vierte Eigenschaft: *architekturzentriert*.

Da ich in dieser Arbeit über Architektur spreche, ist besonders dieser Anteil des RUP/UP interessant. Jacobson et al. schreiben, dass die Architektur in den ersten beiden Phasen Inception und Elaboration geplant, in Prototypen ausprobiert und in UML-Diagrammen dokumentiert wird. Dabei kommen Architekturstile und -muster zum Einsatz, die den grundsätzlichen Aufbau des Softwaresystems vorgeben (Jacobson et al. 1999, S. 61). Als ein Architekturmuster, das für viele Arten von Softwaresystemen geeignet ist, schlagen Jacobson et al. die Schichtenarchitektur vor (Jacobson et al. 1999, S. 73f). Für das Ende der Elaboration-Phase führen Jacobson et al. einen Meilenstein ein, mit dem die Architektur für die aktuelle Iteration festgelegt werden soll (Jacobson et al. 1999, S. 103, 380). In der anschließenden Construction-Phase wird auf Basis dieser Architekturvorlage eine erste Version des Softwaresystems entwickelt und am Ende der Transition-Phase ausgeliefert. In den ersten beiden Phasen zu Beginn des darauf folgenden Zyklus wird die Architektur hinterfragt und ggf. an neue Randbedingungen und Erkenntnisse angepasst (Jacobson et al. 1999, S. 64).

Diese von Jacobson et al. geprägte Sichtweise von einer sich im gesamten Entwicklungsprozess kontinuierlich weiterentwickelnden Architektur bezeichne ich im Weiteren als architekturzentrierte Softwareentwicklung.

9.1.2. Das zyklische Vorgehensmodell STEPS

Als Studierende und in meiner Tätigkeit als wissenschaftliche Mitarbeiterin am Arbeitsbereich Softwaretechnik der Universität Hamburg hat mich der unter der Leitung von Christiane Floyd entwickelte Methodenrahmen STEPS – Softwareentwicklung für evolutionäre, partizipative Systementwicklung – geprägt (vgl. Floyd 1992b, 1993; Floyd et al. 1989).

STEPS vertritt als grundlegende Sichtweise einen menschenzentrierten und anwendungsnahen Ansatz, der Softwareentwicklung als Lern- und Kommunikationsprozess begreift. Softwareentwicklung wird hier als eine spezielle Ausprägung von Design begriffen, die die situative Vielfalt anerkennt, die aus dem wechselseitigen Lernprozess zwischen den Beteiligten entsteht.

Wichtig für meine Arbeit ist das zyklische Projektmodell, das in STEPS vorgeschlagen wird (s. Abbildung 9-3). Softwareentwicklung wird als Folge von versionsorientierten Zyklen betrachtet, bei der Herstellung und Einsatz verschränkt sind. Innerhalb des übergeordneten Zyklus werden weitere

Zyklen im Sinne der Iterationen von RUP/UP durchlaufen. Die Wartung wird in die Pflege der aktuellen Version und den Übergang zur nächsten gegliedert. Berücksichtigt werden nicht nur die Aktivitäten der Entwicklerinnen und Entwickler, sondern auch die der Anwenderinnen und Anwender sowie ihre Interaktion, indem getrennte und partizipative Aufgaben benannt werden. Das Softwaresystem wird durch Prototyping über Ausbaustufen hin zu einer einsetzbaren Version entwickelt und kontinuierlich erweitert.

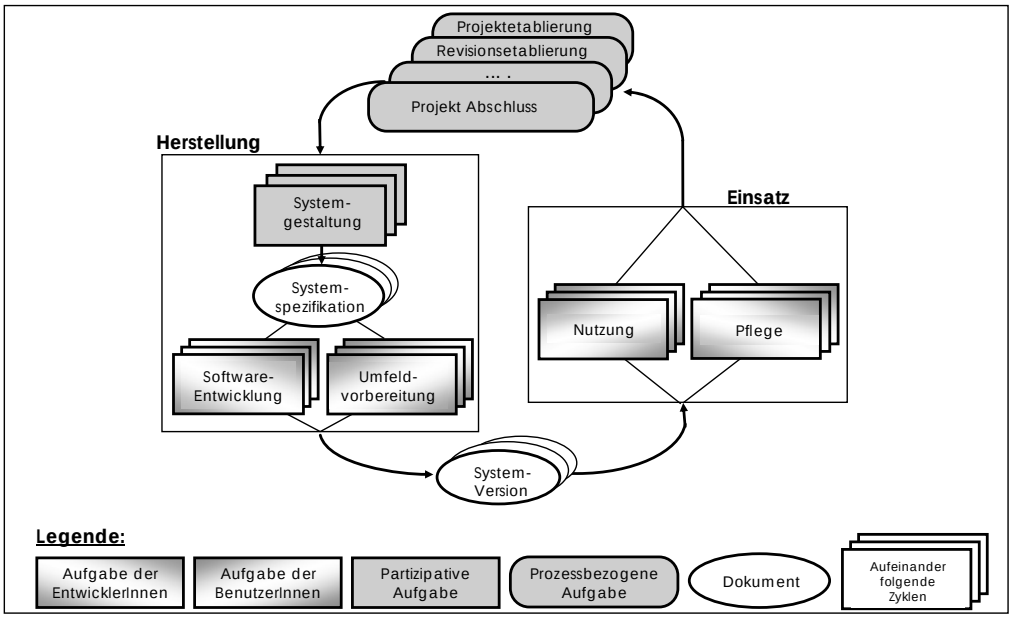

Abbildung 9-3: Das zyklische Projektmodell in STEPS

Die Architektur hat in STEPS immer eine wichtige Rolle gespielt, wurde aber nicht explizit im Projektmodell verankert. Für diese Arbeit habe ich daher das Projektmodell von STEPS um Aufgaben erweitert, die bei architekturzentrierter Softwareentwicklung entstehen.

9.2. Stadien der architekturzentrierten Softwareentwicklung

RUP/UP und STEPS beschreiben jeweils ein zyklisches Vorgehen, bei dem die Architektur ein wichtiger Bestandteil ist. Meine Erfahrungen aus den letzten Jahren sind, dass die Zyklen, in denen Software entwickelt wird, aus Sicht der Architektur unterschiedliche Schwerpunkte haben. Aus diesem Grund führe ich drei Stadien der architekturzentrierten Softwareentwicklung ein, die als eigenständige Teilprozesse jeweils spezifisches Handeln erfordern.

9.2.1. Entwerfen der Architektur

Bei der Neuentwicklung eines Softwaresystems befindet sich ein Entwicklungsteam im initialen Zyklus, der bei STEPS mit der Projektetablierung als erste prozessbezogene Aufgabe eingeleitet wird. In diesem initialen Zyklus hat das Entwicklungsteam die Aufgabe, die Architektur nach sinnvollen Kriterien auszugestalten, um die Anforderungen in eine adäquate Lösung

umzusetzen. Im Phasenmodell wird diese Aufgabe als Entwurf bezeichnet (Balzert 1998). Für das erste Stadium der architekturzentrierten Softwareentwicklung übernehme ich diesen Begriff und bezeichne es als: *Entwerfen der Architektur*.

Dabei fasse ich das Wort Entwurf im Sinne von RUP/UP und der Design-Sicht (Floyd 1992a, S. 93f) weiter als im Phasenmodell. Entwurf besteht nicht nur darin, dass die Architektur mit Dokumenten geplant wird, sondern darin, dass die Architektur in einem verschränkten Prozess aus Planung und Implementierung entworfen wird. Für die Implementierung werden Prototypen eingesetzt, so dass sich die Architektur langsam stabilisieren kann. Bei der Implementierung der Prototypen werden Fragen offensichtlich, die durch einen rein auf Papier erstellten Entwurf der Architektur nicht beantwortet werden. Das Entwicklungsteam muss daher im Rahmen der Implementierung kontinuierlich entwerfend tätig sein.

Abbildung 9-4 stellt das zyklische Projektmodell von STEPS dar, angereichert um die Aufgaben der architekturzentrierten Softwareentwicklung. Aus architekturzentrierter Sicht müssen bei der Projektetablierung den Anwenderinnen und Anwendern bzw. ihrem Management der Nutzen und die Kosten von Architektur vermittelt werden (s. Abbildung 9-4).

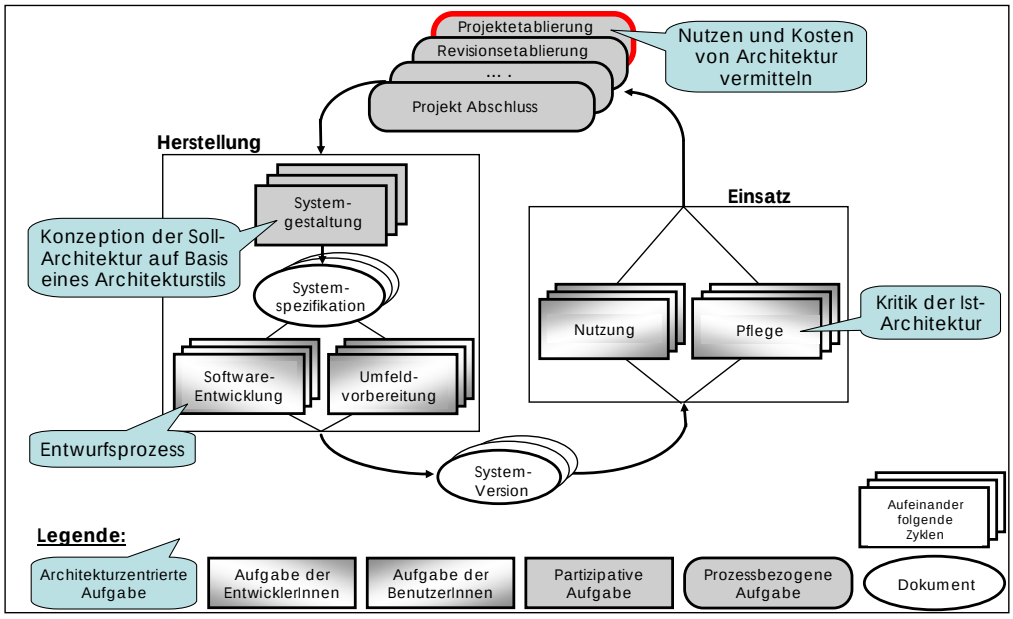

Abbildung 9-4: Vorgehen im Stadium Entwerfen der Architektur

Ist dieser erste Schritt gelungen, so kann mit der Konzeption der Soll-Architektur begonnen werden. Grundlage für diese Aufgabe sollte ein Architekturstil sein. Welcher Architekturstil sich für diesen initialen Zyklus besonders eignet, werde ich in Kapitel 10 in einem Leitfaden deutlich machen.

Nachdem die Soll-Architektur konzipiert ist, wird das Softwaresystem und parallel seine Architektur entwickelt. Ist das Softwaresystem ausgeliefert worden, so beginnt für das Entwicklungsteam die Aufgabe der Pflege. Aus

architekturzentrierter Sicht sollte das Entwicklungsteam im Rahmen der Pflege damit anfangen, die Ist-Architektur des Softwaresystems kritisch zu hinterfragen und erste Refactoring-Ideen für den nächsten Zyklus zu erarbeiten (s. Abbildung 9-4).

Viele Vorgehensmodelle der 1970er und 1980er Jahre beschränken sich auf dieses Stadium der architekturzentrierten Softwareentwicklung. Sie geben methodische Anleitung für die Neuentwicklung von Software, behandeln die Weiterentwicklung aber eher stiefmütterlich. Ein Großteil heutiger Entwicklungsprojekte beschäftigt sich aber nicht mehr mit der Neuentwicklung von Softwaresystemen, sondern modifiziert bereits existierende Anwendungen. Man geht allgemein davon aus, dass an die 70% der Kosten für Entwicklungsprojekte durch die Veränderung vorhandener Anwendungen entstehen (Simon et al. 2006, S. 9). Um die Weiterentwicklung von der Neuentwicklung abzugrenzen, führe ich im nächsten Abschnitt das zweite Stadium der architekturzentrierten Softwareentwicklung ein.

9.2.2. Erhalten der Architektur

Wird ein Team beauftragt, die Entwicklung an einem vorhandenen Softwaresystem wieder aufzunehmen, so kann man nicht mehr davon sprechen, dass die Architektur entworfen wird. Das Entwicklungsteam muss mit einer vorhandenen Architektur umgehen. Die aktuellen Änderungen müssen entweder in die vorhandene Architektur eingepasst werden oder aber ihr muss eine neue Richtung geben werden, weil die vorhandene Architektur im Lichte der Erweiterungen nicht mehr tragfähig ist.

Die Schwierigkeiten, die bei der Erweiterung von Softwaresystemen auftreten, wurden bereits Anfang der 1980er Jahre von Lehman als Gesetze der Programm-Evolution veröffentlicht. Das zweite Gesetz bezieht sich besonders auf die sich verschlechternde Struktur aufgrund von Erweiterungen: „As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.“ (Lehman 1980, S. 1068). Ich spreche deshalb von einem weiteren Stadium der architekturzentrierten Softwareentwicklung: dem *Erhalten der Architektur*.

In Abbildung 9-5 sind die architekturzentrierten Aufgaben für dieses Stadium in das STEPS-Projektmodell integriert. Der Schwerpunkt liegt in diesem Stadium auf der Überarbeitung und Anpassung der bisherigen Architektur, was in der Regel zu Refactoring führen muss. Dass Refactorings unvermeidbar sind, lässt sich dem Management, das die Erweiterung bezahlen soll, oft noch schlechter vermitteln als der Nutzen von Architektur. Aber gerade Refactorings sind für das Erhalten der Architektur besonders wichtig. Welche Architekturstile diesen Refactoring-Prozess unterstützen können, ist Thema des letzten Kapitels zu Strategien und Leitfaden.

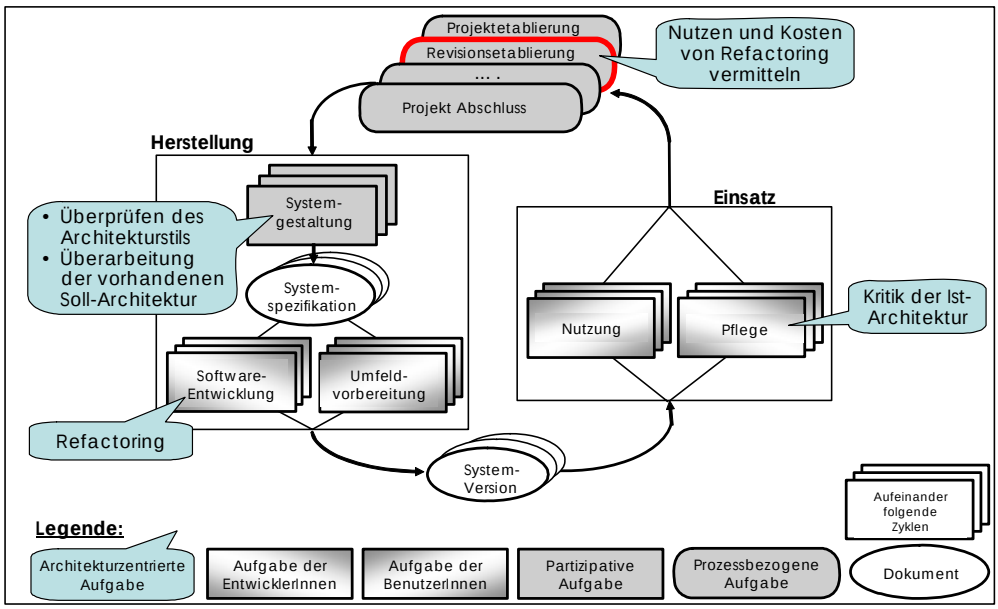

Abbildung 9-5: Vorgehen im Stadium Erhalten der Architektur

In den meisten Projekten, mit denen ich in Kontakt war, wurde beim Erhalten der Architektur „nur noch" das implementierte Softwaresystem mit seiner Ist-Architektur weiterentwickelt. Die auf Papier beschriebene geplante Soll-Architektur wurde vom Entwicklungsteam nicht mehr gepflegt und hatte ihren Bezug zur wirklich implementierten Ist-Architektur verloren. Die Soll-Architektur existiert in diesem Stadium häufig nur noch in den Köpfen der Entwicklerinnen und Entwickler und wird in gemeinsamen Diskussionen bzw. im implementierten Programmtext transportiert. An diesem Punkt setzt der Faktor Mustertreue an, mit dem im Komplexitätsmodell überprüft wurde, ob die Soll-Architektur auf die Ist-Architektur abgebildet werden kann.

9.2.3. Erneuern der Architektur

Die beiden Stadien Entwerfen und Erhalten sollten in einer idealen Welt ausreichen, um die „Lebensspanne" eines Softwaresystems und seiner Architektur abzudecken. In der Praxis trifft man jedoch Softwaresysteme an, bei denen die Architektur nicht erhalten, sondern beschädigt bzw. zerstört worden ist. Wird ein Entwicklungsteam mit einem solchen Altsystem konfrontiert, so kann es die Architektur nicht erhalten, sondern lediglich erneuern. Das Entwicklungsteam befindet sich im dritten Stadium der architekturzentrierten Softwareentwicklung: dem *Erneuern der Architektur*.

In Abbildung 9-6 sind die architekturzentrierten Aufgaben für dieses Stadium in STEPS eingeordnet. In diesem Stadium muss das Thema Architektur bei den Entwicklerinnen und Entwicklern und dem Management neu verankert werden. Falls ein Architekturstil benutzt wurde und zu Beginn eine Soll-Architektur konzipiert wurde, müssen beide hinterfragt und überarbeitet werden. Das Softwaresystem muss tiefgreifend umstrukturiert werden, wobei Refactoring in Kombination mit Testen eingesetzt werden kann.

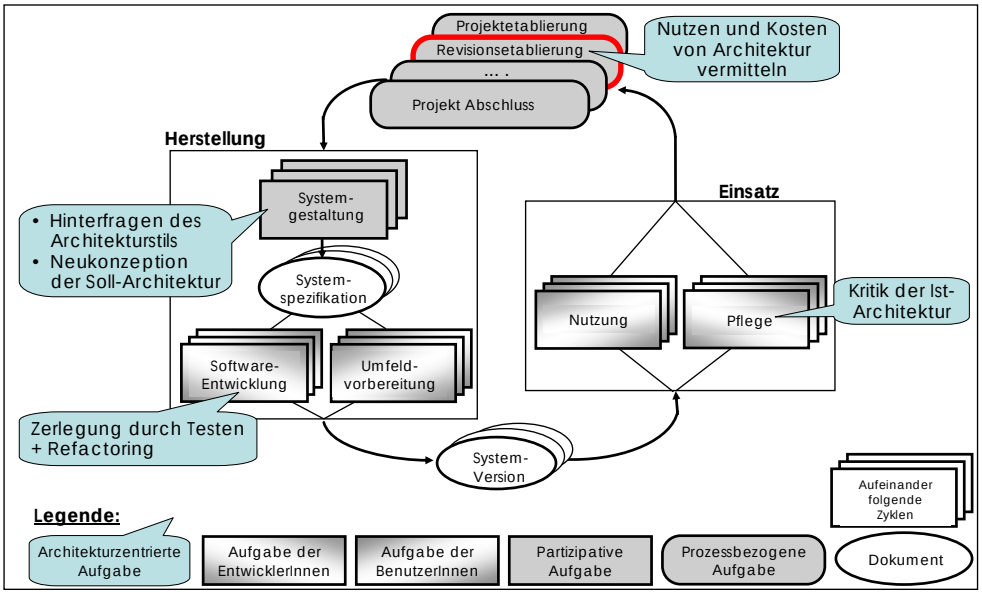

Abbildung 9-6: Vorgehen im Stadium Erneuern der Architektur

Ob es notwendig ist, die Architektur eines Softwaresystems zu erneuern, lässt sich anhand der folgenden Merkmale entscheiden:

- Es gibt kein Mitglied des Entwicklungsteams, das über eine Gesamtarchitektur Auskunft geben kann. In manchen Fällen sind Teilverantwortliche vorhanden, die aber keinen Überblick über die gesamte Architektur haben.
- Das Entwicklungsteam klagt über die Schwierigkeiten, die Erweiterungen am Softwaresystem mit sich bringen, und möchte das Softwaresystem am liebsten neu entwickeln.
- Der Package-Baum des Softwaresystems und die Klassen selbst geben nur rudimentäre Hinweise auf ein Muster, nach dem sie benannt und gebildet wurden, bzw. das ursprünglich geplante Muster wird ständig verletzt (s. Mustertreue im Abschnitt 6.3).

In den Fallstudien konnte ich einige Ursachen beobachten, die dafür verantwortlich sind, dass die Architektur eines Softwaresystems erneuert werden muss:

- *Keine Architektur*: Das Softwaresystem wurde ohne eine gemeinsame Architekturvorstellung im Entwicklungsteam programmiert. Vier Fallstudien habe ich aus diesem Grund aus der Untersuchung zu Architekturkomplexität ausgeschlossen (s. Abschnitt 5.2 zur Auswahl der relevanten Fälle).
- *Nicht-kommunizierte Soll-Architektur*: Die konzipierte Soll-Architektur wurde im Entwicklungsteam nicht transportiert, so dass keine einheitliche Vorstellung vorhanden war. Verstärkt tritt dieses Problem auf, wenn in einer Organisation eine Tren-

nung in Implementierungs- und Wartungsteam vorgenommen wurde.

- *Nicht-kommunizierter Architekturstil*: Der Architekturstil ist den Teammitgliedern im Entwicklungs- oder Wartungsteam nicht durchgehend bekannt. Ein gutes Beispiel hierfür ist das Softwaresystem aus Fallstudie 5, bei dem die Referenzarchitektur nicht an die Wartungsentwickler weitergegeben wurde.
- *Geduldete Abweichungen ohne anschließendes Refactoring*: Unter Zeit- und Kostendruck werden Erweiterungen vorgenommen, die eigentlich ein Refactoring voraussetzen würden. Das Refactoring wird auf unbestimmbare Zeit hinausgeschoben.
- *Schwacher Architekturstil*: Die Regeln des Architekturstils sind zu schwach, um auf Klassen- und Subsystem-Ebene der Architektur Vorgaben zu machen. Dadurch hat das Entwicklungsteam nur auf einer Ebene eine gemeinsame Vorstellung der Architektur. Die Ausgestaltung der anderen Ebenen ist jeder Entwicklerin und jedem Entwickler selbst überlassen. Fallstudie 11, 14 und 15 gehören zu dieser Kategorie. Bei ihnen ist trotz der eingesetzten Schichtenarchitektur immer noch hohe Architekturkomplexität vorhanden.

Dass die Architektur von Softwaresystemen gepflegt werden muss, damit sie nicht in einen unwartbaren Zustand kommt, ist kein neues Phänomen, sondern wurde bereits in den 1980er Jahren erkannt (Lehman 1980, S. 1068). Parnas und andere führten den Begriff *Software Aging* ein, um dieses Phänomen zu beschreiben (vgl. Bianchi et al. 2003; Parnas 1994).

Peter Naur wies darüber hinaus auf die Schwierigkeit hin, dass ein Softwaresystem von Entwicklerinnen und Entwicklern gepflegt wird, die nicht bei seiner Entwicklung dabei waren. Das vorrangige Ziel beim Programmieren ist aus seiner Sicht die Bildung einer Theorie über das Softwaresystem. Kann diese Theorie nicht weitergegeben werden, so ist das Programm „tot“ (vgl. Naur 1985).

An dieser Stelle setzt ein Zweig der Softwaretechnik an, der als *Reengineering* bzw. *Reverse-Engineering* bezeichnet wird (vgl. Yourdon 1992). In der Reengineering- und auch in der Refactoring-Literatur wurden verschiedene Ansätze für den Umgang mit Altsystemen vorgeschlagen (vgl. Bianchi et al. 2001; Deursen et al. 2004; Feathers 2004; Fowler et al. 2001; Kerievsky 2004). Eine bessere Lösung wäre allerdings, wenn das Projektteam rechtzeitig mit Maßnahmen gegen Software Aging beginnt und wenn die Entwicklerinnen und Entwickler, die eine Theorie des Softwaresystems und damit auch seiner Architektur gebildet haben, bei der Wartung weiterhin für das Softwaresystem zuständig sind.

9.3. Einordnung der Fallstudien

Für diese Arbeit konnte ich Projekte in allen drei Stadien der architekturzentrierten Softwareentwicklung untersuchen. Drei Projekte (Fallstudie 1, 3 und 6) konnten dem Stadium *Entwerfen der Architektur* zugeordnet werden, weil ihre Softwaresysteme noch nicht im Einsatz waren. Die Unterscheidung zwischen dem zweiten und dem dritten Stadium habe ich anhand der Merkmale aus Abschnitt 9.2.3 vorgenommen. Die Verteilung auf die verschiedenen Stadien lässt sich aus Abbildung 9-7 entnehmen. Im Anhang in Tablelle A-2 sind alle Fallstudien mit dem jeweils für sie identifizierten Stadium aufgeführt.

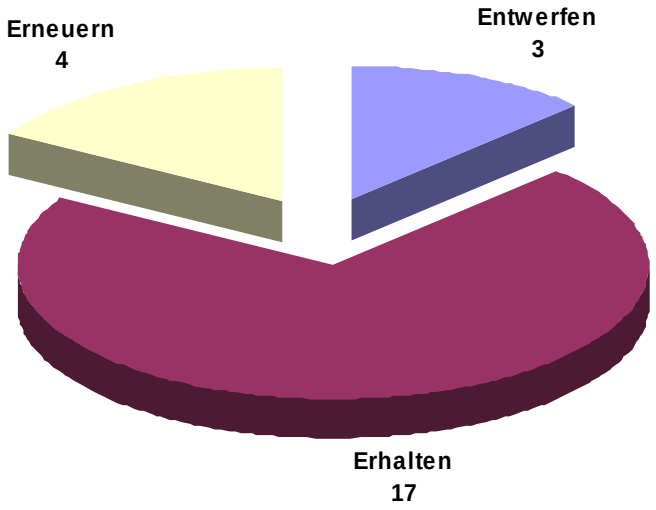

Abbildung 9-7: Verteilung auf die Stadien

Bei den Fallstudien, die zum ersten Stadium zählen, waren die Softwaresysteme bei der Analyse relativ klein (< 25.000 RLOC), noch nicht lange in der Entwicklung (> 2003), und die Architekten berichteten, dass speziell die Subsystem-Ebene immer wieder Veränderungen unterworfen sei. Fallstudie 1 hat eine Referenzarchitektur, Fallstudie 3 eine projekteigene Referenzarchitektur und Fallstudie 6 eine Schichtenarchitektur (s. Tabelle A-1).

In das letzte Stadium *Erneuern der Architektur* habe ich die Fallstudien 5, 11, 14 und 15 eingeordnet. Bei diesen Fallstudien gab es keine Person, die die Architektur des Softwaresystems insgesamt beschreiben konnte (Merkmal 1). Die Entwicklerinnen und Entwickler sprachen von großen Schwierigkeiten bei Wartung und Erweiterung (Merkmal 2), und Mustertreue war kaum vorhanden (Merkmal 3).

Dass Fallstudie 5 in das letzte Stadium gehört, wurde bereits bei den Auswertungen zur Geordnetheit in Abschnitt 8.3 deutlich. Die Referenzarchitektur dieses Softwaresystems ist in der Wartung verletzt worden, weil der Architekturstil nicht an das Wartungsteam kommuniziert werden konnte. Dieses Softwaresystem wurde im Anschluss an die Architekturanalyse überarbeitet und ist inzwischen wieder erweiterbar.

