Componentware:

Methodik des evolutionären Architekturentwurfs

Andreas Rausch

Institut für Informatik
der Technischen Universität München

Componentware:

Methodik des evolutionären Architekturentwurfs

Andreas Rausch

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Rudolf Bayer, Ph.D
Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Hon.-Prof. Dr. Ernst Denert

Die Dissertation wurde am 28.6.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.10.2001 angenommen.

Für meine Familie

Kurzfassung

Heutige Softwaresysteme sind äußerst komplex. Erfahrungsgemäß können sie von einzelnen Softwareingenieuren kaum mehr vollständig erfasst werden. Die Entwicklung eines Systems findet meist in einem hochgradig dynamischen Umfeld statt. Der iterative, evolutionäre Entwurf einer Softwarearchitektur und frühes Prototyping können helfen, Risiken der Softwareentwicklung entscheidend zu minimieren. Die Softwarearchitektur beschreibt dabei eine geeignete Zerlegung des Systems in Komponenten und die Abhängigkeiten zwischen diesen Komponenten.

In dieser Arbeit wird eine Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme entwickelt. Sowohl textbasierte als auch UML-basierte, grafische Spezifikationstechniken für Komponenten, deren Abhängigkeiten und komponentenbasierte Systeme werden erarbeitet. Mit diesen Spezifikationstechniken kann die Softwarearchitektur eines komponentenbasierten Systems vollständig und präzise beschrieben werden. Nach der Veränderung einzelner Komponentenspezifikationen in Rahmen eines evolutionären Entwicklungsschrittes können die Abhängigkeiten zwischen den Komponenten erneut auf ihre Gültigkeit hin überprüft werden. Die Integration der einzelnen Komponentenspezifikationen zu einem konsistenten Architekturmodell erfolgt so bereits auf der Spezifikationsebene.

Grundlage der erarbeiteten Methodik ist ein Systemmodell, das die Menge aller komponentenbasierter Systeme charakterisiert. Die Komponenten eines solchen Systems werden nebenläufig und verteilt ausgeführt. Sie kommunizieren über asynchrone Nachrichten, greifen lesend und schreibend auf Attribute von Komponenten zu und verändern die Struktur des komponentenbasierten Systems.

Das Systemmodell ist die Basis für die formale Fundierung der erarbeiteten Spezifikationstechniken. Über eine prädikatenbasierte formale Semantik wird einer Spezifikation eine Menge von Eigenschaften zugewiesen. Ein komponentenbasiertes System ist genau dann eine korrekte Implementierung der Spezifikation, wenn es diese Eigenschaften besitzt.

Die Konzeption und prototypische Realisierung einer durchgängigen und weitreichenden Werkzeugunterstützung garantieren die effektive und praxisorientierte Umsetzung der erarbeiteten Methodik. Die vorgestellten Konzepte reichen von der Architekturspezifikation über die Konsistenz- und Integrationsüberprüfung bis zu Prototypengenerierung und Testverfahren.

// Danksagung

Für die äußerst angenehme und produktive Atmosphäre, die ich in den letzten Jahren genießen durfte, und in deren Rahmen diese Dissertation entstanden ist, möchte ich Professor Dr. Manfred Broy und allen Mitarbeitern seines Lehrstuhls ganz besonders danken.

Weiterhin möchte ich mich bei Professor Dr. Manfred Broy für die Ermutigung, diese Arbeit zu beginnen, und für die Unterstützung bei ihrer Erstellung bedanken. Professor Dr. Ernst Denert gebührt mein Dank für die Übernahme des zweiten Gutachtens. Ganz besonders möchte ich beiden für die maßgebliche Beeinflussung meines wissenschaftlichen Werdegangs danken.

Ohne die interessante und konstruktive Zusammenarbeit mit allen Kollegen in dem Forschungsprojekt FORSOFT, insbesondere in den Teilprojekten A1 und ZEN, wäre die vorliegende Arbeit in dieser Form nicht möglich gewesen. Für viele interessante Gespräche und Diskussionen möchte ich mich insbesondere bei meinen Kollegen Dr. Klaus Bergner, Marc Sihling und Alexander Vilbig aus dem Forschungsprojekt A1 sowie Michael Gnatz, Frank Marschall, Gerhard Popp und Wolfgang Schwerin aus dem Forschungsprojekt ZEN bedanken.