Das Softwaresystem aus Fallstudie 11 war vor der Untersuchung von einem Softwaresystem ohne Architektur zu einem Softwaresystem mit Schichtenarchitektur umgebaut worden. Trotzdem hatte der Architekt weiterhin Schwierigkeiten bei der Erweiterung. Die Schichtenarchitektur war zu schwach, um die Architekturkomplexität des Softwaresystems wirklich auf ein akzeptables Maß zu reduzieren. Einer Architekturanalyse hat der Architekt in diesen Fällen zugestimmt, weil er sich Unterstützung bei der Klärung der Frage erhoffte, welcher Aufwand für weitere Strukturverbesserungen zu erwarten sei. Dieses Softwaresystem wurde ein halbes Jahr nach der Architekturuntersuchung durch ein neues abgelöst.

Das Softwaresystem aus Fallstudie 15 hatte denselben Umbau wie Fallstudie 11 von keiner Architektur auf eine Schichtenarchitektur erlebt. Das Entwicklungsteam hatte immer noch Schwierigkeiten mit Erweiterungen und führte einen Großteil seiner Probleme auf „das Chaos in den Subsystemen“ zurück.

In Fallstudie 14 war von Beginn an eine Schichtenarchitektur als Architekturstil verwendet worden. Trotzdem bekam das Softwaresystem von seinen Architekten ein schlechtes Zeugnis ausgestellt. Die Gründe sah der Architekt zum Teil in der Tatsache, dass an diesen Softwaresystemen Programmieranfänger mitgewirkt hatten, denen die Soll-Architektur und der Architekturstil nicht ausreichend vermittelt werden konnten. Dass Programmieranfänger mehr Fehler bei der Umsetzung eines Architekturstils machen, wird u. a. von Becker et al. beschrieben (Becker-Pechau et al. 2006, S. 35). Außerdem berichtete der Architekt von verschiedenen Erweiterungen, die Refactorings notwendig gemacht hätten. Diese Refactorings wurden aber nicht umgesetzt.

Bei den verbleibenden siebzehn Fallstudien handelte es sich um Softwaresysteme, die von ihren Architekten als beherrschbar eingestuft wurden. Diese Softwaresysteme werden laufend gewartet, und es wird neue Funktionalität eingebaut.

Auffällig ist, dass die Fallstudien aus dem letzten Stadium *Erneuren der Architektur* eher kleinere Softwaresysteme enthalten (23.000 bis 112.000 RLOC). Es sind also nicht die größten Softwaresysteme, die für nicht mehr wartbar gehalten werden. Gerade bei den größeren Softwaresystemen (> 500.000 RLOC) wurde deutlich, dass diese Systemgröße nur erreicht werden konnte, weil sich die Architektinnen und Architekten von Beginn der Entwicklung an und bei jeder Erweiterung mit der Architektur auseinandersetzten und sie laufend überprüften. Diese Fallstudien waren für meine Untersuchung von besonderem Interesse, weil ich bei ihnen die Strategien zur Komplexitätsreduktion direkt beobachten konnte.

Der Vollständigkeit halber finden sich im Anhang Diagramme, die zeigen, dass das jeweilige Stadium der architekturzentrierten Softwareentwicklung keinen Einfluss auf die Anzahl der Zyklen auf der Klassen-, Package- und Subsystem-Ebene hat. Die Ergebnisse aus Kapitel 8 behalten also auch vor der in diesem Kapitel eingeführten Einteilung in Stadien ihre Gültigkeit (s. Abbildung A-9: Stadium und Zyklenausmaß, Abbildung A-10: Stadium und

Zyklenausmaß auf der Package-Ebene und Abbildung A-11: Stadium und Zyklenausmaß auf der Subsystem-Ebene).

9.4. Zusammenfassung

In diesem Kapitel habe ich architekturzentrierte Softwareentwicklung anhand von RUP/UP und STEPS eingeführt und die von mir konzipierten Stadien Entwerfen, Erhalten und Erneuern von Architektur vorgestellt. Befindet sich ein Entwicklungsteam im Stadium Entwerfen der Architektur, so entwickelt es ein Softwaresystem neu und hat noch keine Version ausgeliefert. Wird ein vorhandenes Softwaresystem erweitert, dem eine Architektur zugrunde liegt, so ist das Entwicklungsteam im Stadium Erhalten der Architektur. Muss schließlich ein Softwaresystem mit einer degenerierten Architektur überarbeitet werden, so ist das Stadium Erneuern der Architektur erreicht.

Für diese Arbeit sind besonders die Stadien Erhalten und Erneuern interessant. In diesen Stadien wird die Architekturkomplexität erst wirklich sichtbar und bereitet den Entwicklungsteams Probleme. Architekturkomplexität taucht aber nicht plötzlich im Stadium Erhalten oder Erneuern auf, sondern hat ihren Ursprung bereits im ersten Stadium. Sie nimmt ihren Anfang, wenn ohne Architekturstil eine Soll-Architektur entworfen wird oder gänzlich ohne Soll-Architektur mit der Implementierung begonnen wird.

Daraus ergeben sich zwei Fragen: Was müssen Entwicklungsteams tun, um nicht ins Stadium Erneuern der Architektur abzurutschen? Wann sollten sie mit entsprechenden Maßnahmen beginnen? Das nächste Kapitel wird – mit der Vorstellung von Strategien zur architekturzentrierten Softwareentwicklung und einem Leitfaden für ihren Einsatz – Antwort auf diese Fragen geben.

10. Strategien und Leitfaden

Im letzten Kapitel habe ich vorgeschlagen, dass in allen Stadien der architekturzentrierten Softwareentwicklung Architekturstile eingesetzt werden sollten, da sonst Softwaresysteme zu komplex werden. Ob ein Entwicklungsteam es schafft, die Architektur seines Softwaresystems zu erhalten, hängt aber ebenso von der Möglichkeit ab, den Architekturstil im Team zu etablieren und seine Umsetzung zu kontrollieren. Welche Strategien dabei relevant sind und wann welche Strategie geeignet ist, werde ich in diesem Kapitel aufzeigen.

Ich schlage in diesem Kapitel vier Strategien vor, die auf den Erkenntnissen aus den Fallstudien und meiner eigenen Erfahrung als Architektin beruhen. Als Namen habe ich für die Strategien das Ziel eingesetzt, das bei ihrem Einsatz verfolgt wird. Meine Diskussions- und Interviewpartner haben von sich aus nicht davon gesprochen, dass sie eine bestimmte Strategie befolgen. Vielmehr habe ich den Fallstudien jeweils eine oder mehrere Strategien zugeordnet und verwende Beispiele aus den Fallstudien, um die Vor- und Nachteile der Strategien zu illustrieren.

Im Folgenden stelle ich die vier Strategien vor. Anschließend beschreibe ich, welche Mittel in den Fallstudien eingesetzt wurden, um die Einhaltung der Strategien sicherzustellen. Den Abschluss dieses Kapitels bildet ein Leitfaden, der Entwicklungsteams dabei unterstützen kann, die passende Strategie für ihre Projektsituation und ihr Softwaresystem auszuwählen.

10.1. Strategien zur Komplexitätsbewältigung

Mein Verständnis von Strategien baue ich auf Luhmanns Theorie der sozialen Systeme auf (s. Abschnitt 2.2). Luhmann verwendet den Begriff Strategie für anpassbare Handlungsprogramme, die von einem sozialen System bewusst geplant werden, um Komplexität zu reduzieren. Überträgt man diese Sichtweise auf architekturzentrierte Softwareentwicklung, so müssen Entwicklungsteams Strategien rechtzeitig im Entwicklungsprozess einplanen und einsetzen, um Architekturkomplexität zu beherrschen. An den führenden Architekturschulen wird der Begriff Strategie in diesem Sinn verwendet; allerdings wird dort kein Bezug zu Luhmann hergestellt (Bachmann et al. 2003, S. 6f; Bass et al. 2003, S. 100f; Hofmeister et al. 2000, S. 366; 2005, S. 188; Nord et al. 1999, S. 5).

10.1.1. Schichtung

Bei dieser Strategie wird der Architekturstil *(erweiterte) Schichtenarchitektur* verwendet; allerdings wird nur der Aspekt der Schichtung, nicht aber die Bildung von Schnittstellen berücksichtigt. Schnittstellenbildung betrachte ich als eigenständige Strategie und stelle sie im nächsten Abschnitt vor. Das Ziel dieser Strategie ist, dass zwischen Schichten keine Zyklen vorkommen. Die Schichtung kann sich dabei sowohl auf horizontale Schichten als auch auf die Schnitte einer erweiterten Schichtenarchitektur beziehen.

Diese Strategie sollte sich leicht in einem Entwicklungsteam etablieren lassen, weil die Schichtenarchitektur allgemein bekannt ist. Aus dem gleichen Grund sollten sich neue Teammitglieder, zumindest was die Strategie angeht, gut integrieren lassen. Die fachlichen Schwierigkeiten mit einem unbekannten Softwaresystem nimmt die Strategie den neuen Teammitgliedern selbstverständlich nicht ab.

Die von dieser Strategie geforderte Zyklenfreiheit auf Subsystem-Ebene lässt sich gut mit technischen Mitteln kontrollieren, so dass wenig Zeit für Code-Reviews eingeplant werden muss. Welche technischen Mittel zu empfehlen sind, werde ich in Abschnitt 10.2 erläutern. Schwierigkeiten sind zu erwarten, wenn festgelegt werden muss, was zu einem Subsystem gehört und wie es intern gestaltet ist; denn diese Strategie macht keine Vorgaben über die Zuständigkeiten von Subsystemen.

In den Fallstudien setzten sechs Teams allein auf Schichtung; sieben weitere Teams kombinierten Schichtung mit anderen Strategien. Alle Entwicklungsteams, denen ich diese Strategie zugeordnet habe, waren sich einig, dass Zyklenfreiheit auf Schichten-Ebene sinnvoll ist und ihnen die Entwicklung ihres Softwaresystems vereinfacht. Inhalt der Architekturdiskussionen im Entwicklungsprozess war meistens, wie neue Funktionalität in die vorhandenen Subsysteme integriert werden kann, ohne dass die Schichtung verletzt wird. Die für die Subsysteme zuständigen Teammitglieder konnten bei der internen Struktur eigenständig ihre Ideen verwirklichen. In den einzelnen Subsystemen wurde bei diesen Fallstudien sehr unterschiedlich programmiert, so dass sich auch erfahrene Teammitglieder in andere Systemteile sowohl inhaltlich als auch architektonisch einarbeiten mussten.

10.1.2. Schnittstellenbildung

Der Fokus dieser Strategie liegt darauf, die Schnittstellen von Subsystemen als Dienste zu gestalten und ihre Verwendung zu kontrollieren. Der passende Architekturstil ist im Grunde die *Subsystemarchitektur*. Teams, die auch die Strategie Schichtung einsetzen, werden als Architekturstil allerdings eher von einer *Schichtenarchitektur* sprechen, bei der sie tatsächlich Schnittstellen verwenden.

Diese Strategie sollte sich leicht in einem Entwicklungsteam etablieren und an neue Teammitglieder weitergeben lassen, weil die Bedeutung von Schnittstellen in der Ausbildung hervorgehoben wird.

Die Überprüfung, ob Schnittstellen eingehalten werden, lässt sich mit technischen Mitteln vornehmen (s. Abschnitt 10.2). Aufwand entsteht bei dieser Strategie, wenn ein Team sicherstellen will, dass die Schnittstellen tatsächlich zusammengehörige Dienste bereitstellen und eine Einschränkung auf einen Teil der in einem Subsystem enthaltenen Subsysteme oder Klassen ist.

Zwei Teams (8, 20) setzten allein auf diese Strategie. Andere Teams (19, 21, 22, 23, 24) setzten sie in Kombination mit anderen Strategien ein. Alle Teams, die

nur auf diese Strategie bauten, machten in der Untersuchung deutlich, dass während der Entwicklung Disziplin und regelmäßige Reviews erforderlich sind, um sicherzustellen, dass einerseits die Schnittstellendokumente gepflegt werden und andererseits die Schnittstellen im Programmtext eingehalten werden.

10.1.3. Zyklenfreiheit

Das zentrale Anliegen dieser Strategie ist es, Zyklenfreiheit auf der Klassen-Ebene und allen weiteren Subsystem-Ebenen (also auch der Package-Ebene) zu erreichen. Für diese Strategie ist die *Schichtenarchitektur* die Grundlage, wobei das Prinzip der Schichtung bis auf die Klassen-Ebene ausgeweitet wird.

Das Team aus Fallstudie 24 hat Zyklenfreiheit als einen wichtigen Grund für das Gelingen des Projekts angegeben. Zu Entwicklungsbeginn war die Schichtenarchitektur als Architekturstil festgelegt worden. Ein Jahr später wurde mein Interviewpartner aus Fallstudie 24 als Architekt eingestellt, der Geordnetheit auf allen Ebenen zur grundsätzlichen Richtlinie erhob. Bei der Größe dieses Softwaresystems und einer Entwicklungsabteilung von knapp unter 1000 Personen bestand nach einer kurzen Anfangszeit nicht mehr die Möglichkeit, in Architekturdiskussionen mit allen Entwicklerinnen und Entwicklern eine gemeinsame Strategie zu etablieren. Vielmehr mussten Vorgaben gemacht werden, die für alle gelten. Der Architekt dieses Softwaresystems sagte mir im Interview: „Das (die Geordnetheit auf allen Ebenen) ist mit Sicherheit einer der Punkte, einer der Gründe, aus denen es überhaupt möglich war, so ein großes System auf die Beine zu stellen!“. Dieses Team setzt als technisches Mittel den Sotographen ein, um seine Strategie sicherzustellen (s. Abschnitt 10.2).

Diese Strategie ist wahrscheinlich nicht so einfach in einem Team zu etablieren wie Schichtung oder Schnittstellenbildung, weil die Vermeidung von Klassenzyklen in der Objektorientierung kein so grundlegendes Konzept ist wie Schichten und Schnittstellen, obwohl das Prinzip der Benutzt-Hierarchie bereits in den 1970er Jahren postuliert wurde. Weiterhin bleibt bei dieser Strategie die Schwierigkeit von Schichtung und Schnittstellenbildung bestehen: Die Strategie macht keine Vorgaben für die Zuständigkeiten von Subsystemen und Klassen. Auf der Klassen-Ebene kann dieses Problem mit der nächsten Strategie angegangen werden.

10.1.4. Referenzarchitektur

Die vierte Strategie bezieht sich auf den Architekturstil *Referenzarchitektur*. In dieser Arbeit habe ich zwei in der Literatur veröffentlichte Referenzarchitekturen vorgestellt und darüber hinaus projekteigene Referenzarchitekturen als Möglichkeit beschrieben. Bei der Wahl dieser Strategie muss sich das Entwicklungsteam auf die Referenzarchitektur festlegen, die verwendet werden soll.

Um diese Strategie in einem Entwicklungsteam zu verankern, müssen die Mitglieder entweder schon Erfahrung mit der Referenzarchitektur haben oder

vor Projektbeginn eine spezielle Ausbildung bekommen. In der Referenzarchitektur erfahrene Entwicklerinnen und Entwickler sollten sich zu einem späteren Zeitpunkt problemlos in die Teams integrieren lassen, weil ihnen die Elementarten und Regeln eine Reihe von Anhaltspunkten für die Einarbeitung geben. Hat man ein Team, das in der Referenzarchitektur erfahren ist, so können neue Entwicklerinnen und Entwickler normalerweise problemlos aufgenommen werden. Ohne eine solche Vermittlung ist nicht zu erwarten, dass die Referenzarchitektur erhalten bleibt, wie in den letzten Kapiteln am Beispiel von Fallstudie 5 deutlich zu Tage trat (s. Abschnitt 8.3.1 und 9.3).

Um sicherzugehen, dass die Vorgaben der Referenzarchitektur eingehalten werden, wird eine Reihe von Entwurfsdiskussionen und Code-Reviews notwendig sein. Ein wichtiger Vorteil von Referenzarchitekturen ist, dass sie eine Sprachebene schafft, so dass Entwurfsdiskussionen spezielle Kristallisationspunkte bekommen, an denen das Team Entwurfsentscheidungen ausrichten kann. Durch die Verwendung der WAM- bzw. Quasar-Architektur hat sich in den beteiligten Firmen darüber hinaus ein Milieu gebildet, das es den Entwicklerinnen und Entwicklern erlaubt, projektübergreifend auf Basis ihrer Referenzarchitektur zu diskutieren.

Zwölf Teams aus den Fallstudien habe ich dieser Strategie zugeordnet. Fünf von ihnen kombinierten sie mit einer anderen. Nur ein Team (22) setzte technische Mittel ein, um die Referenzarchitektur zu kontrollieren (s. Abschnitt 10.2).

10.2. Prüfung der Strategien

Von den insgesamt 24 Fallstudien setzten zehn technische Mittel ein, um automatisch zu überprüfen, ob die von ihnen gewählten Strategien in der Ist-Architektur des Softwaresystems eingehalten werden (Fallstudie 6, 13, 14, 15, 16, 17, 19, 22, 23, 24).

Zwei Zeitpunkte für die Überprüfung lassen sich unterscheiden. Entweder wird bereits beim Implementieren in der Entwicklungsumgebung geprüft, so dass Verletzungen von den Entwicklerinnen und Entwicklern direkt erkannt und behoben werden konnten. Oder die Prüfung erfolgte bei der mehrmals wöchentlich durchgeführten Integration, bei der ein lauffähiges Softwaresystem zusammengebaut wird. In diesem Fall müssen Verletzungen der Strategie im Nachhinein an das Entwicklungsteam gemeldet und später behoben werden (s. Abbildung 10-1).

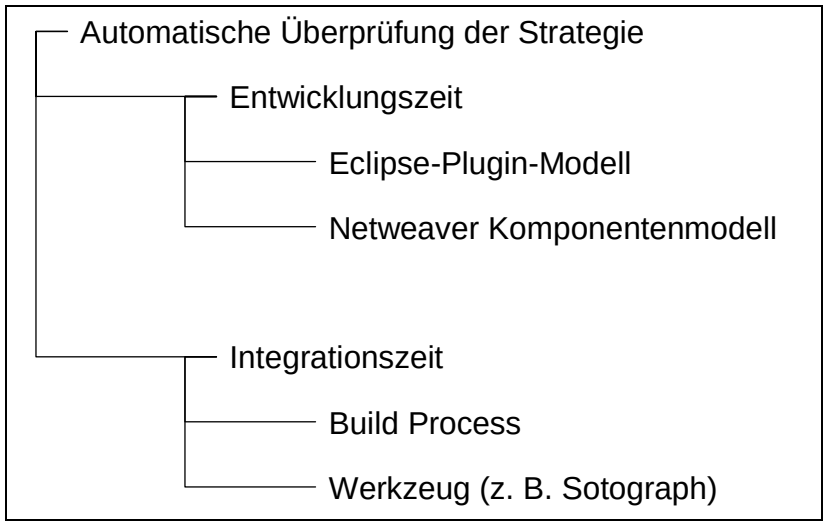

Abbildung 10-1: Mittel zur Prüfung der Strategien

10.2.1. Prüfung zur Entwicklungszeit

Zur Prüfung in der Entwicklungsumgebung wurden in den Fallstudien zwei Komponentenmodelle eingesetzt. Die Fallstudien 6, 15, 17 und 23 verwendeten das Plugin-Modell der Eclipse™-Plattform[26]; in den Fallstudien 13, 14 und 16 wurde das Komponentenmodell aus der NetWeaver™-Plattform[27] von SAP® benutzt.

Eclipse™-Plugin-Modell

Das Eclipse-Plugin-Modell erlaubt den Entwicklerinnen und Entwicklern, in ihrer Entwicklungsumgebung (Eclipse) Subsysteme zu definieren. In Eclipse werden Subsysteme als Komponenten bezeichnet. Jede Komponente enthält einen oder mehrere Package-Bäume und kann andere Komponenten benutzen oder von ihnen verwendet werden. Eclipse stellt sicher, dass die Beziehungen zwischen Komponenten nicht zyklisch sind, indem es verhindert, dass Komponenten in Zyklen übersetzt werden können.

Neben Schichtung unterstützt das Eclipse-Plugin-Modell die Kontrolle des Schnittstellenumfangs und der Abhängigkeiten: Für Komponenten können explizit Schnittstellen definiert werden, die Abhängigkeiten einer Komponente zu anderen Komponenten können festgelegt werden und Erweiterungspunkte können definiert werden, an denen die Funktionalität einer Komponente durch andere Komponenten ausgebaut werden kann.

Die Teams aus den Fallstudien haben das Eclipse-Plugin-Modell auf verschiedenen Wegen in ihr Projekt eingeführt. Ein Team (23) – das Eclipse-Entwicklungsteam selbst – setzt das Eclipse-Plugin-Modell sowohl für Schichtung als auch für Schnittstellenbildung ein, um seine eigene Entwicklung zu kontrollieren.

[26] www.eclipse.org
[27] www.sap.com/platform/netweaver

Zwei Teams (6 und 17) haben von Anfang an mit dem Eclipse-Plugin-Modell gearbeitet, weil sie ihr Softwaresystem oberhalb von Packages zyklenfrei strukturieren wollten. Dieser Wunsch war aus der bei den meisten Teammitgliedern vorhandenen Erfahrung entstanden, dass Softwaresysteme leichter zu beherrschen sind, wenn Subsysteme zyklenfrei sind.

Das letzte Team (15) hatte nach einer längeren Zeit der Entwicklung bemerkt, dass sein inzwischen über 100.000 ROC großes Softwaresystem unbeherrschbar wurde. Um die Struktur zu verbessern, überarbeitete dieses Team das Softwaresystem unter Verwendung des Eclipse-Plugin-Modells hin zu zyklenfreien Subsystemen. Dieser Umbau wurde vom Team als eine erhebliche Anstrengung beschrieben, aber gleichzeitig hielt es das ganze Team für dringend erforderlich. Mit diesem Umbau hat das Team einen ersten Schritt zur Restrukturierung durchgeführt. Die Diskussion mit diesem Team machte jedoch deutlich, dass es sich trotz dieser Verbesserung weiterhin im architekturzentrierten Stadium *Erneuern der Architektur* befindet, weil Erweiterungen für das Team schwer umzusetzen sind.

NetWeaver™ Komponentenmodell

Das NetWeaver-Komponentenmodell ist Teil der SAP NetWeaver Development Infrastructure (NWDI). Neben einer umfassenden Unterstützung des Deployments- und Versionierungsprozesses steht im SAP NetWeaver Development Studio ein Komponentenmodell zur Verfügung. Auch hier werden Subsysteme als Komponenten bezeichnet. Das Komponentenmodell erlaubt es den Entwicklerinnen und Entwicklern, Komponenten mit einer beliebigen Anzahl von Subkomponenten zu definieren. Die Beziehungen zwischen den Komponenten dürfen genau wie beim Eclipse-Komponenten-Modell keine Zyklen enthalten. Neben Schichtung bietet das NetWeaver-Komponentenmodell die Möglichkeit, für jede Komponente eine Schnittstelle zu definieren und die Zugriffsrechte anderer Komponenten auf die Schnittstellenelemente anzugeben.

Drei Teams aus den Fallstudien 13, 14 und 16 haben das NetWeaver-Komponentenmodell erst im Laufe ihrer eigenen Entwicklung einsetzen können, da es nicht von Anfang an zur Verfügung stand. Zyklenfreiheit auf Subsystem-Ebene konnten diese Teams erreichen, waren aber mit der Architektur ihres Softwaresystems insgesamt nicht zufrieden, weil der Schnitt der Subsysteme unausgewogen war (s. Abschnitt 8.3.3 zur Geordnetheit und Modularität auf Subsystem-Ebene). Außerdem setzten sie die Möglichkeit der Schnittstellenbildung nicht ein. Ihre Begründung war, dass die einzelnen Entwicklerinnen und Entwickler der Subsysteme nicht wissen, was die anderen Subsystem-Verantwortlichen benötigen und daher lieber alles an der Schnittstelle zur Verfügung stellen.

10.2.2. Prüfung bei der Integration

Bei der Prüfung zur Integrationszeit sind von vier Teams spezielle Abbruchbedingungen eingefügt worden, die die Integration verhindern, wenn die jeweils gewählte Strategie verletzt wird (Fallstudie 19, 21, 22, 24).

Build Process

Der Build Process, der als Integrationstest in regelmäßigen Abständen mehrmals am Tag durchgeführt wird, ist bei Fallstudie 19 und 24 so gestaltet, dass das Softwaresystem nur zusammengebaut werden kann, wenn es auf Ebene der Subsysteme zyklenfrei ist und wenn keine Schnittstellen verletzt werden. Die zu diesem Zweck eingesetzten Skripte für das Zusammenlinken brechen ggf. mit einer Fehlermeldung ab.

Das Team aus Fallstudie 21 überprüft insbesondere die Schnittstelle zwischen Client und Server bei der Integration. Zu diesem Zweck war die Schnittstelle in ein eigenes Projekt ausgelagert worden. Nur dieses Projekt wurde dem Client-Projekt bei der Integration zur Verfügung gestellt, so dass Zugriffe an der Schnittstelle vorbei zu einem Fehler in der Integration führen.

Werkzeug

Bei Fallstudie 24 setzt der Architekt den Sotographen ein, um Zyklen auf allen Ebenen zu entdecken. Der Zustand des Softwaresystems wird bei der täglichen Integration automatisch analysiert, und die Ergebnisse verschiedener Versionsstände werden in einem Report verglichen. Dieses Monitoring ermöglicht es dem Entwicklungsteam, Veränderungen in der Architektur regelmäßig zu kontrollieren. Durch die ständige Überwachung werden Zyklen früh erkannt, und es lassen sich relativ schnell Gegenmaßnahmen einleiten. Ähnlich wurde in Fallstudie 19 mit dem JDepend[28] gearbeitet, einem OpenSource-Werkzeug zur Qualitätskontrolle für Java-Software.

Das Team aus Fallstudie 22 hatte zu einer Zeit (1998) mit der Entwicklung seines Softwaresystems begonnen, zu der es keine Werkzeuge wie den Sotographen oder JDepend gab. Stattdessen entwickelte dieses Team im Jahr 2000 ein eigenes Prüfprogramm, das anhand von in Textdateien spezifizierten Regeln die Import-Statements im kompilierten Bytecode des Softwaresystems analysiert. Mithilfe dieser Regeln können Verletzungen der Schichten, von Schnittstellen und der projekteigenen Referenzarchitektur gefunden werden.

Zum Zeitpunkt der Untersuchung wurde nur im Team aus Fallstudie 22 das eben beschriebene technische Mittel eingesetzt, um die projekteigene Referenzarchitektur automatisch zu prüfen. Neue Forschungen haben gezeigt, dass Regeln von Referenzarchitekturen mit Hilfe von Analysewerkzeugen überprüfbar gemacht werden können (vgl. Becker-Pechau und Bennicke 2007; Becker-Pechau et al. 2006; Knodel und Popescu 2007). Die Voraussetzung für eine solche Überprüfung ist, dass die Elementarten im Programmtext automatisch identifiziert werden können. Auf der Subsystem-Ebene sind Subsysteme

[28] clarkware.com/software/JDepend.html

und Schichten in der Regel von der Menge her manuell zu beherrschen. Bei Referenzarchitekturen ist die Anzahl der Elemente, die zugeordnet werden müssen, viel größer, so dass ein manuelles Verfahren kaum realisierbar ist. Einheitliche Konventionen für die Elementarten begünstigen die automatische Zuordnung erheblich (Becker-Pechau et al. 2006). Die von Becker-Pechau et al. beschriebene automatische Überprüfung wird zur Integrationszeit durchgeführt. Es laufen zurzeit Forschungsarbeiten, die darüber hinaus die Prüfung zur Entwicklungszeit möglich machen sollen (vgl. Scharping 2007).

10.2.3. Einsatz der automatischen Prüfung

Mit der Größe des Softwaresystems veränderten sich die Mittel, die die Entwicklungsteams einsetzten, um sicherzustellen, dass die Strategie von allen Teammitgliedern eingehalten wird. Bei kleinen Softwaresystemen bedienten sich die Architektinnen und Architekten nur organisatorischer Mittel, wie Code-Review und Entwurfsdiskussion. Bei allen größeren Softwaresystemen (22, 23 und 24) und bei fast allen Schichtenarchitekturen ab 70.000 RLOC (13, 14, 15, 16, 19) wurden technische Mittel eingesetzt (s. Abschnitt 10.2).

Diese Entwicklung hin zur automatischen Überprüfung hat ihre Ursache in der wachsenden Anzahl von Personen, die an einem Softwaresystem mitarbeiten. Das Softwaresystem aus Fallstudie 22 wird beispielsweise von ca. 120 Personen gewartet und weiterentwickelt. An Fallstudie 24 implementieren knapp 1000 Entwicklerinnen und Entwickler. Gemeinsame Architekturdiskussionen werden bei dieser Personenzahl schwierig, und Architekturregeln können nicht mehr so einfach geändert werden, wie es bei kleineren Softwaresystemen der Fall ist. Vielmehr ist es bei großen Softwaresystemen wichtig, dass die vorhandenen Architekturregeln im gesamten Team kommuniziert und eingehalten werden. Bei allen Fallstudien oberhalb von ½ Millionen RLOC machten die Architektinnen und Architekten deutlich, dass sie es für unmöglich halten, dass ihr Softwaresystem ohne automatische Überprüfung bis zur aktuellen Größe entwickelt worden wäre.

10.3. Leitfaden für Strategie

Abschließend bleibt zu klären, für welche Strategien sich Entwicklungsteams am Beginn eines Projekts entscheiden und welche Strategien in späteren Stadien hinzukommen sollen. Auf diese Fragen gibt der nun folgende Leitfaden eine Antwort.

10.3.1. Entwerfen der Architektur

Die erste Strategie, die eingeführt wird, hat sehr viel Einfluss auf den Entwurf des Softwaresystems, so dass die Möglichkeiten, die Strategie in den späteren Stadien zu wechseln, begrenzt sind. Die Wahl der ersten Strategie muss daher mit Bedacht getroffen werden.

Aus meiner Sicht sollte ein Entwicklungsteam im ersten Stadium die Strategie *Referenzarchitektur* wählen (s. 1. Wahl in Abbildung 10-2). Dadurch wird insbesondere auf der Klassen-Ebene für geringe Architekturkomplexität gesorgt. Gerade zu Beginn eines Projekts spielt sich die Entwicklung auf der Klassen-Ebene ab, so dass die Referenzarchitektur ihr volles Potenzial bei der Komplexitätsreduktion entfalten kann. Als Alternative für Teams, die keine Möglichkeit haben, sich eine Referenzarchitektur anzueignen, ist die beste Wahl Zyklenfreiheit (s. 2. Wahl in Abbildung 10-2).

Ausgehend von der ersten Strategie sollten bei der Projektetablierung weitere nachfolgende Strategien geplant werden. Für die Referenzarchitektur sind Schichtung und Schnittstellenbildung sinnvolle Nachfolgestrategien (s. Abbildung 10-2). Für Zyklenfreiheit ist es Schnittstellenbildung. Je nach Größe des Softwaresystems sollte die Nachfolgestrategie entweder noch im ersten Stadium der architekturzentrierten Softwareentwicklung eingeführt oder aber beim Erhalten der Architektur hinzugenommen werden.

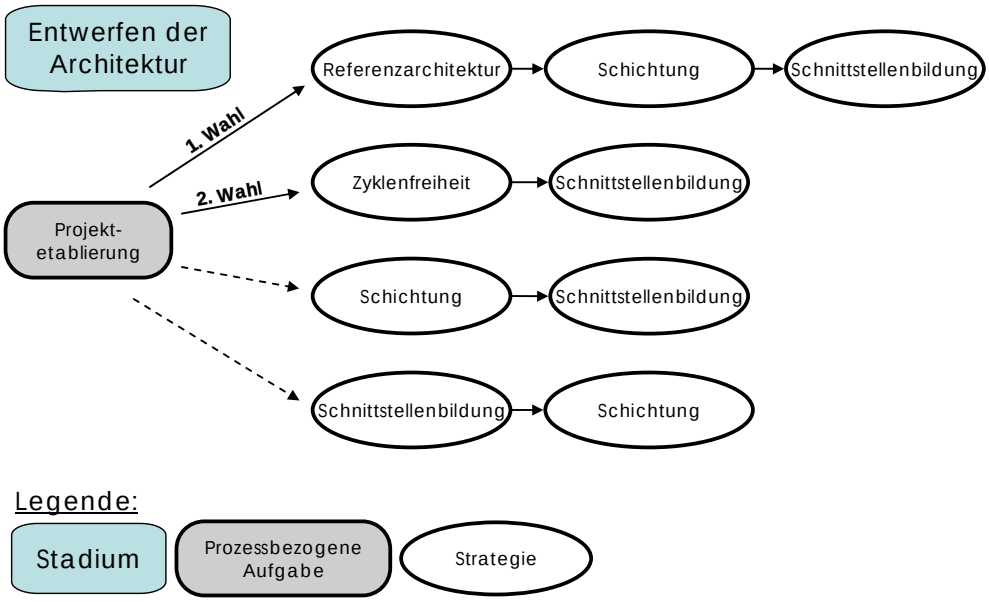

Abbildung 10-2: Strategieplanung beim Entwerfen der Architektur

Schichtung und Schnittstellenbildung sind für das erste Stadium der architekturzentrierten Softwareentwicklung nicht empfehlenswert. Sie geben zwar eine Orientierung, wie das Softwaresystem im Großen strukturiert sein sollte, lassen den Teammitgliedern auf der Klassen-Ebene aber zu viel Freiheit. Falls sie trotzdem als erste Strategie gewählt werden, sollten im Verlauf des Projekts, wie in Abbildung 10-2 zu sehen, weitere Strategien hinzugenommen werden. Beginnt ein Projekt mit der Strategie Schichtung, dann lässt sich vermuten, dass das Team kein Interesse an Zyklenfreiheit hat. Aus diesem Grund wird in der möglichen Strategieplanung in Abbildung 10-2 nach der Strategie Schichtung keine Zyklenfreiheit vorgesehen.

10.3.2. Erhalten von Architektur

Ist die erste Version des Softwaresystems ausgeliefert und wird mit Erweiterungen begonnen, so hat das Entwicklungsteam die Aufgabe, die Architektur zu erhalten. Zu diesem Zeitpunkt sollte im Sinne einer *Projektrevision* überprüft werden, ob es ratsam ist, eine Strategie oder mehrere zusätzliche Strategien einzuführen. Hat das Softwaresystem eine Größe oberhalb von 100.000 RLOC erreicht, sollte auf jeden Fall eine zweite Strategie hinzugenommen werden. Bei den Fallstudien wurden die kleineren Softwaresysteme nur mit einer Strategie (weiter-)entwickelt. Bei den größeren Softwaresystemen wurden zwei bis drei Strategien eingesetzt. Die Architektinnen und Architekten der großen Softwaresysteme (Fallstudien 22 bis 24) haben übereinstimmend berichtet, dass in einem relativ frühen Erweiterungszyklus weitere Strategien eingeführt wurden, auch wenn sie das Wort Strategie dabei nicht direkt verwendeten.

Fingen die Entwicklungsteams mit der Strategie Schichtung an, so nahmen sie in der Regel später die Schnittstellenbildung hinzu. In Abbildung 10-3 empfehle ich darüber hinaus, dass spätestens beim Erhalten einer mit Schichtung entwickelten Architektur Zyklenfreiheit als Strategie hinzugenommen werden sollte, um Architekturkomplexität zu reduzieren. Ein Softwaresystem nachträglich auf eine Referenzarchitektur umzubauen, erscheint mir kaum möglich. In der Regel passen die vorhandenen Klassen nicht zur Referenzarchitektur, und eine Umstrukturierung und Überarbeitung einer großen Anzahl von Klassen wäre nötig. Vor einer solch starken Veränderung schrecken Entwicklungsteams normalerweise zurück.

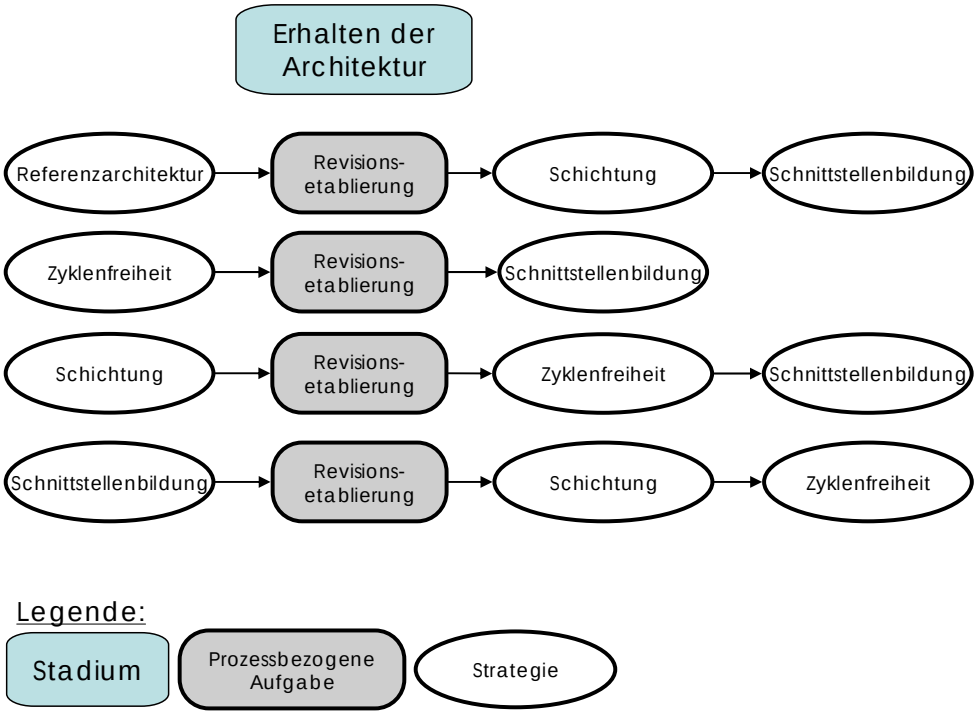

Abbildung 10-3: Strategieplanung beim Erhalten der Architektur

Stehen am Anfang die Strategien Referenzarchitektur oder Zyklenfreiheit, so sollte beim Erhalten der Architektur Schichtung und Schnittstellenbildung hinzugenommen werden (s. Abbildung 10-3), genau wie ich es bei der Strategieplanung aus dem ersten Stadium empfohlen habe (s. Abbildung 10-2).

Bei Referenzarchitekturen lässt sich Schichtung in der Regel leicht hinzunehmen, weil auf der Klassen-Ebene ein guter Grundstein für die Geordnetheit auf der Subsystem-Ebene gelegt worden ist. Um Schnittstellenbildung im Anschluss an Referenzarchitektur oder Zyklenfreiheit einzuführen, ist eine ausführliche Analyse des Softwaresystems erforderlich. Die Schnittstellen der einzelnen Subsysteme müssen untersucht, und eine Einteilung der Klassen bzw. Packages in Subsystem-interne und -externe Klassen muss vorgenommen werden. Diese Restrukturierung des Softwaresystems wird durch die Referenzarchitektur nicht in gleichem Maße vorweggenommen wie die Schichtung, so dass hier zusätzliche Arbeit entsteht.

Der in allen Fallstudien sichtbare Trend, erst bei wachsender Größe mit der Schnittstellenbildung zu beginnen, lässt sich daraus erklären, dass Schnittstellen und Interna eines Subsystems erst bei größeren Softwaresystemen deutlich auseinander gehen. Zu Beginn einer Entwicklung und bei kleineren Softwaresystemen gehören alle Klassen oder Packages in einem Subsystem zu seiner Schnittstelle. Mit einer Einschränkung wird erst begonnen, wenn die Subsysteme größer werden.

10.3.3. Erneuern der Architektur

Hat ein Entwicklungsteam mit einem Softwaresystem zu kämpfen, dessen Architektur nur noch *erneuert* werden kann, so gibt es je nach Ausgangssituation unterschiedliche Möglichkeiten. Wurde das Softwaresystem ohne Strategie entworfen oder sind die Strategien Schichtung, Schnittstellenbildung oder Zyklenfreiheit schlecht umgesetzt worden, so sollte die Architektur als erster Schritt auf der Subsystem-Ebene erneuert werden. Die Strategien, die dabei zum Einsatz kommen sollten, sind Schichtung und Schnittstellenbildung. Diese beiden Strategien sollten in Kombination eingesetzt werden, so dass möglichst unabhängige, lose gekoppelte Subsysteme entstehen (s. Abbildung 10-4).

Anschließend empfiehlt es sich, noch einen weiteren Schritt zu machen und Zyklenfreiheit anzustreben, selbst wenn damit viel Aufwand verbunden ist (s. Abbildung 10-4). Die Fallstudien 11 und 15 haben gezeigt, dass allein der Umbau auf eine Schichtenarchitektur das Softwaresystem noch nicht beherrschbar macht, sondern weitere Refactorings notwendig sind.

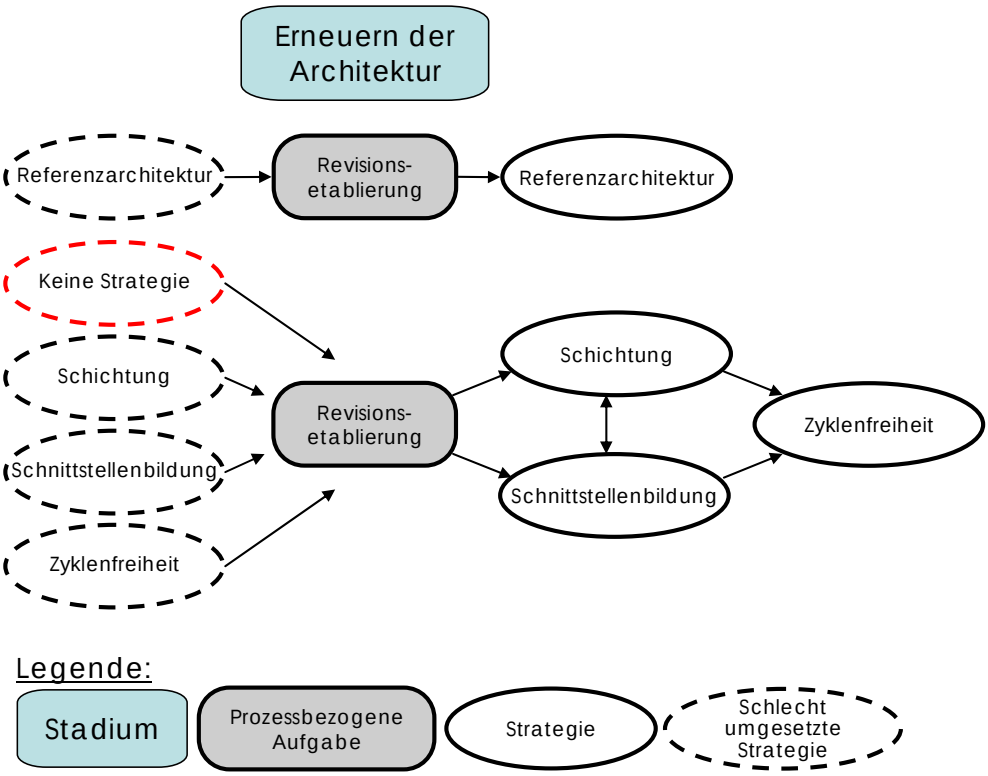

Abbildung 10-4: Strategieplanung beim Erneuern der Architektur

Softwaresysteme, bei denen die Referenzarchitektur schlecht umgesetzt worden ist, sollten zuerst in Richtung der Referenzarchitektur korrigiert werden (s. Abbildung 10-4). Anschließend können Schichtung und Schnittstellenbildung hinzugenommen werden. Fallstudie 5 ist ein gutes Beispiel für dieses Problem. In Abschnitt 8.3 zur Geordnetheit und 9.3 bei der Einordnung der Fallstudien in die Stadien der architekturzentrierten Softwareentwicklung habe ich bereits darüber berichtet, dass das Wartungsteam in Unkenntnis der Referenzarchitektur Erweiterungen vorgenommen hat, die nicht konform zu den Regeln sind.

10.4. Zusammenfassung

In diesem Kapitel habe ich vier Strategien vorgestellt: Schichtung, Schnittstellenbildung, Zyklenfreiheit und Referenzarchitektur. Jede Strategie hat Vor- und jede hat Nachteile, verlangt nach einer geeigneten Einführung und lässt sich mit unterschiedlichen technischen Mitteln prüfen.

Das Kapitel schließt mit einem Leitfaden, mit dem ich die Empfehlung ausspreche, als erste Strategie eine Referenzarchitektur zu wählen und sie anschließend um Schichtung und Schnittstellenbildung anzureichern.

11. Zusammenfassung und Ausblick

In dieser Arbeit habe ich einen weiten Bogen gespannt von Komplexität in der Softwareentwicklung bis zu Strategien und einem Leitfaden zur Komplexitätsbewältigung. Dabei habe ich mir die folgenden Fragen gestellt und sie im Laufe der Arbeit beantwortet: Was ist Architekturkomplexität und wie kann man sie bestimmen? Welche empirischen Erkenntnisse lassen sich über Architekturkomplexität erlangen und was sind ihre Ursachen? Was ist architekturzentrierte Softwareentwicklung? Welche Strategien helfen, Architekturkomplexität zu reduzieren, und nach welchen Kriterien sollten Entwicklungsteams die Strategien auswählen? Den gegangenen Weg und die Ergebnisse fasse ich im Weiteren zusammen.

Zunächst ging es mir darum zu klären, wie sich Komplexität bei der Softwareentwicklung auswirkt. Dazu habe ich in Kapitel 2 verschiedene Ebenen herausgearbeitet, auf denen Entwicklerinnen und Entwickler mit Komplexität konfrontiert werden: Komplexität in der Interaktion im Entwicklungsteam, Verstehenskomplexität und Softwarekomplexität. Um als soziales System Bestand zu haben, müssen Entwicklungsteams Komplexität mithilfe von Strategien beherrschen. Die Aufgabe, Software weiterzuentwickeln, beinhaltet Verstehenskomplexität und verlangt von den Entwicklerinnen und Entwicklern, dass sie mit Softwarekomplexität in Form von problem-inhärenter und lösungs-abhängiger Komplexität umgehen. Daraufhin habe ich als das Thema dieser Arbeit den strukturellen Anteil der *lösungs-abhängigen Komplexität* mit seinen *essentiellen* und *akzidentellen* Anteilen identifiziert.

In Kapitel 3 habe ich objektorientierte *Softwarearchitektur* und *Architekturstile* thematisiert. Softwarearchitekturen lassen sich auf der Klassen- und Subsystem-Ebene modellieren und werden durch Architekturstile in ihrer Struktur beeinflusst. Vier Architekturstile konnte ich in den Fallstudien beobachten: Schichtenarchitektur, Subsystemarchitektur, Erweiterte Schichtenarchitektur und Referenzarchitektur. Schließlich habe ich die Begriffe Soll- und Ist-Architektur eingeführt, um die geplante von der tatsächlich implementierten Architektur abzugrenzen.

Diese Unterscheidung machte es möglich, zwei Quellen von *Architekturkomplexität* zu identifizieren und in einer Definition (s. Definition 3-2) zusammenzufassen:

> *Architekturkomplexität* ist der strukturelle Anteil der lösungsabhängigen Komplexität, der sich aus der Wahl der Architekturelemente und ihrer Beziehungen untereinander ergibt.
>
> Architekturkomplexität speist sich aus zwei Quellen: der *Struktur der Ist-Architektur* und der *Abbildung von Soll-Architektur und Architekturstil auf den Programmtext*.

Die *Struktur der Ist-Architektur* ist schwer zu verstehen, weil sie aus einer Vielzahl von Klassen mit Beziehungen besteht, die zu Subsystemen zusammengefasst werden. Die *Abbildung von Soll-Architektur und Architekturstil auf*

den Programmtext ist eine komplexe Aufgabe, weil die Elemente der Soll-Architektur und des Architekturstils nur mithilfe der Abbildungsvorschrift identifiziert werden können. Weicht die Ist-Architektur von der Soll-Architektur und dem Architekturstil ab, muss zusätzlich mit den Differenzen zwischen beiden umgegangen werden.

Damit war ein Teil der ersten Fragestellung, was Architekturkomplexität ist, beantwortet, und ich konnte mich der zweiten Hälfte, wie Architekturkomplexität bestimmt werden kann, zuwenden.

Das Wissen darum, dass Architekturen komplexe Gebilde sind, hat mich in Kapitel 4 zu der Frage geführt, wie Menschen beim Verstehen, beim Wissenserwerb und bei der Problemlösung mit komplexen Strukturen umgehen. Mit den strukturbildenden Prozessen des *Chunkings*, der *Bildung von Hierarchien* und dem *Aufbau von Schemata* habe ich Wissen der kognitiven Psychologie genutzt, um zu beurteilen, wie objektorientierte Architekturen und Architekturstile zur Verringerung von Architekturkomplexität beitragen. Diese drei Prozesse bildeten die Wurzeln des Komplexitätsmodells in Kapitel 6.

In Kapitel 5 habe ich meinen *Forschungsprozess* beschrieben. Diese Dissertationsschrift ist in einem zyklischen Lernprozess zwischen Untersuchung in der Praxis, Theoriebildung und Überprüfung der Erkenntnisse entstanden. In diesem Lernprozess habe ich 24 Fallstudien durchgeführt. Dabei habe ich einerseits das Werkzeug Sotograph verwendet, um quantitative Ergebnisse zu erzielen, und andererseits Diskussionen und Interviews geführt, um qualitative Ergebnisse zu gewinnen.

Das *Modell für Architekturkomplexität* habe ich parallel zu den Fallstudien entwickelt. Es baut auf den zwei Quellen von Architekturkomplexität sowie den drei Prozessen für den Umgang mit komplexen Strukturen auf. In Kapitel 6 habe ich für das Komplexitätsmodell drei Faktoren sowie passende Kriterien und Fragestellungen erarbeitet:

- *Mustertreue*: Eine Architektur ist mustertreu, wenn die in der Soll-Architektur oder im verwendeten Architekturstil vorgegebenen Muster im Programmtext nicht verletzt werden.
- *Modularität*: Eine Architektur ist modular, wenn sie aus in sich zusammenhängenden, abgeschlossenen, ausgewogenen und lose gekoppelten Klassen und Subsystemen implementiert worden ist.
- *Geordnetheit*: Eine Architektur ist geordnet, wenn sie auf Klassen- und Subsystem-Ebene zyklenfrei ist.

Zum Faktor Geordnetheit habe ich in Kapitel 7 dreizehn Maße definiert und beispielhaft zwei von ihnen validiert, um sicher zu gehen, dass die Maße konform zur Messtheorie sind. Mit dem Modell für Architekturkomplexität war am Ende von Kapitel 7 auch die zweite Hälfte der ersten Fragestellung beantwortet, und ich konnte mich der empirischen Auswertung (Fragestellung 2) zuwenden.

Auf Basis des Komplexitätsmodells habe ich in Kapitel 8 die *Ergebnisse der Fallstudien* präsentiert. Anhand der Maße und Fragestellungen habe ich die Architekturkomplexität der verschiedenen Softwaresysteme in einer Momentaufnahme verglichen. Dabei habe ich nach den Ursachen für die sich jeweils ergebenden Antworten und Messwerte geforscht, um die zweite übergeordnete Fragestellung anzugehen. Die folgenden Ergebnisse konnte ich erzielen:

- Die jeweils *gewählten Architekturstile* haben großen Einfluss auf die Architekturkomplexität. Referenzarchitekturen wirken sich hauptsächlich auf der Klassen-Ebene, Subsystem- und Schichtenarchitekturen auf der Subsystem-Ebene aus.
- Jeder Architekturstil führt entweder auf Subsystem- oder auf Klassen-Ebene zu spezifischen *Mustern* im Programmtext, um die Abbildung zu unterstützen.
- Es mangelt sowohl auf Klassen- als auch auf Subsystem-Ebene an *Unterstützung*, um die Muster aus der Soll-Architektur und dem Architekturstil im Programmtext sichtbar zu machen.
- *Schnittstellen* auf Subsystem-Ebene werden in Schichten- und Referenzarchitekturen erst ab einer bestimmten Größe eingesetzt. Subsystemarchitekturen legen auf Schnittstellen besonderen Wert und führen sie von Beginn der Entwicklung an ein.
- *Geordnetheit* wird durch Referenzarchitekturen ausgehend von der Klassen-Ebene unterstützt und führt so zu weniger Klassenzyklen. Klassenzyklen haben die unangenehme Eigenschaft, dass sie in der Regel nur durch ein Refactoring der beteiligten Klassen aufgelöst werden können. Schichten- und Subsystemarchitekturen haben an diesem Punkt keine vergleichbaren Ergebnisse.

Insgesamt bleibt festzuhalten, dass Schichten- und Subsystemarchitekturen im Vergleich zu Referenzarchitekturen *schwache Architekturstile* sind. Ihre Regeln beziehen sich ausschließlich auf die Subsystem-Ebene und führen nicht dazu, dass das Entwicklungsteam auf der Klassen-Ebene eine gemeinsame Vorstellung der Architektur entwickelt. Auf dieser Ebene sind die Entwicklerinnen und Entwickler sich selbst überlassen, so dass Modularität und Geordnetheit nicht unterstützt werden. Der gleiche Effekt ist bei den Klassen in Referenzarchitekturen zu beobachten, die zu keiner der ausgewiesenen Elementarten gehören.

In Kapitel 9 habe ich architekturzentrierte Softwareentwicklung als Vorgehensweise beschreiben. Dieses grundlegende Vorgehen entstammt dem Ende der 1990er Jahre entwickelten Vorgehen RUP/UP. Aufbauend auf diesem Verständnis habe ich drei *Stadien der architekturzentrierten Softwareentwicklung* eingeführt:

- *Entwerfen* von Architektur: Ein Entwicklungsteam entwirft die Architektur im Rahmen einer Neuentwicklung in einem zykli-

schen Prozess mit Ausbaustufen auf der Basis eines Architekturstils.

- *Erhalten* von Architektur: Die vorhandene Architektur eines Softwaresystems wird bei einer Erweiterung erhalten, indem der Architekturstil überprüft, die Soll-Architektur auf der Grundlage der neuen Anforderungen überarbeitet und die Ist-Architektur entsprechend angepasst wird.
- *Erneuern* von Architektur: Die degenerierte Architektur eines Softwaresystems wird erneuert, indem der Architekturstil hinterfragt, die Soll-Architektur neu konzipiert und die Ist-Architektur mithilfe von Refactorings an die Soll-Architektur angeglichen wird.

Entwicklungsteams, die im Stadium Erhalten von Architektur bleiben und nicht in das Stadium Erneuern von Architektur abrutschen, setzen *Strategien* ein, um die Architekturkomplexität zu beherrschen. Aufgrund der Erfahrungen aus den Fallstudien habe ich in Kapitel 10 vier Strategien vorschlagen, die Entwicklungsteams in Kombination einsetzen sollten:

- *Schichtung*: Die Subsysteme eines Softwaresystems werden in Schichten angeordnet, zwischen denen lediglich Abwärtsreferenzen erlaubt sind.
- *Schnittstellenbildung*: Für die Subsysteme werden Schnittstellen definiert, die den Zugriff auf Klassen und enthaltene Subsysteme einschränken.
- *Zyklenfreiheit*: Auf Klassen- und auf Subsystem-Ebene wird Zyklenfreiheit angestrebt.
- *Referenzarchitektur*: Ein ausgefeilter Satz an Elementarten und Regeln wird als Vorlage für Modellierung und Implementierung verwendet.