Für das aufmerksame Durchlesen und die vielen Kommentare zu Vorversionen dieser Arbeit möchte ich Dr. Klaus Bergner, Professor Dr. Manfred Broy, Michael Gnatz, Frank Marschall, Gerhard Popp und Dr. Bernhard Schätz danken. Für die Hilfe bei der Umsetzung und Realisierung des Werkzeugprototypen DesignIt gebührt mein Dank Lucien Hoogendoorn, Daire Kivlehan, Karen Lavery und Sylvia Pohlmann.

Nicht zuletzt danke ich allen meinen Bekannten, Freunden und vor allem meiner Familie für die Geduld und Rücksicht, die sie während der Erstellung dieser Arbeit mit mir hatten.

Inhaltsverzeichnis

1 EINLEITUNG 1

1.1 Einführung 2

1.2 Ziele und Ergebnisse 5

1.3 Inhalt und Aufbau 6

1.4 Verwandte Arbeiten 8

2 EVOLUTIONÄRE METHODISCHE SOFTWAREENTWICKLUNG 11

2.1 Grundbausteine der methodischen Softwareentwicklung 12

2.2 Methodisches Rahmenwerk dieser Arbeit 20

2.3 Modellbasierte Entwicklung und Softwarearchitekturmodelle 26

2.4 Prozessmusterbasierter Ansatz und der evolutionäre Entwurf 33

2.5 Zusammenfassung 45

3 EVOLUTIONÄRER ENTWURF KOMPONENTENBASIERTER SYSTEME 47

3.1 Systemmodellbasierte formale Semantik 48

3.2 Evolution von Dokumentenmengen und Spezifikationen 49

3.3 Evolution als Vergröberung und Verfeinerung 51

3.4 Prädikatenbasierte formale Semantik 54

3.5 Widerspruchsfreiheit und Angemessenheit von Spezifikationen 60

3.6 Varianten der Evolution von Spezifikationen 62

3.7 Zusammenfassung 65

4 DER PAUSENPLANER – EIN EINFACHES ANWENDUNGSBEISPIEL 67

4.1 Motivation und Hintergrund des Pausenplaners 68

4.2 Erste Kundenanforderungen und Systemvision 68

4.3 Analyse der Anwendungsfälle des Pausenplaners 70

4.4 Identifikation der Komponenten des Pausenplaners 72

4.5 Zusammenfassung 75

5 GRUNDLAGEN KOMPONENTENBASIERTER SYSTEME 77

5.1 Beispielablauf in einem komponentenbasierten System 78

5.2 Grundlegende Konzepte komponentenbasierter Systeme 80

5.3 Zeit und Verhalten komponentenbasierter Systeme 84

5.4 Verhalten und Komposition von Komponenten 86

5.5 Von flachen zu hierarchischen Komponenten und Systemen 89

5.6 Zusammenfassung 99

6 ARCHITEKTURSPEZIFIKATION KOMPONENTENBASIERTER SYSTEME 101

6.1 Beispiel einer Architekturspezifikation 102

6.2 Syntax von Architekturspezifikationen 106

6.3 Semantik von Architekturspezifikationen 115

6.4 UML-basierte grafische Spezifikationstechnik 126

6.5 Zusammenfassung 133

7 ERWEITERTE SPEZIFIKATIONEN FÜR DEN EVOLUTIONÄREN ENTWURF 135

7.1 Evolution einer Architekturspezifikation – Ein Beispiel 136

7.2 Beispiel einer erweiterten Architekturspezifikation 139

7.3 Erweiterte Syntax von Architekturspezifikationen 145

7.4 Semantik von erweiterten Architekturspezifikationen 146

7.5 Erweiterte UML-basierte grafische Spezifikationstechnik 150

7.6 Zusammenfassung 153

8 WERKZEUGUNTERSTÜTZUNG 155

8.1 Konzept einer umfassenden Werkzeugunterstützung 156

8.2 Modellierungs- und Spezifikationswerkzeuge 158

8.3 Konsistenz- und Integrationsüberprüfung 160

8.4 Generierung von Programmcode 162

8.5 Ausführungs- und Testumgebung 164

8.6 Versions- und Migrationsunterstützung 167

8.7 Zusammenfassung 169

9 ZUSAMMENFASSUNG UND AUSBLICK 171

LITERATURVERZEICHNIS 175

ABBILDUNGSVERZEICHNIS 187

DEFINITIONSVERZEICHNIS 189

WEITERE INFORMATIONEN IM INTERNET 191

1 Einleitung