Als Entscheidungsgrundlage bei der Frage, welche Strategie wann einsetzbar ist, habe ich in Kapitel 10 einen *Leitfaden* zusammengestellt. Die empfehlenswerteste Strategie für das Stadium Entwerfen von Architektur ist die Referenzarchitektur, die mit der Zeit um Schichtung und Schnittstellenbildung angereichert werden muss. Wurde ein Softwaresystem nicht von Beginn an mit einer Referenzarchitektur entworfen, so ist der Einsatz dieser Strategie im Stadium Erhalten oder Erneuern von Architektur kaum mehr möglich. Das Einführen der feingranularen Regeln auf Klassen-Ebene verursacht einen zu hohen Refactoring-Aufwand. Um in diesem Fall Architekturkomplexität zu reduzieren, sollte zuerst mit der Strategie Zyklenfreiheit gearbeitet und anschließend Schnittstellenbildung eingeführt werden.

Mit Kapitel 10 habe ich schließlich die letzte Fragestellung dieser Arbeit nach Strategien zur Reduktion von Architekturkomplexität und Kriterien für ihre Auswahl beantwortet. Abschließend hinterfrage ich die erreichten Erkenntnis-

se und gebe einen Ausblick auf weitere Themen, bei denen aus Sicht dieser Arbeit Forschungsbedarf besteht.

Den Schlüssen, die ich in dieser Arbeit ziehe, und den Vorschlägen, die ich mache, liegt eine Forschung von mehreren Jahren zu Grunde, bei der ich eine große Anzahl von Fallstudien durchgeführt habe. Trotzdem muss bedacht werden, dass jede Fallstudie ein einzelnes Softwaresystem beinhaltet, das als Unikat für den jeweiligen Anwendungskontext entwickelt wurde. Vielfältige Einflüsse aus unterschiedlichsten Quellen haben dazu geführt, dass die jeweiligen Softwaresysteme hohe oder geringe Architekturkomplexität haben. Aus diesem Grund kann ich nicht behaupten, dass meine Ergebnisse die Architekturkomplexität der einzelnen Softwaresysteme exakt bestimmen und dass die sie entwickelnden Teams entsprechend scheitern oder weiterhin zurechtkommen werden. Eine alles einschließende Theorie darüber, was Komplexität bei der Softwareentwicklung ausmacht und wie sie insgesamt beherrscht werden kann, halte ich, aufgrund der großen Menge an Einflussfaktoren, nicht für entwickelbar. Allerdings hoffe ich, dass diese Arbeit einen weiteren Baustein zu der Frage beigetragen hat, ob und warum Software komplex ist.

Einige Fragen konnten in dieser Arbeit nicht bearbeitet werden:

- *Maße im Komplexitätsmodell*: In dieser Arbeit habe ich Maße zum Faktor Geordnetheit definiert und zum Teil validiert. Zu den Kriterien Zusammenhalt und Abhängigkeit des Faktors Modularität sind in der Literatur eine große Anzahl an Maßen vorhanden. Welche dieser Maße sollen ins Komplexitätsmodell aufgenommen werden? Lassen sich auch für die anderen Kriterien der Modularität und für den Faktor Mustertreue Maße entwickeln?
- *Architekturkomplexität in der Zeit*: Die für diese Arbeit durchgeführten Fallstudien lieferten Momentaufnahmen von Architekturkomplexität. Softwaresysteme, die eingesetzt werden, werden kontinuierlich weiterentwickelt und ausgebaut. Dieser Prozess sollte dahingehend untersucht werden, wie sich Architekturkomplexität in der Zeit entwickelt und wie es zu dem Übergang aus dem Stadium Erhalten von Architektur zum Erneuern kommt.
- *Softwaresysteme ohne Architektur*: In dieser Arbeit habe ich mich auf Fallstudien konzentriert, bei denen ein Architekturstil eingesetzt wurde oder zumindest eine Soll-Architektur vorhanden war. Softwaresysteme ohne Architekturstil und Soll-Architektur müssten sich mit den Faktoren Modularität und Geordnetheit ebenfalls untersuchen lassen. Der Faktor Mustertreue ist bei diesen Softwaresystemen nicht verwendbar, weil in der Ist-Architektur keine Muster aus Soll-Architektur oder Architekturstil befolgt werden müssen.

- *Problem-inhärente Komplexität*: Abhängig von der Komplexität des Gegenstandsbereichs und dem Einsatzkontext hat eine Softwarearchitektur problem-inhärente Komplexität. Lässt sich diese Komplexität auf ähnliche Weise analysieren und bewerten?
- *Projekteigene Referenzarchitekturen*: In den Fallstudien konnte ich vier projekteigene Referenzarchitekturen beobachten. Sie zeigen die oft erfolgreichen Versuche von Projektteams, mit verbindlichen Regeln auf der Klassen-Ebene den Designraum einzuschränken. Was sind die Auslöser dafür, dass ein Projektteam diese Regeln einführt? Wie und woraus entwickeln sich projekteigene Referenzarchitekturen?
- *Unterstützung für Mustertreue*: Der Faktor Mustertreue untersucht, wie gut sich die Soll-Architektur und der Architekturstil auf die Ist-Architektur abbilden lassen. Diese Abbildung kann in objektorientierten Programmiersprachen nur durch Konventionen wie den Aufbau des Package-Baums oder Namenszusätzen erleichtert werden. Wie können objektorientierte Programmiersprachen erweitert werden, um Mustertreue zu unterstützen?
- *Technische Komplexität*: In der Softwareentwicklung wird heute durch die Entstehung des Worldwide Web eine extreme Technologievielfalt eingesetzt. Nur in ausgewählten Bereichen ist es Entwicklerinnen und Entwicklern möglich, Experten zu sein. Wie kann diese technische Komplexität beherrscht werden? Welche Zusammenhänge oder Produkte führen zu viel und welche zu wenig technischer Komplexität?
- *Komplexität des Entwicklungsprozesses*: Softwareentwicklung findet heute in keinem Projekt mehr auf einer Insel statt, sondern ist integriert in eine unternehmensweite, in manchen Fällen unternehmensübergreifende Gesamtarchitektur aus verteilten Services. Die Abhängigkeiten zwischen den Projekten machen ein umfangreiches Management für die verzahnte Auslieferung von Softwarekomponenten notwendig. Kommt ein Glied in der Kette zu spät, so wird der gesamte Prozess durcheinander gebracht. Wie kann diese organisatorische Komplexität analysiert und beherrscht werden?

Diese sicherlich unvollständige Aufzählung zeigt, dass mit der vorliegenden Arbeit Ergebnisse erzielt worden sind, aus denen sich eine Reihe weiterer interessanter Fragestellungen ergeben. Die Themengebiete Komplexität von Softwarearchitektur, Architekturstile und architekturzentrierte Softwareentwicklung werden in Zukunft noch in vielen spannenden Forschungsvorhaben untersucht werden.

A Material zu den Fallstudien

Tabelle A-1: Größe, Alter, Vorgehen und Architekturstile der Fallstudien

Nr	LOC	RLOC	Start	A[29]	I[29]	Architekturstil
1	26.051	10.679	2005	X		**WAM-Architektur** Schichtenarchitektur ohne Schnittstellen
2	33.935	13.219	2000	X	X	**WAM-Architektur**
3	36.838	16.314	2003	X	X	**Projekteigene Referenzarchitektur**
4	64.156	21.591	2003	X	X	**WAM-Architektur** Erweiterte Schichtenarchitektur ohne S.
5	53.087	22.957	2001	X		**WAM-Architektur**
6	47.167	23.478	2002	X	X	Schichtenarchitektur ohne Schnittstellen
7	108.057	33.651	2001	X	X	**WAM-Architektur**
8	83.709	36.671	2003	X		*Subsystemarchitektur*
9	99.405	43.969	2003	X		**WAM-Architektur**
10	137.418	49.428	1999	X	X	**WAM-Architektur**
11	121.137	65.284	2003	X		Schichtenarchitektur ohne Schnittstellen
12	130.316	70.266	2001	X		**Projekteigene Referenzarchitektur**
13	169.567	74.344	2003	X		Schichtenarchitektur ohne Schnittstellen
14	188.113	93.593	2003	X		Schichtenarchitektur ohne Schnittstellen
15	221.284	112.675	2001	X		Schichtenarchitektur ohne Schnittstellen
16	238.637	120.964	2003	X		Schichtenarchitektur ohne Schnittstellen
17	269.609	133.095	2002	X		**Projekteigene Referenzarchitektur** Schichtenarchitektur ohne Schnittstellen
18	294.414	143.101	2000	X		Schichtenarchitektur ohne Schnittstellen
19	~500.000	~250.000	2002		X	Schichtenarchitektur mit Schnittstellen
20	789.938	344.911	2001	X	X	*Subsystemarchitektur*
21	1465.441	629.674	2003	X	X	**Quasar-Architektur** *Subsystemarchitektur*
22	2.919.550	1.296.455	1998	X	X	**Projekteigene Referenzarchitektur** Erweiterte Schichtenarchitektur mit S.
23	2.869.566	1.384.260	2001	X		Schichtenarchitektur mit Schnittstellen
24	~14 Mio	~8 Mio	2002		X	Erweiterte Schichtenarchitektur mit S.

Legende zu Tabelle A-1:

Referenzarchitektur
Schichtenarchitektur
Subsystemarchitektur

[29] A – Architekturanalyse, I – Interview

Tabelle A-2: Stadium der Fallstudien

Nr	LOC	RLOC	Start	A[30]	I[29]	Architekturstil
1	26.051	10.679	2005	X		**WAM-Architektur** Schichtenarchitektur ohne Schnittstellen
2	33.935	13.219	2000	X	X	**WAM-Architektur**
3	36.838	16.314	2003	X	X	**Projekteigene Referenzarchitektur**
4	64.156	21.591	2003	X	X	**WAM-Architektur** Erweiterte Schichtenarchitektur ohne S.
5	53.087	22.957	2001	X		**WAM-Architektur**
6	47.167	23.478	2002	X	X	Schichtenarchitektur ohne Schnittstellen
7	108.057	33.651	2001	X	X	**WAM-Architektur**
8	83.709	36.671	2003	X		*Subsystemarchitektur*
9	99.405	43.969	2003	X		**WAM-Architektur**
10	137.418	49.428	1999	X	X	**WAM-Architektur**
11	121.137	65.284	2003	X		Schichtenarchitektur ohne Schnittstellen
12	130.316	70.266	2001	X		**Projekteigene Referenzarchitektur**
13	169.567	74.344	2003	X		Schichtenarchitektur ohne Schnittstellen
14	188.113	93.593	2003	X		Schichtenarchitektur ohne Schnittstellen
15	221.284	112.675	2001	X		Schichtenarchitektur ohne Schnittstellen
16	238.637	120.964	2003	X		Schichtenarchitektur ohne Schnittstellen
17	269.609	133.095	2002	X		**Projekteigene Referenzarchitektur** Schichtenarchitektur ohne Schnittstellen
18	294.414	143.101	2000	X		Schichtenarchitektur ohne Schnittstellen
19	~500.000	~250.000	2002		X	Schichtenarchitektur mit Schnittstellen
20	789.938	344.911	2001	X	X	*Subsystemarchitektur*
21	1465.441	629.674	2003	X	X	**Quasar-Architektur** *Subsystemarchitektur*
22	2.919.550	1.296.455	1998	X	X	**Projekteigene Referenzarchitektur** Erweiterte Schichtenarchitektur mit S.
23	2.869.566	1.384.260	2001	X		Schichtenarchitektur mit Schnittstellen
24	~14 Mio	~8 Mio	2002		X	Erweiterte Schichtenarchitektur mit S.

Legende zu Tabelle A-2:

Entwerfen der Architektur
Erhalten der Architektur
Erneuern der Architektur

[30] A – Architekturanalyse, I – Interview

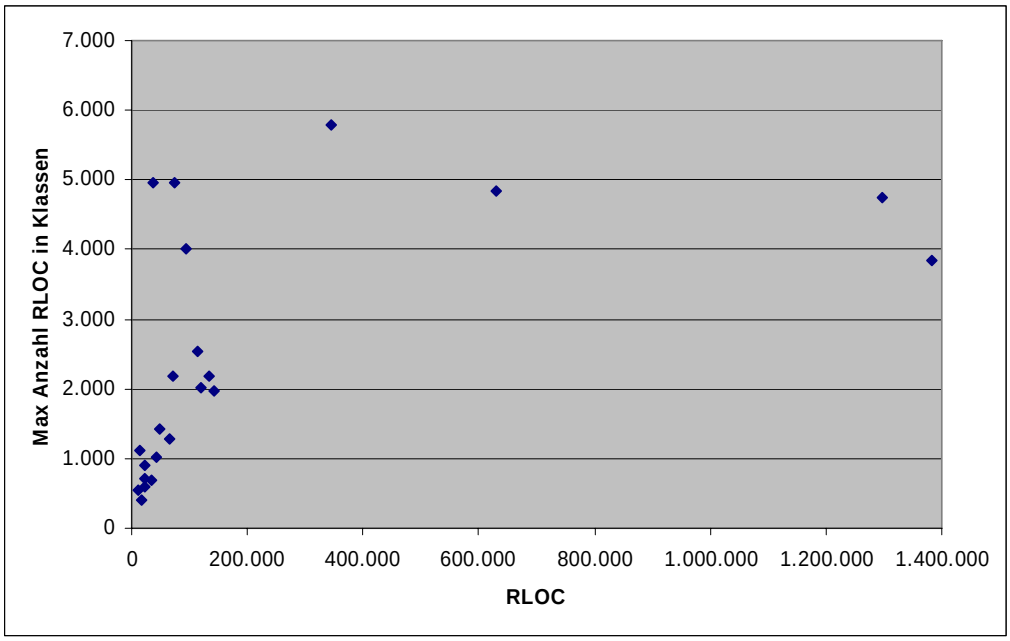

Abbildung A-1: Maximale Anzahl Programmzeilen in Klassen

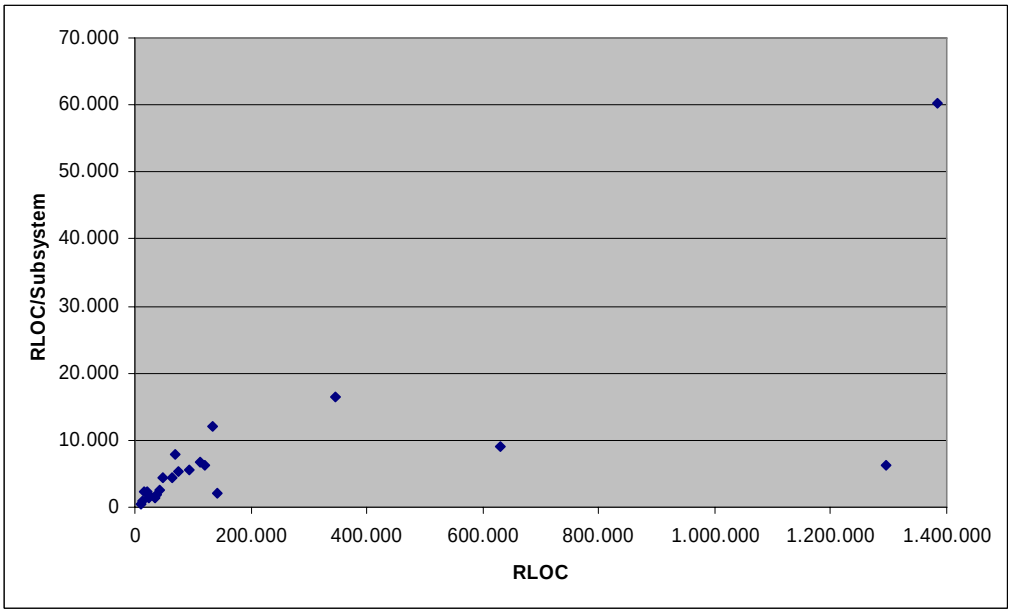

Abbildung A-2: RLOC pro Subsystem

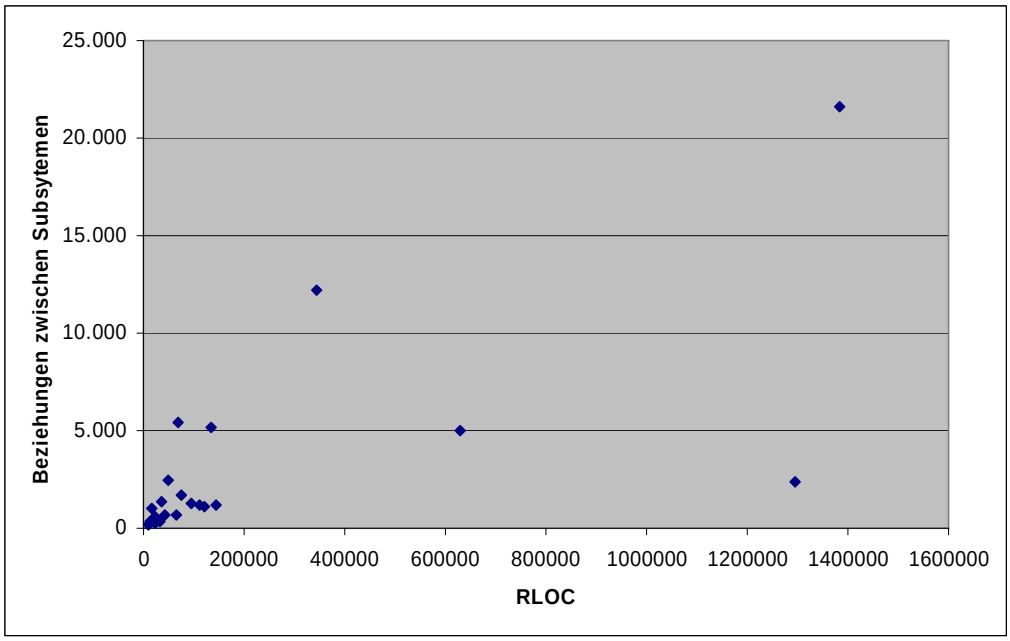

Abbildung A-3: Beziehungen zwischen Subsystemen

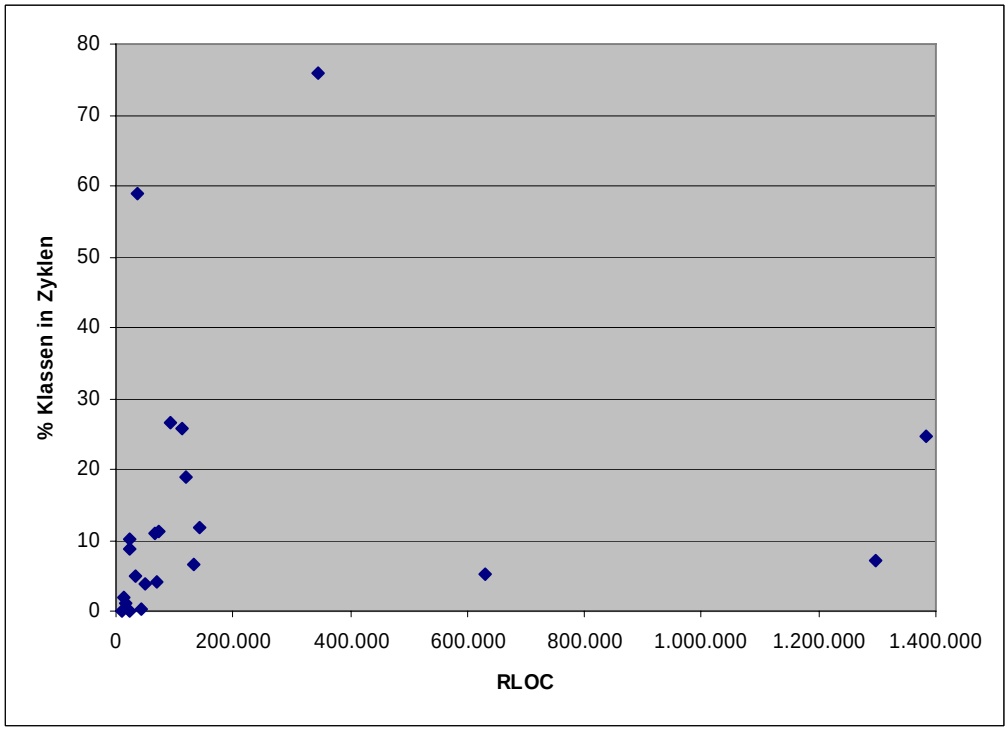

Abbildung A-4: RLOC und Zyklenausmaß auf der Klassen-Ebene

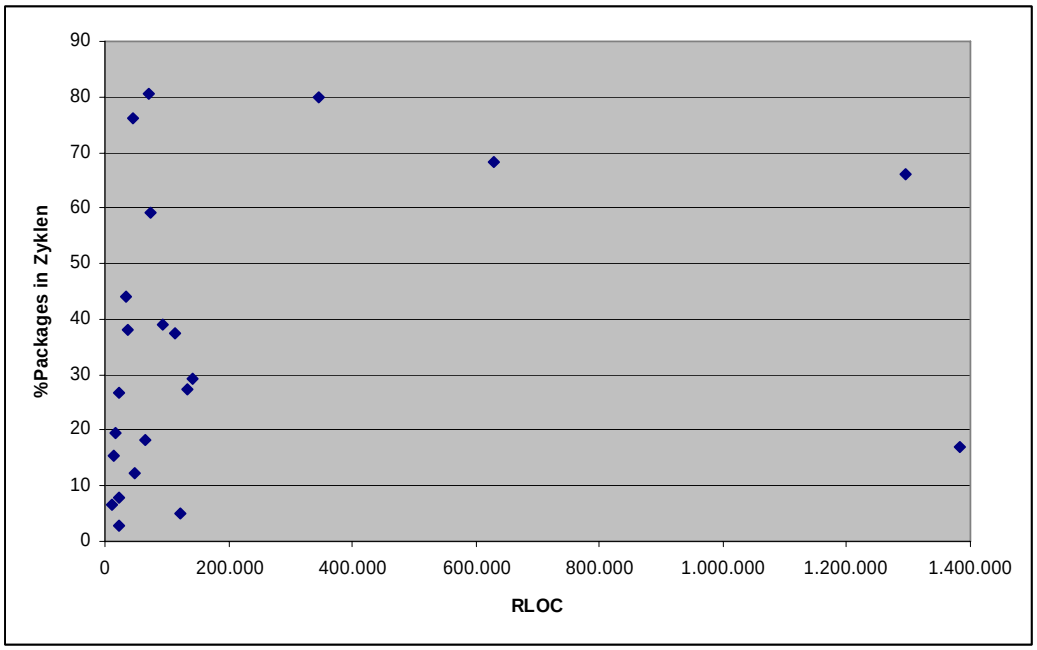

Abbildung A-5: RLOC und Zyklenausmaß auf Package-Ebene

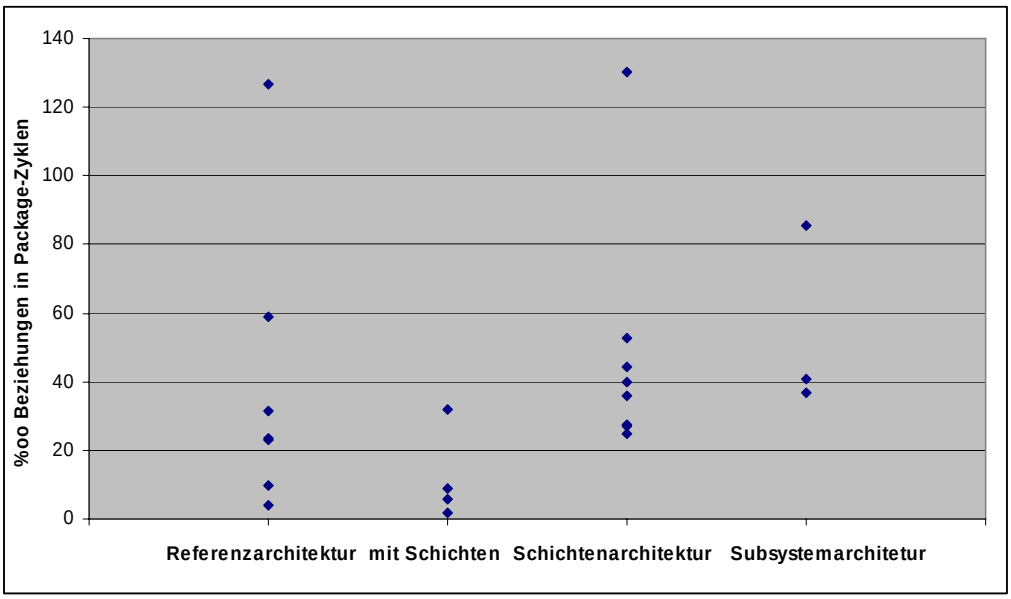

Abbildung A-6: Beziehungen in Package-Zyklen

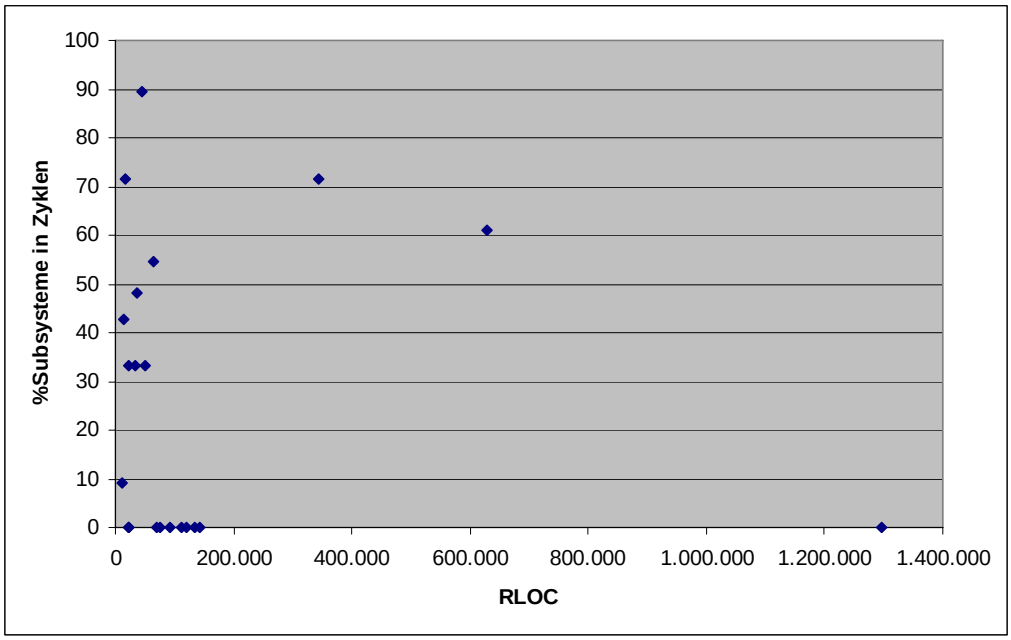

Abbildung A-7: RLOC und Zyklenausmaß auf Subsystem-Ebene

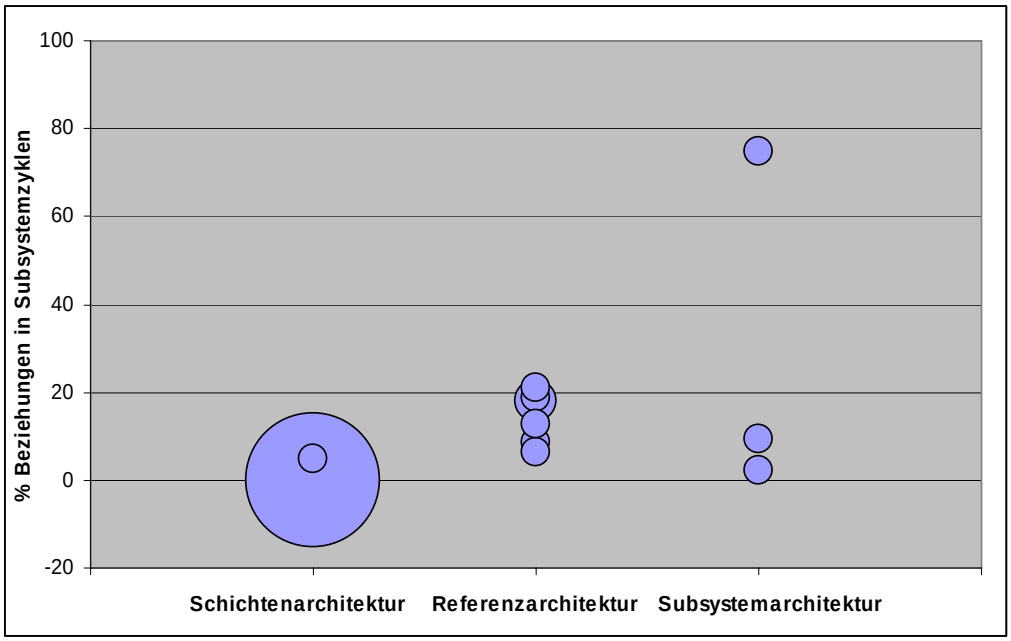

Abbildung A-8: Beziehungen in Subsystemzyklen

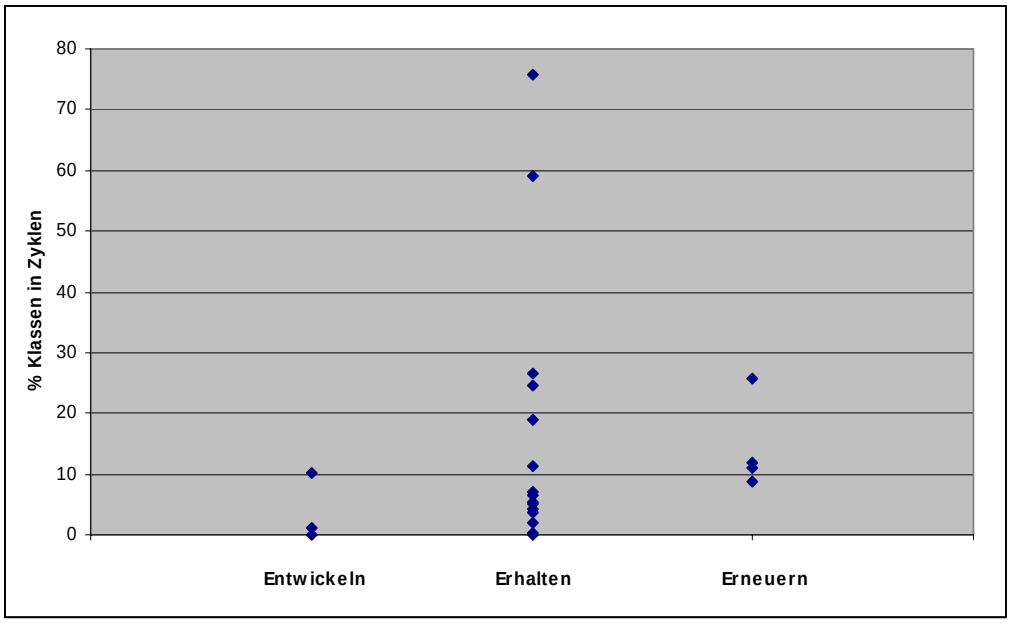

Abbildung A-9: Stadium und Zyklenausmaß auf der Klassen-Ebene

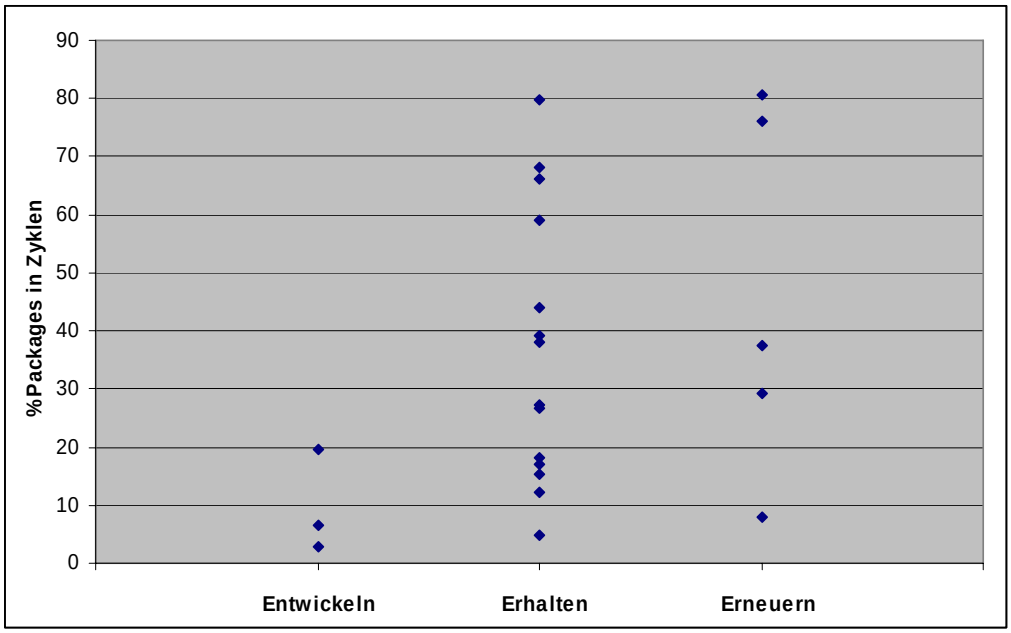

Abbildung A-10: Stadium und Zyklenausmaß auf der Package-Ebene

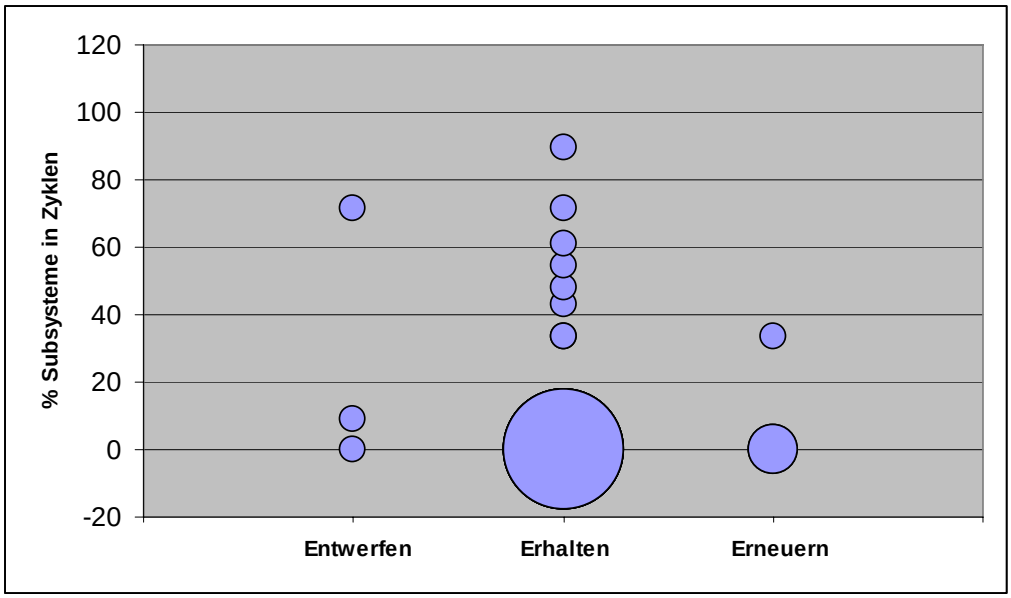

Abbildung A-11: Stadium und Zyklenausmaß auf der Subsystem-Ebene

B Analyseberichte

Im Anschluss an die einzelnen Fallstudien habe ich jeweils einen Analysebericht geschrieben (s. Abschnitt 5.3.1). Im Folgenden sind die Berichte der Fallstudien 3, 11, 20 und 22 angefügt.

B.1 Fallstudie 3

Fallstudie 3 wurde am JCommSy[1] durchgeführt. Das JCommSy ist die in Java-programmierte Variante des seit 1999 an der Universität Hamburg entwickelten, gepflegten und in Lehrveranstaltungen eingesetzte sowie für andere Universitäten und Organisationen bereitgestellte CommSy. CommSy ist eine in PHP-implementierte Kommunikationsplattform, die es ihren Mitgliedern erlaubt, über gemeinsam genutzte virtuelle Räume Informationen auszutauschen, Diskussionen zu führen und dadurch vernetzte Projektarbeit unterstützt.

In jedem Raum des CommSys werden den Benutzerinnen und Benutzern verschiedene Rubriken angeboten, z.B.: Termine, Material, Diskussion, Neuigkeiten, Personen. Diese Rubriken werden für jeden Raum in einer Übersicht dargestellt und können von den Anwenderinnen und Anwendern auf einzelnen Seiten bearbeitet werden. Der Inhalt der verschiedenen Rubriken kann über Links miteinander verknüpft werden, so dass z.B. einem Termin bestimmte Materialien zugeordnet werden können.

Im Herbst 2004 begann im Rahmen einer Lehrveranstaltung ein studentisches Forschungsprojekt mit einer Vorstudie zur Migration des PHP-CommSys nach Java. Das Projekt hatte 20 Studierende und wurde von einem wissenschaftlichen Mitarbeiter geleitet. Ziel des Projekts war es, Erfahrung mit dem Thema Migration zu sammeln. Das CommSy bot sich für diese Untersuchung an, weil es einerseits in der Universität entwickelt wird und andererseits im praktischen Einsatz ist.

Für eine Java-Variante des JCommSy muss besonders die vorhandene verarbeitende Logik ersetzt werden. Die Datenbank des CommSy soll im JCommSy weiter verwendet werden. Die Web-Oberfläche des CommSy soll von ihrer Gestaltung und Handhabung im JCommSy gleich bleiben.

Seit Sommer 2005 hat sich das gesamte CommSy-Team dazu entschlossen, das CommSy vollständig nach Java zu migrieren. Allerdings haben die PHP-Entwickler keine Kapazität, um am JCommSy mitzuentwickeln. Die Migration des CommSys nach Java wird daher von anderen wissenschaftlichen Mitarbeitern und einigen studentischen Hilfskräften durchgeführt. Das erste Ziel ist die Migration der Termin-Rubrik.

[1] www.jcommsy.net

B.1.1 Ablauf der Analyse

Ich habe das JCommSy-System in 2005 drei Mal untersucht und schließlich noch einmal im Dezember 2006. Die Untersuchungszeitpunkte waren: April 2005, Juli 2005, September 2005 und Dezember 2006. Die Ergebnisse der letzten Untersuchung wurden in dieser Arbeit verwendet.

Da das JCommSy eine Web-Anwendung ist, besteht es neben Sourcecode, der in Java programmiert ist, aus JSPs für die Darstellung der Seiteninhalte und einer Reihe von TAG-Libs zur Vereinfachung der JSPs. Der Sotograph ist nicht in der Lage TAG-Libs oder JSPs einzulesen, so dass wir ausschließlich die in Java implementierten Anteile des JCommSy analysiert haben.

Bei der ersten Analyse habe ich JCommSy an einem halben Tag gemeinsam mit dem leitenden wissenschaftlichen Mitarbeiter untersucht. Da das JCommSy noch relativ klein ist, waren die Definition von Subsystemen einfach und die typischen Probleme von älteren Softwaresystemen (Problem-Artefakte, die eine Vielzahl von schlechten Werten verursachen) noch nicht eingetreten. Wir haben Architekturverletzungen, Zyklen und Metriken sowie Regeln auf der Klassen-Ebene untersucht. Dabei wurde deutlich, dass im JCommSy-Team eine projekteigene Referenzarchitektur eingesetzt wird. Die Ergebnisse dieser Untersuchung haben wir auf Folien den PHP-CommSy-Entwicklern präsentiert, um sie an der Diskussion über die Architektur des JCommSys zu beteiligen.

Die zweite Analyse haben ich wiederum an einem halben Tag vorgenommen und die Ergebnisse den JCommSy- und den CommSy-Entwicklern in einer gemeinsamen Sitzung präsentiert. Im Laufe dieser zweiten Analyse haben wir festgestellt, dass die Package-Struktur des Softwaresystems dringend überarbeitet werden muss.

Kurz nach dieser Sitzung haben die Entwickler die Package-Struktur des Softwaresystems umgestellt, so dass wir bei der dritten Analyse eine komplett andere Package- und Subsystem-Struktur vorgefunden haben. Die abschließende vierte Analyse habe ich allein durchgeführt und die Ergebnisse hinterher mit den zuständigen wissenschaftlichen Mitarbeitern besprochen.

B.1.2 Entwicklungssituation

Die Entwicklung des JCommSys wurde in einem studentischen Projekts begonnen. Die Studierenden sollten in diesem Projekt verschiedene Aspekte der Softwareentwicklung kennen lernen und vertiefen. Die JCommSy-Entwicklung wurde also nicht von fertig ausgebildeten Entwicklerinnen und Entwicklern durchgeführt. Darüber hinaus standen die Studierenden nicht Vollzeit für die Entwicklung zur Verfügung, sondern nur ein bis zwei Tage pro Woche.

Im Juli 2005 war das studentische Projekt zu Ende und die Entwicklung wird von bezahlten studentischen Hilfskräften und zwei wissenschaftlichen Mitarbeitern weitergeführt. Die studentischen Hilfskräfte dürfen maximal 20

Stunden in der Woche neben ihrem Studium arbeiten. Die wissenschaftlichen Mitarbeiter müssen im Semester Lehre durchführen und diese in der lehrefreien Zeit vorbereiten. Durch die Situation im studentischen Projekt und bei der Weiterentwicklung geht die Programmierung des JCommSys im Vergleich zu Systementwicklungen in der Wirtschaft etwas langsamer voran.

B.1.3 Ergebnisse der Vermessung

Für JCommSy habe ich bei den vier Analysen die folgenden Ergebnisse gemessen.

Analyse	LOC	RLOC	Klassen	Pckg	Sub.	KZ[2]	PZ[2]	SZ[2]
April 2005	6.900	3.200	53	15	-	-	6	-
Juli 2005	11.650	5.145	102	26	3	2	14	3
Sep. 2005	15.028	7.192	98	13	7	2	0	0
Dez. 2006	36.838	16.314	398	51	7	4	10	5

Beim ersten Termin haben wir das JCommSy aufgrund der geringen Größe des Softwaresystems nicht in Subsysteme aufgeteilt.

Auf Ebene der Packages gab es in den ersten beiden Versionen jeweils einen großen Zyklus. Die erste Version des JCommSy hatte einen Zyklus, an dem sechs der insgesamt fünfzehn Packages beteiligt waren. In der zweiten Version waren von den insgesamt 26 Packages 14 in einem Zyklus. Die sechs am Zyklus beteiligten Packages aus der ersten Version waren in dem größeren Zyklus der zweiten Version enthalten. Der Zyklus aus der ersten JCommSy-Version war in der Zwischenzeit also gewachsen. Nach der Umstrukturierung der Packages waren in der dritten Version keine Package-Zyklen mehr zu finden. In der letzten vierten Analyse waren wieder neue Package-Zyklen zu vorhanden.

Bei der ersten Version von JCommSy hatten wir keine Subsysteme gebildet und daher auch keine Subsystem-Zyklen untersucht. In der zweiten Version des JCommSy bilden die drei definierten Subsysteme einen Subsystem-Zyklus. Nach einer entsprechenden Restrukturierung weist die dritte Version keine Subsystem-Zyklen mehr auf. In der vierten Analyse waren schließlich wieder neue Subsystem-Zyklen entstanden.

Die Größen der Methoden, Klassen, Packages und Subsysteme waren bei allen Versionen unauffällig.

[2] KZ = Klassen in Zyklen, PZ = Packages in Zyklen, SZ = Subsysteme in Zyklen

B.1.4 Bewertung auf der Subsystem-Ebene

Da das JCommSy ein junges und kleines Softwaresystem ist, ist die Subsystem-Ebene noch nicht besonders ausgeprägt: In der ersten Version gab es noch gar keine Subsysteme, in der zweiten Version drei Subsysteme und in der dritten und vierten Version sieben Subsysteme und vier Schichten. Alle Subsysteme bestanden aus relativ wenigen Packages.

In der vierten Version war eine erste Schnittstelle zwischen Client und Server vorhanden. Der Package-Baum entspricht in seinem Aufbau den Kategorien der projekteigenen Referenzarchitektur und enthält außerdem fachliche Packages.

Durch den Vergleich von vier Versionen konnte, die Entstehung und Entwicklung von Architekturkomplexität auf der Subsystem-Ebene für den Faktor Geordnetheit beobachtet werden.

In diesem Softwaresystem entstand Komplexität in der ersten und zweiten Version durch die Aufteilung der Klassen auf Packages. Einen Teil des Package-Zyklus in der ersten Version konnten wir durch Umstrukturierungen von Klassen aufheben. Ein Teil des Zyklus wurde aber durch die gewählte Package-Struktur verursacht. Die Entwickler des JCommSy hatten sich entschieden, ihren Package-Baum an den Rubriken des CommSys zu orientieren, also eine anwendungsfachliche Strukturierung vorzunehmen (s. Abbildung B-1).

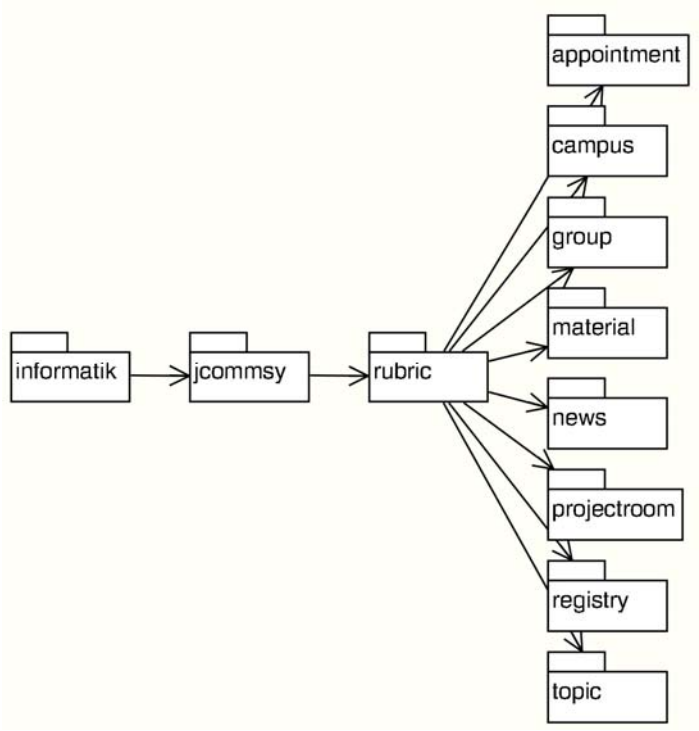

Abbildung B-1: Anwendungsfachlicher Package-Baum

Das Ergebnis war ein hoher Zusammenhalt (Faktor Modularität) in den Packages, weil alle Klassen einer Rubrik gemeinsam abgelegt sind. Diese

Struktur führte aber zu einem großen Package-Zyklus. Die einzelnen Seiten der Rubriken sind über Hyperlinks mit Seiten aus anderen Rubriken verbunden und enthalten zum Teil Listen von Einträgen aus anderen Rubriken. Die Servlets, die die Seiten aufbauen, müssen sich deshalb den Inhalt für die Seiten aus den verschiedenen Rubriken besorgen, so dass die Klassen in dem Package jeder Rubrik Klassen aus allen anderen Packages benötigen. So entsteht durch die Aufteilung der Klassen in Rubrik-Packages ein Zyklus.

Bei der ersten Analyse, als lediglich zwei Rubriken implementiert waren, wurde dieser Widerspruch zwischen Zusammenhalt und Geordnetheit sichtbar und wir haben ihn lange diskutiert. Bei der zweiten Sitzung, bei der alle Rubriken abdeckt waren, war die Anzahl der Packages in dem Package-Zyklus von sechs auf vierzehn gestiegen. Das Entwicklerteam hat sich an diesem Punkt zugunsten der Geordnetheit für eine Restrukturierung entschieden, so dass in der dritten Version keine Package-Zyklen mehr vorhanden waren. Das Ergebnis war eine Struktur des Package-Baums, die an den Elementarten der projekteigenen Referenzarchitektur des JCommSy orientiert war (s. Abbildung B-2).

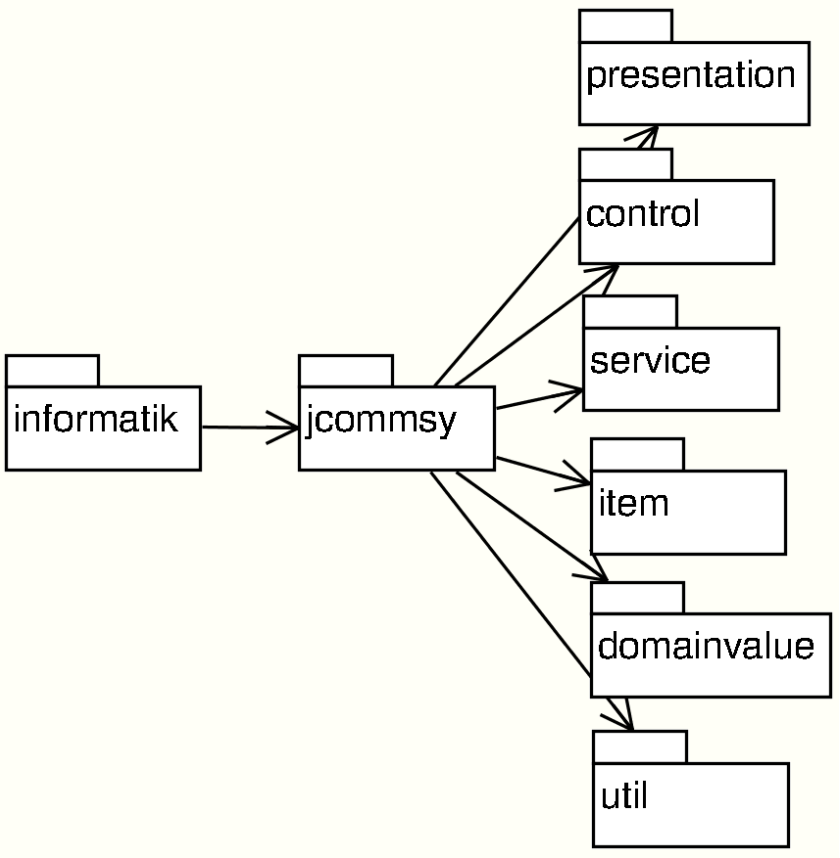

Abbildung B-2: An der Referenzarchitektur orientierter Package-Baum

In der vierten Version war wieder ein Package-Zyklus vorhanden (s. Abbildung B-3). Allerdings ist dieser Zyklus lokal im Package-Baum des Subsystems `Control` angesiedelt. Das oberste Package dieses Subsystems (`jcommsy.control`) verwendet seine direkten Subpackages (`common`,

`announcement` und `action`) und diese haben wiederum Beziehungen zu dem übergeordneten Package (`jcommsy.control`).

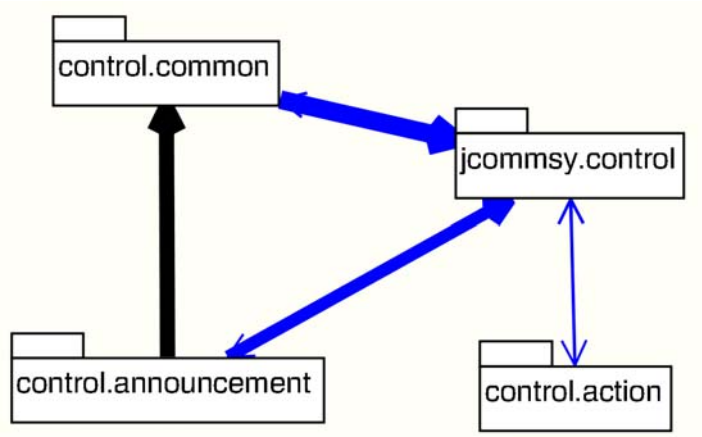

Abbildung B-3: Ursache des Package-Zyklus im JCommSy

Eine Restrukturierung innerhalb dieses Subsystems konnte den lokalen Package-Zyklus auflösen.

B.1.5 Bewertung auf der Klassen-Ebene

Auf der Klassen-Ebene hat das JCommSy geringe Architekturkomplexität. Es gibt fast keine Klassenzyklen und Größe und Abhängigkeiten von Klassen und Packages, sind nicht auffällig. Um das zu erreichen, wird im JCommSy-Team eine projekteigene Referenzarchitektur eingesetzt. Die Klassen lassen sich zum einen über Namenskonventionen und zum größten Teil auch über Strukturkonventionen den Elementarten zuordnen (s. Abbildung B-2).

B.1.6 Erkenntnisse aus der Untersuchung

An diesem Softwaresystem sind mir die folgenden Punkte deutlich geworden:

- In diesem Projekt wird eine eigene Referenzarchitektur eingesetzt, die ähnlich wie WAM und Quasar ein ausgefeiltes Repertoire an Elementarten und Beziehungsregeln hat. Mithilfe dieser projekteigenen Referenzarchitektur wird die Architekturkomplexität auf der Klassen-Ebene vermindert.
- Ein Aufbau des Package-Baums nach fachlichen Subsystemen führt in diesem Softwaresystem zu Zyklen. Wird die erste Ebene der Package-Knoten anhand der Elementarten der Referenzarchitektur gebildet und darunter nach fachlichen Gesichtspunkten, so werden Package-Zyklen verhindert.
- Eine Analyse mehrerer Versionen zeigt, wie Architekturkomplexität auf Klassen- und Subsystem-Ebene entsteht bzw. wieder aufgelöst wird.

B.2 Fallstudie 11

Das Unternehmen, bei dem Fallstudie 11 durchgeführt werden konnte, ist Outsourcing-Dienstleister und übernimmt für seine Auftraggeber ganzheitliche Aufgaben in Vertrieb und Kundenservice. Die Leistungsbreite reicht von vorgelagerten Aufgaben wie Planung und Umsetzung von Businessprocess-mangement-Konzepten über das flexibel Kontaktrouting mit Servicerufnummern sowie die aktive Kundenkommunikation via Communication Centern bis zu nachgelagerten Administrationsabläufen wie Fulfillment und Inkasso.

Das Softwaresystem ist in Java mit Swing-Oberfläche und EJBs für die Serverfunktionalität programmiert. Bei der Entwicklung wird Eclipse eingesetzt. Mehrere hundert Agenten in verschiedenen Call Centern und ihre vorgesetzten Manager arbeiten mit diesem Softwaresystem, um In- und Outbound-Kampagnen durchzuführen. Kampagnen werden als Inbound-Kampagnen bezeichnet, wenn der Anruf durch den Kunden erfolgt. Eine Outbound-Kampagne wird durch den Agent initiiert. Für unterschiedliche Mandanten werden mithilfe des Softwaresystems Hotlines für Kundenanfragen, Bestellannahmen und andere kundennahe Prozesse durchgeführt. Im Outbound-Geschäft tätigen die Agenten der Call Center mit der Software Verkaufsgespräche oder Kundenbefragungen. Bei seiner Arbeit füllt der Agent Masken aus, in denen er den Verlauf des Kundengesprächs vermerkt und festhält, ob mit dem Kunden zu einem späteren Zeitpunkt erneut Kontakt aufgenommen werden soll.

Das Softwaresystem ist von Entwicklern gebaut worden, die die Firma inzwischen alle verlassen haben. Sechs Entwickler sind für das Softwaresystem oder bestimmte Teilsysteme zuständig und passen es an neue Anforderungen der Kunden an. Einer dieser Entwickler hat über die Weiterentwicklung und Fehlerbehebung hinaus die Aufgabe, die Qualität des Softwaresystems zu erhöhen, indem er Teilsysteme erneuert.

B.2.1 Ablauf der Analyse

Während der zweitägigen Analyse stand mir die ganze Zeit der zuständige Entwickler zur Verfügung. Darüber hinaus hatten wir engen Kontakt zum aktuellen Leiter der Entwicklungsabteilung und zu seinem Nachfolger.