Moderne Technologien, neue Beschreibungstechniken, verschiedene Vorgehensmodelle – in den letzten zehn Jahren haben diese Themen den Wandel in der Informatik entscheidend mitbestimmt. Anfang der 90'er Jahre setzte der Siegeszug objektorientierter Softwareentwicklung einen neuen technologischen Umbruch mit einer enormen Eigendynamik in Gang. Ausgehend von dem breiten Erfolg objektorientierter Programmiersprachen entwickelten sich Client/Server-Programmierung, Drei-Schichten-Architekturen und verschiedene Middleware-Ansätze. Inzwischen werden die ersten Application Server, objektorientierten Transaktionsmonitore und Embedded Workflow Engines im industriellen Umfeld eingesetzt. Ein Ende dieser technologischen Innovationswelle ist bis heute nicht in Sicht. Stetig kommen neue Technologien auf den Markt und werden mehr oder weniger erfolgreich in der Industrie eingesetzt.

Nahezu zeitgleich mit dem Aufstieg der objektorientierten Programmierung erschienen die ersten Arbeiten im Bereich der Methodik objektorientierter Analyse und Design. Die bekanntesten stammen von den Hauptautoren der Unified Modeling Language (UML) [BJR98, RJB98]: Grady Booch [Booc94], Jim Rumbaugh [RBP+91] und Ivar Jacobson [Jaco92]. Darüber hinaus haben aber auch eine Vielzahl anderer Autoren maßgeblich die Entwicklungen der ersten objektorientierten Methoden bestimmt, wie zum Beispiel Bertrand Meyer [Meye88], Sally Shlaer und Steve Mellor [SM89], Peter Coad und Ed Yourdon [CY90, CY91], Rebecca Wirfs-Brock [WWW90], Derek Coleman [Cole93] sowie James Martin und Jim Odell [MO92], um nur einige von ihnen zu nennen.

Aus diesen Aktivitäten entstand zuerst eine Fülle unterschiedlicher methodischer Ansätze sowie Beschreibungs- und Modellierungstechniken. Dieser „Wildwuchs“ konvergierte in der Entwicklung eines internationalen Standards für objektorientierte Modellierung, der UML. 1997 wurde die Version 1.1 der UML von der Object Management Group (OMG) standardisiert [OMG97]. Die neueste Version des UML Standards ist die Version 1.3 [OMG00a]. Die Vorbereitungen zur Standardisierung der Version 2.0 sind bereits angelaufen [OMG01a].

Zunächst blieb die Frage offen, wie dieser Standard anzuwenden sei. Denn im Gegensatz zu ihren Vorgängern ist die UML keine Methode. Sie definiert nur Notation und Semantik der Modellierungselemente. Mit etwas Verzögerung begannen die Initiatoren der UML unter der Federführung von Ivar Jacobson mit den Arbeiten an einem Vorgehensmodell. Das Ergebnis ist der Unified Software Development Process [JBR99, Kruc00]. Parallel dazu entstanden neue Vorgehensmodelle und vorhandene wurden weiter entwickelt, wie zum Beispiel das V-Modell '97 [DW99], der Catalysis-Ansatz [DW98], der Object Engineering Process [OOSE01] und die Ansätze der Process Patterns Community [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b]. Zwar ist die Standardisierung von Vorgehensmodellen problematischer als die von Notation und Semantik und auch nur bedingt sinnvoll, da die meisten Unternehmen ihre spezifischen Vorgehensmodelle definieren wollen, dennoch zeichnet sich auch hier eine gewisse Konvergenz ab.

Mit Hilfe von zwei neuen Konzepten versuchen die Autoren den bekannten Problemen klassischer Verfahren zu begegnen: Eine iterative und inkrementelle Vorgehensweise soll durch frühzeitige Rückkopplung Risiken minimieren. Die anwendungsfallgetriebene und architekturzentrierte Modellierung soll zu einer adäquaten und stabilen Architektur führen.