Zu Beginn hat mir der Architekt das Softwaresystem vorgestellt und grob seinen Funktionsumfang erklärt. Daraufhin haben wir das Softwaresystem in den Sotographen eingelesen und mit der Analyse begonnen. Nachdem wir die Größe und den Dokumentationsgrad des Softwaresystems ermittelt hatten, haben wir uns den größten Teil des ersten Tages mit der Analyse von Klassen- und Package-Zyklen befasst.

Am Ende des ersten Tages haben wir ein Subsystem- und ein Schichtenmodell für das Softwaresystem erstellt und dabei seine Subsystem-Struktur untersucht. Am nächsten Tag haben wir die verschiedene Metriken auf der Ebene

der Subsysteme und Schichten ausgewertet und gegebenenfalls das Subsystem- oder Schichtenmodell angepasst.

Auf der Klassen-Ebene haben wir Metriken und Regeln überprüft und auffällige Befunde (große Klassen, hohe zyklomatische Komplexität von Operationen etc.) in einem Dokument festgehalten.

Schließlich haben wir eine Trendanalyse durchgeführt, bei der wir den Teil des Softwaresystems, für den es bereits eine überarbeitete Version gibt, mit dieser überarbeiteten Version verglichen haben. Die Messwerte der überarbeiteten Version waren ähnlich zu der bisher eingesetzten Variante. Die Komplexität des Softwaresystems konnte bisher also nicht verringert werden.

B.2.2 Entwicklungssituation

Der für das Softwaresystem zuständige Entwickler hat das Softwaresystem nicht selbst mit entwickelt. Er muss daher Änderungen und Erweiterungen an einem vorhandenen Softwaresystem vornehmen, dessen ursprüngliche Designidee ihm nicht bekannt ist. Da das Softwaresystem nicht ausreichend mit Testklassen abgesichert ist, sind Veränderungen und Erweiterungen nur durch Tests über die Benutzungsschnittstelle zu überprüfen. Die mangelnde Testabdeckung macht es dem Entwickler schwer, schnelle kurze Änderungen durchzuführen, die die Qualität des Softwaresystems verbessern würden.

Die Klassen, die in der Analyse besonders auffällig waren, waren dem Entwickler schon als Problemfälle bekannt. Er hatte auch schon einige Verbesserungen an diesen Klassen vorgenommen, was wir bei der Trendanalyse feststellen konnten. Neben dem Wissen, um die zentralen Problemklassen des Softwaresystems gab es aber auch Punkte, die dem Entwickler nicht klar waren.

Zum einen hatte der Mitarbeiter bei der Definition der Subsysteme die Vermutung, dass er ein Subsystem mit einer Schnittstelle und zwei verschiedenen Implementierungen dieser Schnittstelle im Softwaresystem hätte. Als wir diese Schnittstelle untersucht haben, mussten wir feststellen, dass die beiden Implementierungen eigene Schnittstellen haben. Die eine Schnittstelle war in einem eigenen Package abgelegt, die Schnittstellen-Klassen der anderen Implementierungsvariante waren auf die einzelnen Packages des Subsystems verteilt.

Zu anderen machte eine Diskussion, bei der wir uns einen im Softwaresystem vorhandenen Klassenzyklus genauer angesehen haben, deutlich, welche Designvorstellungen der Entwickler selbst hat. Als wir versuchten, die Ursache für den Zyklus zu finden und den Zyklus aufzulösen, sagte der Entwickler: „Wenn ich diese Operation aus der Klasse in eine andere neue Klasse verschieben, dann ist der Zyklus doch weg.“ Durch mehrere Nachfragen wurde deutlich, dass für den Entwickler Klassen kein eigenes Konzept sind, sondern nur Sammlungen von Operationen sind, die umsortiert werden können, wenn es für das Auflösen eines Zyklus notwendig ist.

B.2.3 Ergebnisse der Vermessung

Das Softwaresystem hat 121.137 LOC und 65.284 RLOC. Es besteht aus 902 Klassen, die in 118 Packages organisiert sind. Die 118 Packages haben wir bei der Analyse in 15 Subsysteme aufgeteilt und die Subsysteme wiederum in vier Schichten gegliedert.

Von den 902 Klassen sind 99 Klassen an insgesamt 15 Zyklen beteiligt. Sechs dieser 15 Klassenzyklen bestehen aus Zweierzyklen. Es gibt zwei Zyklen aus drei Klassen und zwei Zyklen aus vier Klassen. Die verbleibenden fünf Zyklen bestehen aus sieben, acht, dreizehn, sechzehn und neunundzwanzig Klassen.

Im Softwaresystem gibt es 22 Packages, die an sechs Package-Zyklen beteiligt sind. Es gibt zwei Zweierzyklen, zwei Dreierzyklen und je einen Package-Zyklus mit fünf und mit sieben Packages. Auf der Ebene der Subsysteme hat das Softwaresystem keine Zyklen.

Im Softwaresystem haben 89% aller Klassen weniger als 10.000 Zeichen[3] und 10,75% der Klassen zwischen 70.000 und 10.000 Zeichen. Zwei Klassen überschreiten die Grenze von 70.000 Zeichen: Die Klasse MainFrameEx hat 129.076 Zeichen und 119 Operationen. Die Klassen KtoCheck hat 89.680 Zeichen und 13 Operationen.

Beide Klassen fallen bei weiteren Messungen auf:

Die Klasse MainFrameEx habt im Softwaresystem eine verhältnismäßig hohe Anzahl von Verbindungen zu anderen Klassen: MainFrameEx kennt 63 Klassen und wird von 18 Klassen benutzt. Im gesamten Softwaresystem kennen 98% aller Klassen weniger als 10 Klassen und werden von weniger als 10 Klassen benutzt.

Die Klasse KtoCheck hat eine Operation Kto_Check, die aus 74.764 Zeichen besteht. Diese Operation macht 83% des gesamten Sourcecodes dieser Klasse aus und ist mit Abstand die größte Operation des gesamten Softwaresystems. Die nächst kleinere Operation hat nur noch 21681 Zeichen und 99,85% aller Operationen haben weniger als 10.000 Zeichen. Die Operation Kto_Check aus der Klasse KtoCheck hat nicht nur auffällig viele Zeichen sondern auch eine extrem hohe zyklomatische Komplexität nach McCabe von 639, d.h. es gibt 639 mögliche Pfade durch den Algorithmus, der in dieser Operation implementiert ist.

Die Größe der Packages und Subsysteme ist bei diesem Softwaresystem nicht auffällig, d.h. den Subsysteme sind allen weniger als 15 Packages zugeordnet und alle Packages enthalten weniger als 50 Files.

B.2.4 Bewertung auf der Subsystem-Ebene

Das Softwaresystem hat auf der Subsystem-Ebene eine ausgewogene Struktur mit einer geringen Anzahl von Abhängigkeiten und daher eine niedrige

[3] Als ich diese Analyse gemacht habe, gab es im Sotograph nur eine Metrik um die Anzahl der Zeichen pro Klasse zu ermitteln. Eine Metrik für die Anzahl der Programmzeilen wurde erst später eingeführt.

Architekturkomplexität für zwei Kriterien zum Faktor Modularität. Bei der Messung der Package-Größen ebenfalls wurden keine auffälligen Werte registriert.

Höhere Komplexität hat das Softwaresystem bei der Mustertreue auf der Subsystem-Ebene. Bei der Definition der Subsysteme stellt sich heraus, dass die Package-Struktur nicht an den Subsystemen orientiert ist. Wir mussten Subpackages aus verschiedenen Package-Teilbäumen zu einem Subsystem zusammenstellen. Bei vier Packages mussten wir darüber hinaus die Klassen in zwei neue und zwei bereits existierende Packages aufteilen, damit die Klassen in das richtige Subsystem eingeordnet werden konnten.

Die sich schließlich ergebenden Schichten spiegeln den Aufbau einer Client-Server-Architektur wieder: Anwendungen, Client, Server und Basis (s. Abbildung B-4).

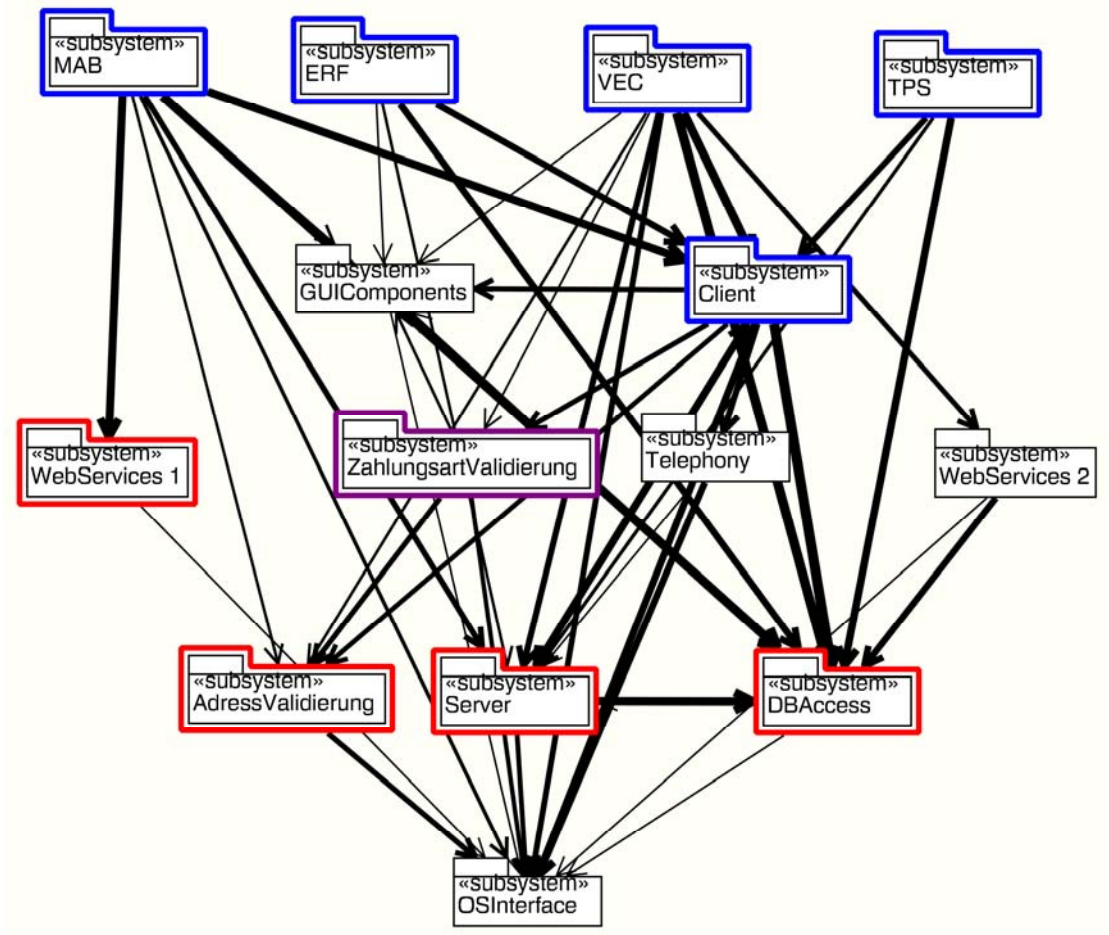

Abbildung B-4: Subsysteme und Schichten von Fallstudie 11[4]

[4] Die rot umrandeten Package-Symbole sind Subsysteme, die Implementierungen von EJB-Komponenten enthalten. Die blau umrandeten Package-Symbole sind Subsystem, die Klassen beinhalten, die EJB-Komponenten benutzen. Die lila umrandeten Package-Symbole sind Subsysteme, die sowohl aus Implementierungen von EJB-Komponenten als auch aus EJB-Komponenten benutzende Klassen bestehen.

Die Subsysteme und Schichten sind im Softwaresystem zyklenfrei, so dass auf der Subsystem-Ebene auch in Bezug auf Geordnetheit eine niedrige Komplexität herrscht.

Auf der Package-Ebene waren viele Zyklen zu finden, die nicht durch das Verschieben von Klassen zwischen Packages aufzulösen werden konnten. Die Zyklenreichweite ist in diesem Softwaresystem hoch.

Die Möglichkeiten zur Komplexitätsminderung, die man durch Schnittstellen hat, wurden im Softwaresystem nicht genutzt. Im Softwaresystem befinden sich, wie in der Diskussion mit dem Entwickler deutlich wurde (s. Ablauf der Analyse), Subsysteme, die eine gemeinsame oder wenigstens eine jeweils eigene definierte Schnittstelle haben sollten. Dadurch dass Subsysteme restrukturiert und die jeweilige Schnittstelle in einem eigenen Package zusammengefasst wird, könnte durch Schnittstellen die Komplexität weiter reduziert werden.

Für die Subsystem-Ebene lässt sich insgesamt feststellen, dass das Softwaresystem nur bei der Modularität und der Geordnetheit der obersten Subsysteme wenig Architekturkomplexität hat. Mustertreue war nicht vorhanden, Schnittstellen wurde nicht gebildet und Geordnetheit auf Package-Ebene war nicht vorhanden.

B.2.5 Bewertung auf der Klassen-Ebene

Auf der Klassen-Ebene kommt durch den Schnitt der Klassen höhere Architekturkomplexität zustande. Es besteht keine Ausgewogenheit und die Abhängigkeiten zwischen den Klassen sind hoch (s. Abschnitt Komplexitätsmessung).

Die Diskussion mit dem Entwickler während der Analyse hat außerdem deutlich gemacht, dass die Komplexität auf der Klassen-Ebene im Softwaresystem nicht nur durch die Größe der Klassen, sondern auch durch den Verteilung der Funktionalität auf die einzelnen Klassen erhöht wird. Klassen enthalten sowohl Programmcode für die Oberfläche, als auch Geschäftslogik und technische Lösungen. Wären die verschiedenen Zuständigkeiten eindeutig Klassen zugeordnet, so würde sich die Komplexität auf der Klassen-Ebene im Kriterium Zusammenhalt verringern.

Die Zyklen zwischen Klassen sprechen für geringe Geordnetheit auf der Klassen-Ebene. Von den im Softwaresystem gefundenen Klassenzyklen ließen sich einige Zyklen durch Diskussion mit dem Entwickler auflösen: die kleineren Zweier- und Dreierzyklen sowie der Zyklus aus sechzehn Klassen. Dieser Zyklus wurde durch einen einzigen Aufruf einer Konstante zwischen den zwei zentralen Klassen dieses Zyklus verursacht. Die anderen Klassenzyklen aus sieben, acht, dreizehn und neunundzwanzig Klassen konnten wir in der kurzen Zeit einer zweitägigen Analyse nicht auflösen. Diese Zyklen beinhalten viele Beziehungen zwischen den einzelnen Klassen, so dass das Auflösen dieser Zyklen nur durch ein Redesign der Klassen möglich gewesen wäre.

Für die Klassen-Ebene lässt sich zusammenfassend sagen, dass das Softwaresystem hohe Architekturkomplexität hat, weil es schlecht modularisiert und nicht geordnet ist.

B.2.6 Erkenntnisse aus der Untersuchung

An diesem Softwaresystem sind mir die folgenden Punkte deutlich geworden:

- Ein Softwaresystem kann auf der Subsystem-Ebene zum Teil niedrige Komplexität haben und trotzdem auf der Klassen-Ebene hohe Komplexität aufweisen, so dass es den Entwicklern Probleme bei der Wartung und Weiterentwicklung bereitet.
- Die Schwierigkeiten auf der Klassen-Ebene entsteht, weil die Entwickler Klassen zum Teil nur als Sammlungen von Operationen betrachtet, zwischen denen Operationen verschoben werden bzw. deren Operationen in neue Klassen eingefügt werden, wenn dadurch die Komplexität auf der Klassen-Ebene verringert werden kann. Das Klassen eine Zuständigkeit haben, wurde hier nicht beachtet oder war nicht bekannt.
- Das Fehlen von Mustertreue auf der Klassen-Ebene führt dazu, dass das Konzept der Klasse keine Differenzierung aufweist. Jede Klasse darf potentiell zu jeder anderen Klasse in Beziehungen stehen.
- Die Komplexität von Klassenzyklen kann man nicht allein an der Anzahl der Klassen in einem Zyklus festmachen. Es gab im Softwaresystem einen Zyklus mit 16 beteiligten Klassen, der sich auflösen ließ, indem eine einzige Verbindung zwischen zwei Klassen getrennt wurde. Bei der Betrachtung der Geordnetheit spielt daher nicht nur die Anzahl der beteiligten Klassen sondern auch die Anzahl der Beziehungen zwischen den Klassen eine Rolle.

B.3 Fallstudie 20

Fallstudie 20 wurde bei einem deutschen Finanzdienstleister mit Stammsitz in Hamburg durchgeführt, der bedarfsgerechte, an Lebensphasen ausgerichtete Vorsorge-Lösungen für Privatkunden bieten. Die Palette der angebotenen Versicherungsprodukte reicht von Lebensversicherungen über Sach-, Renten- und Krankenversicherungen bis zu Finanzdienstleistungen, wie Bausparverträge und Investmentfonds.

B.3.1 Ablauf der Analyse

Für drei Tage war ich mit einer Kollegin bei dieser Firma, um Mitarbeiterinnen und Mitarbeiter aus der Entwicklungsabteilung in der Benutzung des Sotographen zu schulen. Dabei haben wir das Softwaresystem als Beispiel verwendet, das von diesem Entwicklungsteam betreut wird. Dieses Softwaresystem wird von Außendienstmitarbeiterinnen und -mitarbeitern verwendet wird, um für Kunden Versicherungsprodukte auszurechnen und zu buchen.

Während der Sotographen-Schulung haben wir die Funktionalität des Sotographen anhand einer Analyse ihres eigenen Softwaresystems vermittelt. Die Ergebnisse der einzelnen Analyseschritte haben wir gemeinsam mit den Entwicklerinnen und Entwicklern diskutiert und so einen Einblick in die Komplexität des Softwaresystems gewonnen.

Am ersten Tag der Schulung haben wir uns einen Überblick über das Softwaresystem verschafft, die Packages in Subsysteme eingeteilt und Zyklen und Metriken auf Subsystem-Ebene ausgewertet. Am zweiten Tag haben wir Zyklen und Metriken auf der Klassen-Ebene überprüft. Am dritten Tag haben wir den Entwicklern weitere Einsatzkontexte des Analyse-Werkzeugs, wie die Webschnittstelle und das Einbinden des Sotographen in automatisierte Build-Prozesse erklärt.

Schon am ersten Tag wurde deutlich, dass die Entwicklerinnen und Entwickler eine genaue Architekturvorstellung von ihrem Softwaresystem hatten. Es war sehr einfach, Subsysteme zu definieren und ihnen Packages zuzuordnen. Die Aufteilung der Packages in Subsysteme fand sich direkt in der Package-Struktur wieder.

Außerdem konnten wir für die Subsysteme Schnittstellen festlegen, die jeweils aus den Klassen im obersten Package eines Subsystems bestanden. Die Entwicklerinnen und Entwickler nahmen dabei immer wieder ihre im Netz verfügbare Dokumentation zur Hand und lasen dort nach, welche Schnittstellen die einzelnen Subsysteme haben sollten. Für die Entwicklerinnen und Entwickler war es sehr wichtig, mit Hilfe des Analyse-Werkzeugs zu ermitteln, wie die Schnittstellen ihrer Subsysteme verletzt werden und wo ihre Architekturvorstellung über die Benutzung zwischen den Subsystemen missachtet wurde. Wir konnten feststellen, dass einige Schnittstellen mehr Klassen enthalten mussten, als ursprünglich angenommen bzw. in der

Dokumentation vermerkt und dass einige Schnittstellenverletzungen durch Refactorings behoben werden sollten (s. rote Subsysteme in Abbildung B-5).

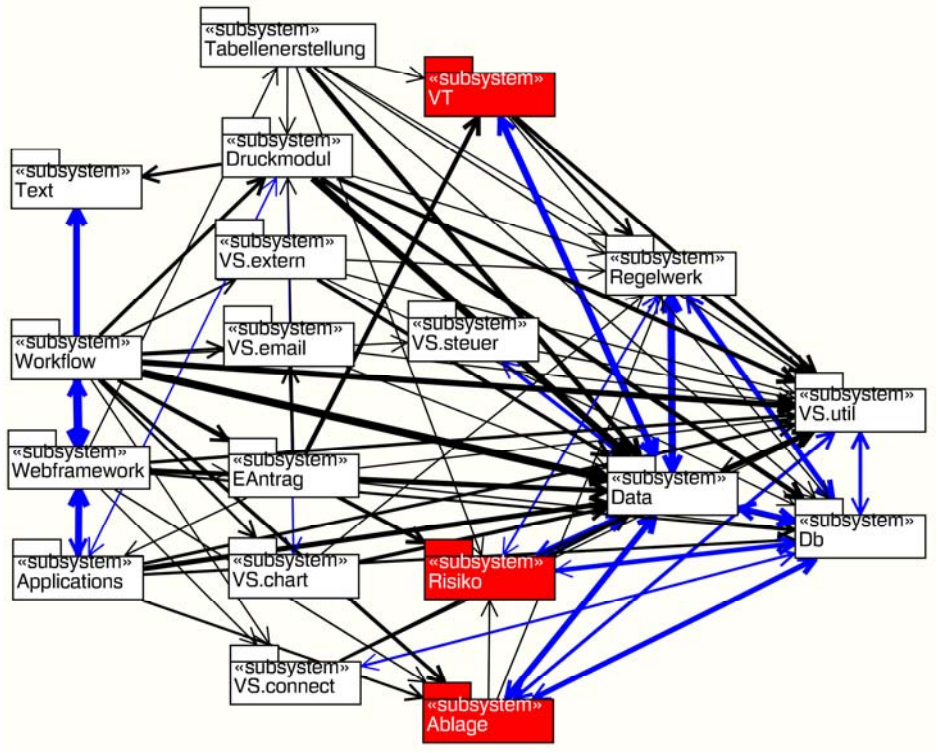

Abbildung B-5: Subsysteme mit Zyklen und Schnittstellenverletzungen[5]

Neben den Schnittstellen für die Subsysteme gab es ebenfalls klare Vorstellungen darüber, welche Subsysteme welche anderen Subsysteme benutzen dürfen. Die Entwickler beschrieben ihr Softwaresystem als eine Sammlung von Subsystemen, für deren Beziehungen Regeln definiert werden konnten. Als wir die Architektur im Sotographen definieren wollten, stellten wir fest, dass wir in diesem Fall keine Schichtenarchitektur verwenden konnten. Dieses Softwaresystem hat als Architekturstil eine Subsystemarchitektur. Die Beziehungen zwischen den Subsystemen sind über Regeln definiert, die Subsysteme stehen nicht in einer hierarchischen Beziehung zueinander und sind daher nicht in hierarchische Schichten zu gliedern.

Zyklen waren für die Entwickler dieses Softwaresystems bisher kein Problem (s. direkte Zyklen als blaue Pfeile in Abbildung B-5). Als wir die entsprechenden Messungen zu Zyklen durchgeführt hatten und auf ein schlechtes Ergebnis kamen, fanden die Entwicklerinnen und Entwickler diesen Befund nicht problematisch.

Für uns wurde bei der Analyse deutlich, dass dieses Entwicklungsteam mit Hilfe des Analyse-Werkzeugs Schwachstellen ihre Schnittstellen finden wollten. Architekturkomplexität auf der Klassen-Ebene war für dieses Entwicklungsteam weniger wichtig.

[5] Rote Subsysteme haben Schnittstellenverletzungen. Die blauen Pfeile machen bidirektionale Beziehungen zwischen Subsystemen sichtbar.

B.3.2 Entwicklungssituation

Gleich zu Anfang des Workshops haben die Entwicklerinnen und Entwickler uns gesagt, dass sie nicht das Gefühl hätten, dass ihr Softwaresystem ein architektonisches Problem habe. Während der Analyse würde deutlich, dass das Team eine gut durchdachte Soll-Architektur hat und diese in für alle zugängliche Dokumenten festgehalten hat.

Für die einzelnen Subsysteme sind in diesem Team jeweils einzelne Entwicklerinnen und Entwickler zuständig. Die Zusammenarbeit zwischen den Subsystemen gestaltet dieses Team so, dass sie die Schnittstellen zwischen den Subsystemen festlegen und nur über diese Schnittstellen kommuniziert werden darf.

B.3.3 Ergebnisse der Vermessung

Das Softwaresystem hat 789.938 LOC und 344.911 RLOC. Es besteht aus 1.396 Klassen, die in 119 Packages organisiert sind. Die 119 Packages haben wir bei der Analyse in 21 Subsysteme aufgeteilt.

Von den 1.396 Klassen sind 1.059 Klassen an 3 Zyklen beteiligt. Einer dieser 3 Klassenzyklen bestehen aus 1.050 Klassen. Die anderen beiden Zyklen haben zwei und sieben Klassen.

Im Softwaresystem sind 95 von 119 Packages in einem Package-Zyklus miteinander verbunden. Auf der Ebene der Subsysteme gibt es zwei Zyklen mit sieben und acht beteiligten Subsystemen.

Das Softwaresystem enthält große Klassen mit langen Operationen. 55 Klassen, also 4% aller Klassen, haben mehr als 100.000 Zeichen, und 481 Klassen, also 35,5% aller Klassen, haben mehr als 10.000 Zeichen. Auch eine Reihe von Operationen ist groß und hat eine hohe zyklomatische Komplexität: Es gibt 24 Operationen mit dem Namen „performAktion“ die alle zwischen 39.432 und 111.298 Zeichen haben. Die längste dieser Operationen hat eine zyklomatische Komplexität (Anzahl möglicher Pfade) von 194. Der empfohlene Grenzwert für zyklomatische Komplexität liegt bei 10.

Die Größe einiger Packages ist bei diesem Softwaresystem ebenfalls auffällig. Es gibt ein Package mit 268 und ein weiteres Package mit 112 Klassen. Die Anzahl der Packages pro Subsystem hingegen liegt unterhalb von 30 Packages.

B.3.4 Bewertung auf der Subsystem-Ebene

Das Softwaresystem hat auf der Subsystem-Ebene eine mustertreue und modulare Struktur, wodurch die Architekturkomplexität maßgeblich reduziert wird. Die Definition der 21 Subsysteme war leicht, weil die Package-Struktur den Subsystemen genau entsprach. Die Subsysteme hatten klar definierte Aufgaben mit hohem Zusammenhalt. Außerdem hat das Entwicklungsteam die einzelnen Subsysteme mit Schnittstellen versehen und die Schnittstellen bei allen Subsystemen im obersten Package angesiedelt. Sehr

viel Arbeit des Entwicklerteams ist bei diesem Softwaresystem darauf verwendet worden, die Komplexität durch Subsystembildung und Schnittstellen zu reduzieren.

Die Möglichkeit, die Architekturkomplexität durch Geordnetheit zu reduzieren, haben die Entwicklerinnen und Entwickler nicht genutzt. Die zwei Zyklen auf der Subsystem-Ebene enthalten zwischen den sieben bzw. acht Subsystemen siebzehn direkte Zweierbeziehungen, die zu größeren Zyklen zusammengeschlossen sind.

B.3.5 Bewertung auf der Klassen-Ebene

Auf der Klassen-Ebene hat das Softwaresystem sowohl bei der Modularität als auch bei der Geordnetheit eine hohe Komplexität. Die Anzahl der Klassen ist im Vergleich zu den anderen Softwaresystemen gering. Es gibt verhältnismäßig wenig Klassen und viele Klassen sind relativ groß (4% größer als 100.000 Zeichen, 35,5% mehr als 10.000 Zeichen). Weiter lässt sich feststellen, dass die Klassen im Vergleich mit anderen Softwaresystemen eine ähnliche Anzahl von Operationen pro Klasse aufweisen. Dieses Verhältnis kommt dadurch zustande, dass die einzelnen Operationen doppelt bis dreimal so lang sind wie in vergleichbaren Softwaresystem. Auch die zyklomatische Komplexität der Operationen ist hoch. Würden die langen Operationen in mehrere einzelne Operationen aufgeteilt und die Operationen gleichzeitig auf weitere Klassen aufgeteilt, so könnte die Komplexität auf der Klassen-Ebene reduziert werden. Neben der Größe der Klassen ist auch die Anzahl der Klassen in Zyklen im Vergleich zu anderen Softwaresystemen hoch und die Zyklen sind groß.

Während der Analyse machte uns das Entwicklungsteam deutlich, dass es mit der Komplexität auf Klassen-Ebene wenige Schwierigkeiten hat, weil die jeweiligen Subsystem-Verantwortlichen ihren Systemteil sehr gut kennen und selbst implementiert haben.

B.3.6 Erkenntnisse aus der Untersuchung

An diesem Softwaresystem sind mir die folgenden Punkte deutlich geworden:

- Ein Softwaresystem kann auf der Subsystem-Ebene niedrige Architekturkomplexität aufweisen und gleichzeitig auf der Klassen-Ebene eine hohe Komplexität haben. Sind in einem Team wie in diesem Fall einzelne fähige Entwicklerinnen und Entwickler für die verschiedenen Teilsysteme zuständig und die Schnittstellen klar definiert, so hat das Team keine Probleme mit Architekturkomplexität (Geordnetheit, Modularität) auf der Klassen-Ebene.
- Die Klassen, die zusammen die Schnittstelle für die Subsysteme bilden, sind nur die Export-Schnittstelle des Subsystems. Die Importschnittstelle und damit die Abhängigkeiten, die das Subsystem von anderen Subsystemen hat, sind so nicht zu erkennen.

Schnittstellenumfang und Abhängigkeit müssen als Kriterien unterschieden werden.

B.4 Fallstudie 22

Fallstudie 22 habe ich bei einer bundesweit tätigen gesetzlichen Krankenkasse mit rund 4 Millionen Mitgliedern und Hauptsitz in Hamburg durchgeführt. Das Softwaresystem deckt einen großen Teil der Funktionalität ab, die die Mitarbeiter der Krankenkasse für ihre Arbeit benötigen.

Das Softwaresystem wird seit 1998 in Java entwickelt, um die Altsysteme abzulösen. Ca. 60% der Altanwendungen waren zum Zeitpunkt der Analyse bereits migriert, die restlichen 40% sollen in den nächsten Jahren folgen.

Die Entwicklungsabteilung besteht aus ca. 120 Entwicklerinnen und Entwicklern und ist in zwei Bereiche: Framework-Abteilung und Anwendungsentwicklung aufgeteilt.

B.4.1 Ablauf der Analyse

An drei verschiedenen Tage habe ich die Entwicklungsabteilung besucht und Interviews mit zwei Framework-Architekten geführt. Parallel habe ich das Softwaresystem in den Sotographen eingelesen und mit einem Architekten untersucht. Aufgrund seiner Größe und der zeitlichen Restriktionen der Architekten war eine vollständige Analyse der Subsystem-Ebene nicht möglich, so dass die Messwerte der Subsystem-Ebene nicht aussagekräftig sind.

Bei der Diskussion mit den Architekten wurde deutlich, dass das Softwaresystem eine projekteigene Referenzarchitektur hat. Die Elementarten und Regeln der Referenzarchitektur wurden mir anhand von Schulungsunterlagen vermittelt. Zusätzlich zur Referenzarchitektur wird eine erweiterte Schichtenarchitektur mit Schnittstellen verwendet, um die Architekturkomplexität zu kontrollieren.

B.4.2 Entwicklungssituation

Dieses Softwaresystem wurde im Jahr 1999 bei einer Größe von ca. 250.000 LOC auf seine Qualität hin untersucht. Diese Untersuchung hat dazu geführt, dass Architekturverletzungen bereinigt wurde und die Architekten sich viele Gedanken über die Architektur und ihre zukünftige Entwicklung gemacht haben. Die projekteigene Referenzarchitektur und die Schichtenarchitektur waren als Architekturstile zu dieser Zeit schon vorhanden und wurden weiter ausdifferenziert. Dass die Referenzarchitektur und die erweiterte Schichtenarchitektur eingehalten werden, wird durch die folgenden Mittel sichergestellt:

1. Die Referenzarchitektur und die erweiterte Schichtenarchitektur sind in Dokumenten ausführlich dokumentiert und alle neuen Entwicklerinnen und Entwicklern müssen mehrtägige Schulungen durchlaufen, bevor sie in der Entwicklung mitarbeiten können.
2. Grundlegende Funktionalitäten der Elementarten sind im Framework implementiert, so dass die Anwendungs-

entwicklerinnen und -entwickler in vielen Bereichen dazu gezwungen sind, sich an die Vorgaben der Referenzarchitektur zu halten.

3. In der Architekturgruppe wurden verschiedene Prüfprogramme entwickelt, die dazu verwendet werden, qualitative Mängel des Softwaresystems zu finden und zu beheben. Soll ein neu entwickelter Teilbereich des Softwaresystems in das Gesamtsystem integriert werden, so müssen die von den Qualitätsprüfern gefundenen Mängel vorher behoben werden.

Die Architekten waren überzeugt davon, dass ihr Softwaresystem nur durch diese Maßnahmen so groß werden und so lange beherrschbar bleiben konnte.

B.4.3 Ergebnisse der Vermessung

Das Softwaresystem aus Anwendungs- und Framework-Programmtext hat ohne generierten Programmtext 2,9 Mio LOC und 1,3 Mio RLOC. Der generierte Programmtext umfasst weitere 1,2 Mio LOC. Es besteht aus 23.915 Klassen, die in 3.496 Packages organisiert sind. Die 3.496 Packages haben wir bei der Analyse in 210 Subsysteme aufgeteilt. Die Subsysteme des Frameworks wurden in sieben Schichten strukturiert und die Subsysteme der fachlichen Anwendung in sechs Schichten.

Von den 23.915 Klassen sind 1.691 Klassen an 364 Zyklen beteiligt. Das heißt lediglich 7% aller Klassen sind in Klassenzyklen. Im Softwaresystem sind 599 von 3.496 Packages in 95 Package-Zyklus miteinander verbunden (= 17%). Auf der Ebene der Subsysteme gibt es zwei Zyklen: Ein Zyklus im Framework mit 21 beteiligten Subsystemen und ein Zyklus zwischen Subsystemen der fachlichen Anwendung mit 107 beteiligten Subsystemen.

Zwischen den Schichten des Softwaresystems gibt es 92 Aufwärtsbeziehungen. Ein Teil dieser Verletzungen rührt daher, dass das erstellte Subsystemmodell verbessert werden muss und Klassen in falschen Packages und damit in falschen Subsystemen untergebracht sind.

Das Softwaresystem enthält lediglich 4 große Klassen, die mehr als 100.000 LOC lang sind. 95% aller Klassen haben weniger als 50.000 LOC. 85% aller Packages haben weniger als 50 Klassen, das größte Package hat 112 Klassen.

B.4.4 Bewertung auf der Subsystem-Ebene

Das Softwaresystem hat auf der Subsystem-Ebene eine mustergetreue Struktur. Die Abbildung des Package-Baums auf 210 Subsysteme hat den Architekten keine Schwierigkeiten gemacht, weil sie direkt aus der Package-Struktur ablesbar waren.

Die Subsysteme waren relativ klein, weil dieses Team für jeden Schnitt alle Schichten als eigene Subsysteme gebildet hat und außerdem Framework- und Anwendungs-Subsysteme unterscheidet. Außerdem hat das Entwicklungs-

team die einzelnen Subsysteme mit Schnittstellen versehen und die Schnittstellen bei allen Subsystemen in einem eigenen Package angesiedelt.

Durch die erweiterte Schichtenarchitektur enthält die Schichtung eine geringe Anzahl an Zyklen zwischen Subsystemen in verschiedenen Schichten. Über die Anzahl der Zyklen zwischen den Subsystemen kann keine exakte Aussage getroffen werden, weil aus Zeitgründen nur ein grobes Subsystemmodell erstellt wurde.

Das Softwaresystem auf der Subsystem-Ebene durch seine gute Strukturierung geringe Architekturkomplexität. Die Größe des Softwaresystems führt allerdings dazu, dass die Entwickler mit 210 Subsystemen konfrontiert werden und allein dadurch Komplexität entsteht.

B.4.5 Bewertung auf der Klassen-Ebene

Auf der Klassen-Ebene hat das Softwaresystem durch die Referenzarchitektur geringe Architekturkomplexität. Für die einzelnen Klassen ist klar festgelegt, welche Aufgaben sie haben sollen und mit welchen anderen Teilen des Softwaresystems sie deshalb kommunizieren dürfen. Durch das Einführen von Regeln und das Überprüfen dieser Regeln mit eigenen Prüfprogrammen wird diese geringe Komplexität sichergestellt.

B.4.6 Erkenntnisse aus der Untersuchung

An diesem Softwaresystem sind mir die folgenden Punkte deutlich geworden:

- Softwaresysteme, mit einer Referenzarchitektur haben auch bei enormer Größe noch wenige Klassenzyklen. Package-Zyklen sind davon unbeeinflusst.
- Bei großen Softwaresystemen sind zusätzlich Architekturstile für die Subsystem-Ebene notwendig.
- Frühes und kontinuierliches Prüfen der Architektur ist die Voraussetzung für große Softwaresysteme.

C Tabellen der Messergebnisse

Legende zu den Tabellen der Messergebnissen auf den folgenden Seiten:

Referenzarchitektur
Schichtenarchitektur
Subsystemarchitektur

Tabelle C-1: Ausgewogenheit und Abhängigkeit

Fallstudie	RLOC	Sys-Classes	RLOC/SysClasses	RefBetween-Classes	RefBetClasses/SysClasses
1	10.679	242	44,1	1.064	4,4
2	13.219	250	52,9	897	3,6
3	16.314	398	41,0	2.763	6,9
4	21.591	356	60,6	2.716	7,6
5	22.957	358	64,1	1.591	4,4
6	23.478	686	34,2	2.806	4,1
7	33.651	624	53,9	3.099	5,0
8	36.671	590	62,2	4.122	7,0
9	43.969	731	60,1	3.106	4,2
10	49.428	686	72,1	2.716	4,0
11	65.284	902	72,4	4.025	4,5
12	70.266	365	192,5	7.325	20,1
13	74.344	782	95,1	5.151	6,6
14	93.593	1.158	80,8	6.276	5,4
15	112.675	1.950	57,8	13.344	6,8
16	120.964	2.124	57,0	16.402	7,7
17	133.095	2.249	59,2	12.531	5,6
18	143.101	1.140	125,5	12.623	11,1
20	344.911	1.396	247,1	36.096	25,9
21	629.674	9.382	67,1	79.888	8,5
22	1.296.455	23.915	54,2	279.128	11,7
23	2.869.566	23.442	59,1	262.064	11,2

Tabelle C-2: Geordnetheit auf Klassen-Ebene

Fallstudie	SysClasses	SysReferences	ClassesInCycle	% von SysClasses	RefClassesInCycles	‰ von SysReferences	Maxmimaler Wert von ClassesInCycle	% von SysClasses	DirectRefsInClassCycles	‰ von SysReferences
1	242	11.452	0	0,0	0	0,0	0	0,0	0	0,0
2	250	11.824	5	2,0	9	0,8	3	1,2	3	2,5
3	398	16.473	4	1,0	6	0,4	4	1,0	1	0,6
4	356	19.544	0	0,0	0	0,0	0	0,0	0	0,0
5	358	16.743	31	8,7	65	3,9	5	1,4	16	9,6
6	686	18.609	69	10,1	114	6,1	18	2,6	25	13,4
7	624	31.892	31	5,0	48	1,5	7	1,1	13	4,1
8	590	49.646	348	59,0	2.091	42,1	270	45,8	218	43,9
9	731	36.168	3	0,4	4	0,1	3	0,4	2	0,6
10	686	59.469	26	3,8	51	0,9	8	1,2	7	1,2
11	902	50.828	99	11,0	218	4,3	29	3,2	45	8,9
12	365	79.491	15	4,1	58	0,7	3	0,8	11	1,4
13	782	75.355	88	11,3	283	3,8	66	8,4	90	11,9
14	1.158	68.516	308	26,6	1.234	18,0	150	13,0	83	12,1
15	1.950	106.008	500	25,6	1.352	12,8	119	6,1	224	21,1
16	2.124	118.958	403	19,0	1.112	9,4	101	4,8	189	15,9
17	2.249	215.696	147	6,5	249	1,2	19	0,8	105	4,9
18	1.140	132.212	207	11,9	1.409	10,7	44	3,9	76	5,8
20	1.396	506.229	1.059	75,9	16.086	31,8	1.050	75,2	2.077	41,0
21	9.382	665.857	492	5,2	990	1,5	42	0,5	296	4,5
22	23.915	1.265.990	1.691	7,1	4.029	3,2	401	1,7	709	5,6
23	23.442	1.740.670	5.764	24,6	26.685	15,3	917	3,9	2.718	15,6

Tabelle C-3: Ausmaß und Reichweite auf Package-Ebene

Fallstudie	SysPackages	RefsBetweenPackages	PckgInCycle	% von SysPackages	RefsPackagesInCycles	% von SysPackages	PackageCycleGroups	PackageCycleGroups-FromClassCycles	% von Package-CycleGroups	RefsInClassCycles-BetweenPackages	% von RefsInPackage-Cycles
1	31	4.446	2	6,5	4	9,0	2	0	0,0	0	0,0
2	26	4.644	4	15,4	11	23,7	2	1	50,0	3	27,3
3	51	9.099	10	19,6	21	23,1	2	0	0,0	0	0,0
4	70	10.119	2	2,9	2	2,0	1	0	0,0	0	0,0
5	65	7.116	5	7,7	7	9,8	2	1	50,0	4	57,1
6	142	8.376	38	26,8	109	130,1	6	5	83,3	42	38,5
7	66	14.773	29	43,9	87	58,9	5	1	20,0	10	11,5
8	25	25.788	19	76,0	105	40,7	1	1	100,0	84	80,0
9	41	14.896	35	85,4	189	126,9	1	1	100,0	2	1,1
10	66	39.050	8	12,1	15	3,8	2	1	50,0	3	20,0
11	118	14.078	22	18,6	35	24,9	6	5	83,3	26	74,3
12	31	54.018	25	80,7	171	31,7	1	1	100,0	12	7,0
13	195	39.565	57	29,2	176	44,5	5	3	60,0	73	41,5
14	71	31.962	42	59,2	168	52,6	4	4	100,0	158	94,1
15	105	36.264	41	39,1	98	27,0	10	6	60,0	60	61,2
16	171	48.875	64	37,4	194	39,7	11	7	63,6	67	34,5
17	610	74.952	30	4,9	42	5,6	11	4	36,3	8	19,1
18	297	87.518	81	27,3	241	27,5	14	10	71,4	112	46,5
20	119	321.612	95	79,8	1.183	36,8	1	1	100,0	1.048	88,6
21	516	413.143	341	66,1	3.535	85,6	9	6	66,7	584	16,5
22	3.496	641.649	599	17,1	2.052	32,0	95	50	52,6	498	24,3
23	978	1.025.240	667	68,2	3.694	36,0	58	52	89,7	1.996	54,0

Tabelle C-4: Umfang und Verflochtenheit auf Package-Ebene

Fallstudie	SysPackages	RefsBetweenPackages	Maxmimaler Wert von PckgIn-Cycle	% von SysPackages	DirectRefsInPackageCycles	‰ von RefsBetweenPackages	PackagesInCycleGroupFrom SuperSubRefs	% von PckgInCycle
1	31	4.446	2	6,5	1	4,5	2	100,0
2	26	4.644	4	15,4	2	8,6	3	75,0
3	51	9.099	8	15,7	3	4,4	4	40,0
4	70	10.119	2	2,9	1	1,0	2	100,0
5	65	7.116	3	4,6	2	4,2	4	80,0
6	142	8.376	13	9,2	13	31,0	16	42,1
7	66	14.773	21	31,8	9	8,8	6	20,7
8	25	25.788	19	76,0	28	10,9	0	0,0
9	41	14.896	35	85,4	18	12,1	14	40,0
10	66	39.050	6	9,1	3	1,0	4	50,0
11	118	14.078	7	5,9	4	7,8	8	36,4
12	31	54.018	25	80,7	16	2,4	4	16,0
13	195	39.565	36	18,5	21	8,3	13	26,5
14	71	31.962	24	33,8	20	9,1	22	52,4
15	105	36.264	10	9,5	16	8,8	24	58,5
16	171	48.875	23	13,5	21	10,6	22	34,4
17	610	74.952	4	0,7	3	2,0	14	46,7
18	297	87.518	36	12,1	34	6,7	45	55,6
20	119	321.612	95	79,8	118	3,7	21	22,1
21	516	413.143	225	43,6	122	4,7	116	34,0
22	3.496	641.649	160	4,6	83	4,9	178	29,7
23	978	1.025.240	86	8,8	145	7,7	319	47,8

Tabelle C-5: Ausmaß und Reichweite auf Subsystem-Ebene

Fallstudie	SysSubsystems	RefsBetweenSubsystems	SubsysInCycle	% von SysSubsystems	RefSubsysInCycles	% von RefsInSubsystem-Cycles	SubsysCycleGroups	SubsysCycleGroups-FromClassCycles	% von SubsysCyc-leGroups	RefsInClassCycles-BetweenSubsystems	% von RefsInSubsystem-Cycles
1	22	4.177	2	9,1	2	2,4	1	0	0,0	0	0,0
2	14	4.586	6	42,9	9	6,5	2	1	50,0	3	33,3
3	7	7.174	5	71,4	12	4,2	1	0	0,0	0	0,0
4	9	4.843	0	0,0	0	0,0	0	0	0,0	0	0,0
5	12	4.673	4	33,3	4	4,3	2	1	50,0	2	50,0
6	18	5.503	0	0,0	0	0,0	0	0	0,0	0	0,0
7	25	25.378	12	48,0	16	2,4	4	1	25,0	8	50,0
8	19	12.700	17	89,5	95	20,5	1	1	100,0	76	80,0
9	18	7.705	6	33,3	14	7,8	1	0	0,0	0	0,0
10	11	9.668	6	54,6	18	4,1	1	1	100,0	2	11,1
11	15	49.067	0	0,0	0	0,0	0	0	0,0	0	0,0
12	9	26.791	3	33,3	34	3,0	1	1	100,0	8	23,5
13	6	24.295	0	0,0	0	0,0	0	0	0,0	0	0,0
14	7	21.583	0	0,0	0	0,0	0	0	0,0	0	0,0
15	19	19.709	0	0,0	0	0,0	0	0	0,0	0	0,0
16	11	21.176	0	0,0	0	0,0	0	0	0,0	0	0,0
17	70	56.514	0	0,0	0	0,0	0	0	0,0	0	0,0
18	17	83.159	0	0,0	0	0,0	0	0	0,0	0	0,0
20	21	256.716	16	71,4	57	0,7	2	2	100,0	46	80,7
21	69	342.724	42	60,9	314	2,1	3	2	66,7	24	7,6
22	210	497.203	101	51,0	1.053	0,6	2	0	0,0	43	4,1
23	23	504.525	0	0,0	0	0,0	0	0	0,0	0	0,00

Tabelle C-6: Umfang und Verflochtenheit auf Subsystem-Ebene

Fallstudie	SysSubsystems	RefsBetweenSubsystems	Maxmimaler Wert von SubsysInCycle	% von SysSubsystems	DirectRefsInSubsysCycles	‰ von RefsBetweenSubsystems
1	22	4.177	2	9,1	1	2,4
2	14	4.586	3	21,4	3	6,5
3	7	7.174	5	71,4	3	4,2
4	9	4.843	0	0,0	0	0,0
5	12	4.673	2	16,7	2	4,3
6	18	5.503	0	0,0	0	0,0
7	25	25.378	5	20,0	6	2,4
8	19	12.700	17	89,5	26	20,5
9	18	7.705	6	33,3	6	7,8
10	11	9.668	6	54,6	4	4,1
11	15	49.067	0	0,0	0	0,4
12	9	26.791	8	88,9	8	3,0
13	6	24.295	0	0,0	0	0,0
14	7	21.583	0	0,0	0	0,0
15	19	19.709	0	0,0	0	0,0
16	11	21.176	0	0,0	0	0,0
17	70	56.514	0	0,0	0	0,0
18	17	83.159	0	0,0	0	0,0
20	21	256.716	9	42,9	18	0,7
21	69	342.724	28	40,6	71	2,1
22	210	497.203	107	51,0	31	0,3
23	23	504.525	0	0,0	0	0,0

Literatur

Alexander, C. (1982): *A City is not a Tree*. In Kaplan, S. and Kaplan, R. (eds.): Humanscape - Environments for People, S. 377-402.

AlSharif, M., Bond, W. P. und Al-Otaiby, T. (2004): *Assessing the Complexity of Software Architecture*. In ACMSE'04, (Huntsville, Alabama), ACM Press, S. 98-103.

Anderson, J. R. (1989): *Kognitive Psychologie: eine Einführung*. Spektrum der Wissenschaft Verlagsgesellschaft mbH & Co., Heidelberg.

ANSI/IEEE-Standard-1471 (2000): *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - Description.* ANSI/IEEE Computer Society.

Bachmann, F., Bass, L. und Klein, M. (2003): *Deriving Architectural Tactics: A Step toward Methodical Architectural Design*. Software Engineering Institute, Nr. SEI/CMU-2003-TR-004.

Balzert, H. (1998): *Lehrbuch der Softwaretechnik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, Heidelberg, Berlin.

Banker, R. D., Datar, S. M., Kemerer, C. F. und Zweig, D. (1993): *Software complexity and maintenance costs*. Communications of the ACM, 36, (11), S. 81-94.

Basili, V. R. (1995): *Applying the Goal/Question/Metric paradigm in the experience factory*. In Fenton, N.E., Whitty, R. and Iizuka, Y. (eds.): Software Quality: Assurance and Measurement - A worldwide Perspective, International Thomson Computer Press, London, S. 23-44.

Basili, V. R. (1980): *Quanitative Software complexity models: a panel summary*. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost (Oct. 1979), IEEE Computer Society Press, Los Alamitos, S. 243-245.

Basili, V. R. und Weiss, D. M. (1984): *A Methodology for Collecting Valid Software Engineering Data*. IEEE Transaction on Software Engineering, 6, (SE-10), S. 728-738.

Bass, L., Clements, P. und Kazman, R. (2003): *Software Architecture in Practice*. Addison-Wesley, Reading, Mass.

Bass, M., Mikulovic, V., Bass, L., Herbsleb, J. und Cataldo, M. (2007): *Architectural Misalignment: An Experience Report*. In Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, S. 17-26.

Bäumer, D. (1998): *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme.* Dissertationsschrift, Universität Hamburg.

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. und Züllighoven, H. (1997): *Framework Development for Large Systems*. Communications of the ACM, 40, (10), S. 52-59.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H. und Tempero, E. (2006): *Understanding the shape of Java software*. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, ACM Press, Portland, Oregon, USA, S. 397-412.

Beck, K. (2005): *Extreme Programming Explained: Embrace Change, Second Edition*. Pearson Education, Upper Saddle River.

Becker-Pechau, P. und Bennicke, M. (2007): *Concepts of Modeling Architectural Module Views for Compliance Checks Based on Architectural Styles*. In Software Engineering and Application (SEA 2007), (Cambridge, Massachusetts).

Becker-Pechau, P., Karstens, B. und Lilienthal, C. (2006): *Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln*. In Software Engineering 2006, (Leipzig), Gesellschaft für Informatik, S. 27-38.

Bennicke, M. und Rust, H. (2004): *Messen im Software-Engineering und metrikbasierte Qualitätsanalyse*. Virtuelles Software Engineering Kompetenz Center, Nr. VISEK/024/D.

Bianchi, A., Caivano, D., Marengo, V. und Visaggio, G. (2001): *Iterative Reengineering of Legacy Functions*. In International Conference on Software Maintenance, (Florence, Italy), IEEE, S. 632-641.

Bianchi, A., Caivano, D., Marengo, V. und Visaggio, G. (2003): *Iterative Reengineering of Legacy Systems*. IEEE Transaction on Software Engineering, 29, (3), S. 225-241.

Binder, R. V. (1999): *Testing object-oriented systems: models, patterns, and tools*. Addison Wesley Longmann Publishing Co., Boston, MA.

Bischofberger, W., Kühl, J. und Löffler, S. (2004): *Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking*. In EWSA 2004, Springer-Verlag Berlin Heidelberg, S. 1-9.

Boisot, M. und Child, J. (1999): *Organizations as Adaptive Systems in Complex Environments: The Case of China*. Organization Science, 10, (3), S. 237-252.

Bonja, C. und Kidanmariam, E. (2006): *Metrics for class cohesion and similarity between methods*. In: Proceedings of the 44th annual Southeast regional conference, ACM Press, Melbourne, Florida, S. 91-95.

Booch, G. (2004): *Object-Oriented Analysis and Design with Applications*. Addison Wesley Longman Publishing Co., Inc.

Booch, G., Rumbaugh, J. und Jacobson, I. (1999): *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc.

Bortz, J. und Döring, N. (2006): *Forschungsmethoden und Evaluation*. Springer Medizin Verlag, Heidelberg.

Briand, L. C., Morasca, S. und Basili, V. R. (1996): *Property-Based Software Engineering Measurement*. IEEE Trans. Softw. Eng., 22, (1), S. 68-86.

Brooks, F. P. (1986): *No Silver Bullet - Essence and Accidents of Software Engineering*. In International Federation of Information Processing (IFIP) Congress '86, (Dublin, Ireland), Elsevier Science Publisher B.V., S. 1069-1076.

Brügge, B. und Dutroit, A. H. (2004): *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson Studium, München.

Buhr, M. (1976): *Philosophisches Wörterbuch*. Verlag das Europäische Buch, Berlin.

Card, D. N. und Glass, R. L. (1990): *Measuring Software Design Quality*. Prentice-Hall, Englewood Cliffs, N.J.

Chidamber, S. R. und Kemerer, C. F. (1994): *A Metrics Suite for Object Oriented Design*. IEEE Trans. Softw. Eng., 20, (6), S. 476-493.

Ciupke, O. (2002): *Problemidentifikation in objektorientierten Softwarestrukturen*. Shaker Verlag, Aachen.

Clark, J. und Holton, D. A. (1994): *Graphentheorie - Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg.

Coad, P. und Yourdon, E. (1994): *OOD: Objektorientiertes Design*. Prentice Hall, München.

Cockburn, A. (2003): *People and Methodologies in Software Development*. Faculty of Mathematics and Natural Sciences, University of Oslo, Norway.

Collberg, C., Myles, G. und Stepp, M. (2004): *An Empirical Study of Java Bytecode Programs*. Department of Computer Science, University of Arizona, Nr. TR 04-11.

Coveney, P. und Highfield, R. (1995): *Frontiers of Complexity, The Search of Order in a Chaotic World*. Ballantine Books Inc., New York.

Darcy, D. P. (2001): *Software Complexity: Integrating the Task Complexity Model and Cognition*. Joseph M. Katz Graduate School of Business, University of Pithsburgh, Pittsburgh.

Darcy, D. P. und Slaughter, S. A. (2005): *The Structural Complexity of Software: An Experimental Test*. IEEE Transaction on Software Engineering, 31, (11), S. 982-995.

Davis, J. S. (1984): *Chunks: A Basis for Complexity Measurement*. Information Processing & Management, 20, (1), S. 119-127.

Deursen, A. v., Hofmeister, C., Koschke, R., Moonen, L. und Riva, C. (2004): *Symphony: View-Driven Software Architecture Reconstruction*. In: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), IEEE Computer Society, S. 122.

Dijkstra, E. W. (1976): *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey.

Dijkstra, E. W. (1968): *The Structure of the T.H.E. Multiprogramming System*. Communications of the ACM, 11, (5), S. 341-346.