Ein Grund für die Probleme mit dem Wasserfallprozess [Royc70] ist beispielsweise die Annahme, dass alle Anforderungen zu Beginn des Projektes bekannt sind und stabil bleiben. In der Praxis ergeben sich viele Erkenntnisse und Änderungen erst sehr spät, während der Implementierung oder der Systemintegration. Dies führt regelmäßig dazu, dass Projekte bis zur Integration gut im Plan liegen, danach aber plötzlich unvorhergesehene Probleme auftreten, die zu Verzögerungen, Kostenexplosionen und Qualitätsproblemen führen.

Beim iterativen und inkrementellen Vorgehen wird versucht das System in mehreren Inkrementen zu entwickeln, anstatt es „in einem Rutsch" zu erstellen. Jedes Inkrement wird in einer oder mehreren Iterationen erstellt. Bei dieser Vorgehensweise, die wir auch als evolutionäre Vorgehensweise bezeichnen, wird von vornherein angenommen, dass die Anforderungen unvollständig sind, sich während des Projektes teilweise ändern können und durch die gewonnen Erkenntnisse im Laufe der Zeit vervollständigt werden. Änderungen von Anforderungen und deren Auswirkungen werden dabei kontrolliert und nicht chaotisch in das Projekt eingebracht. Das Management erhält so eine sehr viel feinere Steuerung des Ablaufes als dies beispielsweise mit dem Wasserfallmodell möglich ist.

Das zweite Schlüsselkonzept neben dem evolutionären Vorgehen ist die anwendungsfallgetriebene und architekturzentrierte Modellierung. Anhand von konkreten Anwendungsfällen werden ein klares Grundverständnis der Anforderungen und ein erstes stabiles Grundgerüst erarbeitet. Dieses anwendungsspezifische Grundgerüst wird iterativ in weiteren Inkrementen zu einer adäquaten und stabilen Softwarearchitektur vervollständigt.

Ergebnis ist eine Softwarearchitektur, die einen Rahmen für die Abwicklung des Projektes bildet. Die Architektur muss klar formuliert und stabil sein, um einen festen Halt zu geben. Andererseits muss sie so flexibel sein, dass man auf neue Anforderungen und Erkenntnisse im Laufe des Projektes angemessen reagieren kann, ohne dabei die Architektur grundlegend ändern zu müssen.

Die Modularität der Softwarearchitektur beeinflusst dabei entscheidend die Möglichkeiten, das System und die Architektur evolutionär weiter zu entwickeln. Komponentenmodelle und Schnittstellenkonzepte, die Basis von Componentware, sind essentielle Faktoren einer modularen Softwarearchitektur, die einen stabilen und erweiterbaren Rahmen bieten soll. Somit ist ein wesentlicher Schlüsselfaktor einer erfolgreichen Softwareentwicklung der evolutionäre Architekturentwurf komponentenbasierter Softwaresysteme.

1.1 Einführung

Kernthema der Arbeit ist es, eine möglichst durchgängige Methodik für einen zentralen aber kleinen Ausschnitt der Softwareentwicklung bereit zu stellen – den Architekturentwurf komponentenbasierter Systeme. Wesentliche Elemente dieser Methodik sind die evolutionäre Modellierung von Softwarearchitekturen und eine frühzeitige Rückkopplung durch Architekturprototypen. Somit steht im Zentrum der Arbeit, drei aktuelle Themen zu integrieren: Componentware, Softwarearchitekturen und ein evolutionäres Vorgehensmodell.

Zielsetzung des Softwarearchitekturentwurfs ist es, einen stabilen und gut dokumentierten Rahmen zu finden, der Flexibilität und Erweiterbarkeit des Systems unterstützt und dabei Performanz und Zuverlässigkeit garantiert. Das Ergebnis ist ein abstraktes Modell des zu entwickelnden Systems. Häufig wird dieses Architekturmodell mit Hilfe von Diagrammen und zusätzlichem erklärendem Prosatext beschrieben. Bei den Diagrammen wird überwiegend eine standardisierte Notation verwendet, wie zum Beispiel die UML. Das Architek-

turmodell spezifiziert auf einer abstrakten Ebene mindestens die statische Komponentenstruktur des zu erstellenden Systems und die dynamischen Interaktionen zwischen diesen Komponenten. Es repräsentiert damit den grundlegenden Bauplan eines komponentenbasierten Anwendungssystems [BMR+96].