DIN9126 (1991): *Beurteilen von Softwareprodukten, Qualtitäsmerkmale und Leitfaden zu deren Verwendung*. Deutsches Institut für Normung.

Dutke, S. (1994): *Mentale Modelle: Konstrukte des Wissens und Verstehens*. Verlag für Angewandte Psychologie, Göttingen.

Ebert, C. (1995): *Complexity traces: an instrument for software project management*. In Fenton, N.E., Whitty, R. and Iizuka, Y. (eds.): Software Quality: Assurance and Measurement - A worldwide Perspective, International Thomson Computer Press, London, S. 166-176.

El-Emam, K., Benlarbi, S. d., Goel, N. und Rai, S. N. (2001): *The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics*. IEEE Transaction on Software Engineering, 27, (7), S. 630-650.

Etzkorn, L., Davis, C. und Li, W. (1998): *A Practical Look at the Lack of Cohesion in Methods Metric*. Journal of Object-Oriented Programming, 11, (5), S. 27-34.

Evans, E. (2004): *Domain Driven Design: Tackling Complexity in the heart of Software*. Addison-Wesley.

Feathers, M. C. (2004): *Working Effectively with Legacy Code*. Pearson Education, Upper Saddle River, NJ.

Fenton, N. E. und Pfleegler, S. L. (1997): *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, Boston.

Flick, U. (2006): *Qualitative Sozialforschung, Eine Einführung*. Rowohlt Taschenbuch Verlag GmbH, Reinbek bei Hamburg.

Floyd, C. (1992a): *Software Development as Reality Construction*. In Floyd, C., Züllighoven, H., Budde, R. and Keil-Slawik, R. (eds.): Software Development and Reality Construction, Springer-Verlag, Berlin, S. 86-100.

Floyd, C. (1992b): *STEPS Projekthandbuch*. University of Hamburg.

Floyd, C. (1993): *STEPS – a methodical approach to PD*. Communications of the ACM, 36, (6), S. 83.

Floyd, C., Reisin, F.-M. und Schmidt, G. (1989): *STEPS to Software Development with Users*. In ESEC'89, (University of Warwick, Coventry, UK), Springer-Verlag, S. 48-64.

Floyd, C. und Züllighoven, H. (2002): *Softwaretechnik*. In Rechenberg, P. and Pomberger, G. (eds.): Informatik-Handbuch, Hanser Verlag, München, Wien, S. 763-790.

Foote, B. und Yoder, J. W. (2000): *Big ball of mud*. In Foote, B. and Rohnert, H. (eds.): Pattern Languages of Program Design, Addison-Wesley, S. 654-692.

Fowler, M. (2003): *Patterns of enterprise application architecture*. Addison-Wesley.

Fowler, M. (2001): *Reducing Coupling*. IEEE Software, 18, (4), S. 102-104.

Fowler, M., Beck, K., Bryant, J., Opdyke, W. und Roberts, D. (2001): *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, Boston.

Frese, M. (1987): *A theory of control and complexity: implications for software design and integration of computer systems into the work place*. In: Psychological issues of human-computer interaction in the work place, North-Holland Publishing Co., S. 313-337.

Frese, M. und Brodbeck, F. C. (1989): *Computer in Büro und Verwaltung. Psychologisches Wissen für die Praxis*. Springer-Verlag, Berlin Heidelberg.

Gamma, E., Helm, R., Johnson, R. und Vlissides, J. (1994): *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garlan, D. (2000): *Software architecture: a roadmap*. In: Proceedings of the Conference on The Future of Software Engineering, ACM Press, Limerick, Ireland, S. 91-101.

Garlan, D., Allen, R. und Ockerbloom, J. (1994): *Exploiting style in architectural design environments*. In: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, ACM Press, New Orleans, Louisiana, United States, S. 175-188.

Garlan, D. und Shaw, M. (1994): *An Introduction to Software Architecture*. Carnegie Mellon University, Nr. CS-94-166.

Gell-Mann, M. (1994): *Das Quark und der Jaguar*. Piper, München.

Glaser, B. G. und Strauss, A. L. (1967): *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing.

Glass, R. L. (2002): *Sorting out software complexity*. Communications of the ACM, 45, (11), S. 19-21.

Gui, G. und Scott, P. D. (2006): *Coupling and cohesion measures for evaluation of component reusability*. In: Proceedings of the 2006 international workshop on Mining software repositories, ACM Press, Shanghai, China, S. 18-21.

Henderson-Sellers, B. (1996): *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

Hess, A., Humm, B. und Voß, M. (2006): *Regeln für serviceorientierte Architekturen hoher Qualität.* Informatik-Spektrum.

Hofmeister, C., Nord, R. und Soni, D. (2000): *Applied Software Architecture*. Addison-Wesley.

Hofmeister, C., Nord, R. und Soni, D. (2005): *Global Analysis: Moving from Software Requirements Specification to Structural Views of the Software Architecture*. IEE Proceedings Software, 152, (4), S. 187-197.

Horstmann, C. S. (2006): *Object Oriented Design and Patterns*. John Wiley & Sons.

IEEE (1990): *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society.

Jacobson, I. (1992): *Object-Oriented Software Engineering*. Addison-Wesley, Reading, Massachusetts.

Jacobson, I., Booch, G. und Rumbaugh, J. (1999): *The Unified Software Development Process*. Addison-Wesley, Reading, Massachusetts.

Kerievsky, J. (2004): *Refactoring to Patterns*. Addison-Wesley, Boston.

Kim, J. S. und Garlan, D. (2006): *Analyzing architectural styles with alloy*. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis, ACM Press, Portland, Maine, S. 70-80.

Kluge, F. (2002): *Etymologisches Wörterbuch der deutschen Sprache*. Walter de Gruyter & Co.

Kluwe, R. H. (1992): *Gedächtnis und Wissen*. In Spada, H. (ed.): Allgemeine Psychologie, Verlag Hans Huber, Bern, S. 115-188.

Knodel, J. und Popescu, D. (2007): *A Comparison of Static Architecture Compliance Checking Approaches*. In: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07), IEEE Computer Society, S. 12.

Kornwachs, K. (2004): *Technik als System - System Engineering und Systemverstehen*. In Kornwachs, K. (ed.): Technik - System - Verantwortung. Reihe Technikphilosophie, Lit Verlag, Münster, S. 593-608.

Koschke, R. und Simon, D. (2003): *Hierarchical Reflexion Models*. In: Proceedings of the 10th Working Conference on Reverse Engineering, IEEE Computer Society, S. 36.

Kruchten, P. (1995): *The 4+1 View Model of Architecture*. IEEE Software 12, 6, S. 42-50.

Kruchten, P. (1999): *Der Rational Unified Process: Eine Einführung*. Addisson-Wesley, Massachusetts.

Lakos, J. (1996): *Large-scale C++ Software Design*. Addison Wesley Longmann Publishing Co., Redwood City, California.

Lamnek, S. (1995): *Qualitative Sozialforschung: Methodologie*. Beltz PVU, Weinheim.

Larman, C. (2004): *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR.

Lehman, M. M. (1980): *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, 68, (9), S. 1060-1076.

Lilienthal, C. (2007): *Architekturstile in der Praxis*. In Informatik 2007, Informatik trifft Logistik, (Bremen, Germany), Lecture Notes in Informatics (LNI), P-109, S. 320-325.

Lilienthal, C. und Züllighoven, H. (1997): *Application-oriented usage quality: the tools and materials approach*. Interactions, 4, (6), S. 35-41.

Lippert, M., Wolf, H. und Roock, S. (2002): *Extreme Programming in Action: Practical Experiences from Real World Projects*. John Wiley & Sons, Inc.

Lippert, M. und Züllighoven, H. (2002): *Using extreme programming to manage high-risk projects successfully*. In: Software quality and software testing in internet times, Springer-Verlag New York, Inc., S. 85-100.

Luhmann, N. (1986): *Ökologische Kommunikation. Kann die moderne Gesellschaft sich auf ökologische Gefährdungen einstellen?* Westdeutscher Verlag, Opladen.

Luhmann, N. (1987): *Soziale Systeme: Grundriß einer allgemeinen Theorie*. Suhrkamp, Frankfurt am Main.

Maaß, S. (1994): *Transparenz - eine zentrale Software-ergonomische Forderung*. Universität Hamburg, Fachbereich Informatik, Nr. 170.

Martin, R. C. (2003): *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education, Upper Saddle River.

Mayrhauser, A. v. und Vans, A. M. (1997): *Program understanding behavior during debugging of large scale software*. In: Papers presented at the seventh workshop on Empirical studies of programmers, ACM Press, Alexandria, Virginia, United States, S. 157-179.

McBride, M. R. (2007): *The software architect*. Communications of the ACM, 50, (5), S. 75-81.

McCabe, T. (1976): *A Complexity Measure*. IEEE Transaction on Software Engineering, SE-2, (4), S. 308-320.

McCabe, T. und Butler, C. W. (1989): *Design Complexity Measurement and Testing*. Communications of the ACM, 32, (12), S. 1415-1425.

McCall, J. A., Richards, P. K. und Walters, G. F. (1977): *Factors in Software Quality*. US Rome Air Development Center, Nr. RADC TR-77-369, Vols I,II,III, NTIS AD/A-049 014,015,055.

Melton, H. und Tempero, E. (2006): *An Empirical Study of Cycles among Classes in Java*. Department of Computer Science, University of Auckland, Nr. UoA-SE-2006-1.

Meyer, B. (1995): *Object success: a manager's guide to object orientation, its impact on the corporation, and its use for reengineering the software process*. Prentice-Hall, Inc.

Miller, G. A. (1956): *The magical number seven plus minus two: Some limits on our capacity for processing informations*. Psychological Review 63, S. 81-97.

Murphy, G. C., Notkin, D. und Sullivan, K. (1995): *Software reflexion models: bridging the gap between source and high-level models*. In: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, ACM Press, Washington, D.C., United States, S. 18-28.

Murphy, G. C., Notkin, D. und Sullivan, K. J. (2001): *Software Reflexion Models: Bridging the Gap between Design and Implementation*. IEEE Transaction on Software Engineering, 27, (4), S. 364-380.

Myers, G. J. (1978): *Composite/Structured Design*. Van Nostrand Reinhold, New York.

Nagappan, N., Ball, T. und Zeller, A. (2006): *Mining Metrics to Predict Component Failures*. In 28th International Conference on Software Engineering (ICSE '06), (Shanghai, China), ACM Press, S. 452-461.

Nagl, M. (1990): *Softwaretechnik: methodisches Programmieren im Großen*. Springer-Verlag, Berlin.

Naur, P. (1985): *Programming as Theory Buildung*. Microprocessing and Microprogramming, 15, S. 253-261.

Neumann, R. (2005): *Identifikation von Softwarequalitätsproblemen anhand historischer Daten des Änderungsmanagements*. Diplomarbeit, Institut für Informatik, Lehrstuhl für Software-Systemtechnik, Brandenburgische Technische Universität Cottbus, Cottbus.

Nikitina, N. (2007): *Bewertung von softwaretechnischen Prinzipien auf der Basis historischer Daten am Beispiel von Eclipse*. Diplomarbeit, MIN-Fakultät, Arbeitsbereich Softwaretechnik, Universität Hamburg, Hamburg.

Noak, A. (2007): *Unified Quality Measures for Clusterings, Layouts, and Orderings of Graphs, and Their Application as Software Design Criteria*. Dissertation, Fakultät für Mathematik, Naturwissenschaften und Informatik, Brandenburgische Technische Universität Cottbus, Cottbus.

Nord, R., Hofmeister, C. und Soni, D. (1999): *Preparing for Change in the Architecture Design of Large Software Systems*. In First Working IFIP Conference on Software Architecture (WICSA'99), (San Antonio, Texas).

Norman, D. A. (1982): *Learning and Memory*. W. H. Freeman & Co, ACM Press.

Oberquelle, H. (1991): *MCI - quo vadis? Perspektiven für die Gestaltung und Entwicklung der Mensch-Computer-Interaktion*. In Ackermann, D. and Ulich, E. (eds.): Software-Ergonomie '91. Benutzerorientierte Software-Entwicklung, Teubner, Stuttgart, S. 9-24.

Oetinger, B. v., Ghyczy, T. v. und Bassford, C. (2005): *Clausewitz, Strategie denken*. Carl Hanser Verlag, München, Wien.

Parnas, D. L. (1979): *Designing Software for Ease of Extension and Contraction*. In Hoffmann, D.M. and Weiss, D.M. (eds.): Software fundamentals: Collected Papers by David L. Parnas, Pearson, Saddle River, S. 269-286.

Parnas, D. L. (1974): *On a 'Buzzword': Hierarchical Structure*. In IFIP Congress 74, North-Holland Publishing Company, S. 336-339.

Parnas, D. L. (1972): *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, 15, (12), S. 1053-1058.

Parnas, D. L. (1994): *Software Aging*. In 16th International Conference on Software Engineering, (Sorento Italy), IEEE Press, S. 279-287.

Parnas, D. L., Clements, P. und Weiss, D. M. (1985): *The Modular Structure of Complex Systems*. IEEE Transaction on Software Engineering, SE-11, (3), S. 259-266.

Perry, D. E. und Wolf, A. L. (1992): *Foundations for the Study of Software Architecture*. ACM Sigsoft, Software Engineering Notes, 17, (4), S. 40-52.

Poels, G. und Dedene, G. (1997): *Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties"*. IEEE Transaction on Software Engineering, 23, (3), S. 190-195.

Prnjat, O. und Sacks, L. (2001): *Complexity Measurements of the Inter-Domain Management System Design*. In: Proceedings of the 9th IEEE International Conference on Networks, IEEE Computer Society, S. 2-7.

Reussner, R. und Hasselbring, W. (eds.) (2006): *Handbuch der Software-Architektur*. dpunkt.verlag, Heidelberg.

Riehle, D. und Züllighoven, H. (1995): *A pattern language for tool construction and integration based on the tools and materials metaphor*. In: Pattern languages of program design, ACM Press/Addison-Wesley Publishing Co., S. 9-42.

Riel, A. J. (1996): *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc.

Roock, S. und Lippert, M. (2004): *Refactorings in großen Softwareprojekten*. dpunkt.verlag, Heidelberg.

Sangal, N., Jordan, E., Sinha, V. und Jackson, D. (2005): *Using dependency models to manage complex software architecture*. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, ACM Press, San Diego, CA, USA, S. 167-176.

Savernik, L. (2007): *Entwicklung einse automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen*. In Software Engineering 2007 - Beiträge zu den Workshops, (Hamburg), Lecture Notes in Informatics (LNI), S. 357-360.

Scharping, A. (2006): *Architekturvereinbarungen des JCommSys*. Bachelorarbeit, MIN-Fakultät, Arbeitsbereich Softwaretechnik, Universität Hamburg, Hamburg.

Scharping, A. (2007): *Automatisierte Prüfung von Architekturregeln zur Entwicklungszeit*. Diplomarbeit, MIN-Fakultät, Arbeitsbereich Softwaretechnik, Universität Hamburg, Hamburg.

Schreier, O. (2006): *Einsatz von Softwaremaßen zur Qualitätssicherung*. Diplomarbeit, MIN-Fakultät, Arbeitsbereich Softwaretechnik, Universität Hamburg, Hamburg.

Siedersleben, J. (2004): *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, Heidelberg.

Simon, F. (2001): *Meßbasierte Qualitätssicherung*. Dissertation, Fakultät für Mathematik, Naturwissenschaften und Informatik, Brandenburgische Technische Universität Cottbus, Cottbus.

Simon, F. und Meyerhoff, D. (2002): *OO-Metriken zeigen grosse Qualitätspotenziale in komplexen Softwaresystemen*. Objektspektrum, 06, S. 28-35.

Simon, F., Seng, O. und Mohaupt, T. (2006): *Code-Quality-Management*. dpunkt.verlag, Heidelberg.

Simon, H. A. (1994): *Die Wissenschaften vom Künstlichen*. Springer-Verlag, Wien.

Soni, D., Nord, R. und Hofmeister, C. (1995): *Software Architecture in Industrial Applications*. In International Conference on Software Engineering, (Seattle, Washington), ACM, S. 196-207.

Stevens, S. S. (1946): *On the theory of Scales and Measurement*. Science, 103, S. 677-680.

Storey, M.-A. D., Fracchia, F. D. und Mueller, H. A. (1999): *Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration*. Journal of Software Systems, 44, (3), S. 171-185.

Strauss, A. L. und Corbin, J. (1998): *Basics of Qualitative Research: Techniques and Procedures for developing Grounded Theory*. Sage Publications.

Strube, G., Becker, B., Freksa, C., Hahn, U., Opwis, K. und Palm, G. (1996): *Wörterbuch der Kognitionswissenschaft*. Klett-Cotta, Stuttgart.

Tanenbaum, A. S. (1995): *Verteilte Betriebssysteme*. Prentice Hall, München.

Waldrop, M. (1992): *Complexity: The Emerging Science at the Edge of Order and Chaos*. Simon and Schuster, New York.

Wallnau, K., Clements, P., Morris, E. und Krut, R. (1996): *The Gadfly: An Approach to Architectural-Level System Comprehension*. In: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96), IEEE Computer Society, S. 178.

Wegner, P. (1990): *Concepts and paradigms of object-oriented programming*. SIGPLAN OOPS Messenger, 1, (1), S. 7-87.

Wetzel, I., Klischewski, R., Krabbel, A. und Lilienthal, C. (1998): *Kooperation für Software für Kooperation - Erfahrungen aus einem partizipativen Softwaretechnikprojekt*. In: Informatik und Ausbildung, GI-Fachtagung 98, Informatik und Ausbildung, Springer-Verlag, S. 73-81.

Weyuker, E. J. (1988): *Evaluating Software Complexity Measures*. IEEE Transaction on Software Engineering, 14, (9), S. 1357-1365.

Wirfs-Brock, R. und McKean, A. (2002): *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education.

Woodfield, S. N. (1979): *An experiment on unit increase in problem complexity*. IEEE Transactions on Software Engineering, SE-5, (2), S. 76-79.

Yourdon, E. (1992): *Re-engineering, Restructuring, Reverse Engineering*. Software Reeingineering, IEEE Computer Society Press, Los Alamitos, S. 26-33.

Zemanek, H. (1992): *Das geistige Umfeld der Informationstechnik*. Edition SEL-Stiftung.

Zhao, J. (1998): *On assessing the complexity of software architectures*. In: Proceedings of the third international workshop on Software architecture, ACM Press, Orlando, Florida, United States, S. 163-166.

Zimmermann, T. und Nagappan, N. (2006): *Predicting Subsystem Failures using Dependency Graph Complexities*. Microsoft Research, Nr. MSR-TR-2006-126.

Züllighoven, H. (2005): *Object-Oriented Construction Handbook*. Morgan Kaufmann Publishers, San Francisco.

Züllighoven, H. und Raasch, J. (2006): *Softwaretechnik*. In Rechenberg, P. and Pomberger, G. (eds.): Informatik-Handbuch, Hanser Verlag, München, Wien, S. 837-879.

Zuse, H. (1998): *A Framework of Software Measurement*. Walter de Gruyter & Co., Berlin.

Zuse, H. (1990): *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., Berlin.

Index

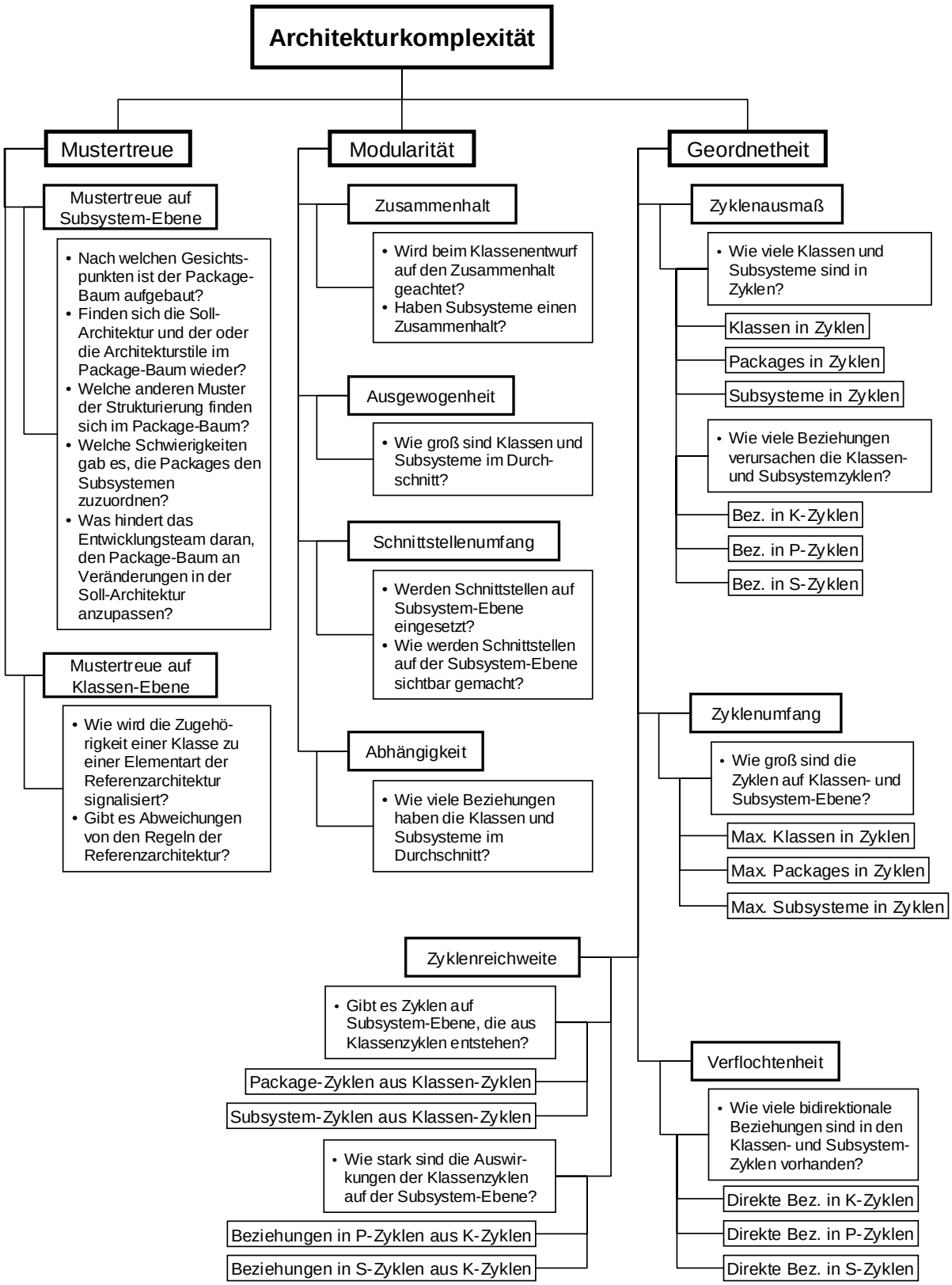
Architekturkomplexität
Mustertreue
Mustertreue auf Subsystem-Ebene
• Nach welchen Gesichtspunkten ist der Package-Baum aufgebaut?
• Finden sich die Soll-Architektur und der oder die Architekturstile im Package-Baum wieder?
• Welche anderen Muster der Strukturierung finden sich im Package-Baum?
• Welche Schwierigkeiten gab es, die Packages den Subsystemen zuzuordnen?
• Was hindert das Entwicklungsteam daran, den Package-Baum an Veränderungen in der Soll-Architektur anzupassen?
Mustertreue auf Klassen-Ebene
• Wie wird die Zugehörigkeit einer Klasse zu einer Elementart der Referenzarchitektur signalisiert?
• Gibt es Abweichungen von den Regeln der Referenzarchitektur?
Modularität
Zusammenhalt
• Wird beim Klassenentwurf auf den Zusammenhalt geachtet?
• Haben Subsysteme einen Zusammenhalt?
Ausgewogenheit
• Wie groß sind Klassen und Subsysteme im Durchschnitt?
Schnittstellenumfang
• Werden Schnittstellen auf Subsystem-Ebene eingesetzt?
• Wie werden Schnittstellen auf der Subsystem-Ebene sichtbar gemacht?
Abhängigkeit
• Wie viele Beziehungen haben die Klassen und Subsysteme im Durchschnitt?
Geordnetheit
Zyklenausmaß
• Wie viele Klassen und Subsysteme sind in Zyklen?
Klassen in Zyklen
Packages in Zyklen
Subsysteme in Zyklen
• Wie viele Beziehungen verursachen die Klassen- und Subsystemzyklen?
Bez. in K-Zyklen
Bez. in P-Zyklen
Bez. in S-Zyklen
Zyklenumfang
• Wie groß sind die Zyklen auf Klassen- und Subsystem-Ebene?
Max. Klassen in Zyklen
Max. Packages in Zyklen
Max. Subsysteme in Zyklen
Zyklenreichweite
• Gibt es Zyklen auf Subsystem-Ebene, die aus Klassenzyklen entstehen?
Package-Zyklen aus Klassen-Zyklen
Subsystem-Zyklen aus Klassen-Zyklen
• Wie stark sind die Auswirkungen der Klassenzyklen auf der Subsystem-Ebene?
Beziehungen in P-Zyklen aus K-Zyklen
Beziehungen in S-Zyklen aus K-Zyklen
Verflochtenheit
• Wie viele bidirektionale Beziehungen sind in den Klassen- und Subsystem-Zyklen vorhanden?
Direkte Bez. in K-Zyklen
Direkte Bez. in P-Zyklen
Direkte Bez. in S-Zyklen

Tabelle A-1: Fallstudien, Größe, Start, Vorgehen und Architekturstile

Nr	LOC	RLOC	Start	A	I	Architekturstil
1	26.051	10.679	2005	X		**WAM-Architektur**
						Schichtenarchitektur ohne Schnittstellen
2	33.935	13.219	2000	X	X	**WAM-Architektur**
3	36.838	16.314	2003	X	X	**Projekteigene Referenzarchitektur**
4	64.156	21.591	2003	X	X	**WAM-Architektur**
						Erweiterte Schichtenarchitektur ohne S.
5	53.087	22.957	2001	X		**WAM-Architektur**
6	47.167	23.478	2002	X	X	Schichtenarchitektur ohne Schnittstellen
7	108.057	33.651	2001	X	X	**WAM-Architektur**
8	83.709	36.671	2003	X		*Subsystemarchitektur*
9	99.405	43.969	2003	X		**WAM-Architektur**
10	137.418	49.428	1999	X	X	**WAM-Architektur**
11	121.137	65.284	2003	X		Schichtenarchitektur ohne Schnittstellen
12	130.316	70.266	2001	X		**Projekteigene Referenzarchitektur**
13	169.567	74.344	2003	X		Schichtenarchitektur ohne Schnittstellen
14	188.113	93.593	2003	X		Schichtenarchitektur ohne Schnittstellen
15	221.284	112.675	2001	X		Schichtenarchitektur ohne Schnittstellen
16	238.637	120.964	2003	X		Schichtenarchitektur ohne Schnittstellen
17	269.609	133.095	2002	X		**Projekteigene Referenzarchitektur**
						Schichtenarchitektur ohne Schnittstellen
18	294.414	143.101	2000	X		Schichtenarchitektur ohne Schnittstellen
19	~500.000	~250.000	2002		X	Schichtenarchitektur mit Schnittstellen
20	789.938	344.911	2001	X	X	*Subsystemarchitektur*
21	1465.441	629.674	2003	X	X	**Quasar-Architektur**
						Subsystemarchitektur
22	2.919.550	1.296.455	1998	X	X	**Projekteigene Referenzarchitektur**
						Erweiterte Schichtenarchitektur mit S.
23	2.869.566	1.384.260	2001	X		Schichtenarchitektur mit Schnittstellen
24	~14 Mio	~8 Mio	2002		X	Erweiterte Schichtenarchitektur mit S.

Legende zu Tabelle A-1:

Referenzarchitektur
Schichtenarchitektur
Subsystemarchitektur

A – Architekturanalyse, I – Interview