An vielen Stellen würde man sich ein tieferes und formaleres Verständnis von Softwarearchitekturen und deren Beschreibungen wünschen. Dies würde eine frühere Risikominimierung und weitgehend automatisierte Test- und Qualitätssicherungsverfahren ermöglichen, wie zum Beispiel durch die Generierung von Prototypen und deren kontrollierte Ausführung in Testumgebungen.

Im Vergleich zum Verständnis von Softwarearchitekturen und deren Beschreibungen ist aber die Unkenntnis um den Prozess des Architekturentwurfs selbst wesentlich drastischer: Beim Entwurf steht der Softwarearchitekt typischerweise vor einem sehr komplexen Problem. Er beherrscht die Sprache, um die Lösung zu kommunizieren und kennt die Form, in der er die Lösung beschreiben soll. Jedoch sind ihm weder die Lösung noch der Weg zur Lösung – der Prozess – bewusst. Die für die Umsetzung des Lösungsweges notwendigen Hilfsmittel und Werkzeuge sind ihm ebenfalls noch nicht bekannt.

Das Wesen des Architekturentwurfs ist der Akt der Modellbildung. Dazu vergegenwärtigt sich der Softwarearchitekt die Anforderungen an das System. Diese liegen meist in Form eines unvollständigen und eventuell inkonsistenten Analysemodells vor. Aus seinen Erfahrungen, der Literatur, dem Gespräch mit Kollegen oder beliebigen anderen Quellen sammelt der Architekt Modelle oder Teilmodelle, die eine erfolgreiche Lösung für eine ähnliche Problemstellung geboten haben. Mit einer Portion Kreativität und Intuition entwirft er daraus ein neues Architekturmodell und dokumentiert dieses entsprechend. Der Architekturentwurf wird also im wesentlichen durch vier Faktoren beeinflusst: Wissen, Erfahrung, Kreativität und Intuition.

Hat der Architekt nun ein erstes Architekturmodell gefunden, so wird dieses auf Herz und Nieren getestet, beispielsweise mit Hilfe von Testszenarien, Anwendungsfällen, Reviews oder Prototypen. Hierbei zeigen sich normalerweise noch viele Schwächen des Modells. Der Softwarearchitekt verbessert dann das Modell an den entsprechenden Stellen so lange, bis es alle Tests besteht. Sollte dieser Zustand nicht in einer bestimmten Zeit erreicht werden, so wird entweder eine andere Architektur in Betracht gezogen, oder die Anforderungen an das System müssen geändert werden.

Der Architekturentwurf ist somit nicht ein linearer Prozess, sondern eine Folge von Entwurfsiterationen, an deren Ende jeweils eine verbesserte Version des Modells steht. Im Gegensatz zur Vorstellung der iterativen und inkrementellen Vorgehensweise moderner Prozessmodelle ist der Architekturentwurf zwar iterativ, aber nicht inkrementell.

Ein inkrementeller Entwurf würde die schrittweise Erweiterung einer unvollständigen Anfangslösung vorsehen. Die Anfangslösung wird dabei nur erweitert, bestehendes aber nicht verändert. Am Ende einer Iteration steht die Frage nach der Vollständigkeit im Mittelpunkt: Erfüllt die Lösung alle Anforderungen oder muss ein weiteres Inkrement hinzugefügt werden?

Tatsächlich ist aber zu berücksichtigen, dass sich die Anforderungen an das System ändern und meist nicht vollständig präzisierbar sind. Deshalb braucht der Softwarearchitekt die Freiheit, mit jeder neuen Iteration das existierende Modell ohne hemmende Einschränkungen ver-

ändern zu dürfen. Die iterative Weiterentwicklung des Modells ist somit keine inkrementelle Erweiterung, sondern eine evolutionäre Verbesserung.

Diese Art des Architekturentwurfs bezeichnen wir als „evolutionär“: Beim evolutionären Architekturentwurf wird im Rahmen eines Evolutionsschrittes ein bestimmter Teil des Architekturmodells verändert. Am Ende dieser Iteration stellt sich die Frage nach der Widerspruchsfreiheit und der Angemessenheit:

- Widerspruchsfreiheit: Ist das weiter entwickelte Modell syntaktisch und semantisch in sich konsistent und kann eine effiziente Implementierung realisiert werden?
- Angemessenheit: Erfüllt das weiter entwickelte Modell die Anforderungen an das System besser als das ursprüngliche Modell?

Sowohl Industrie als auch Forschung haben für diese Fragestellungen nur unzureichende Konzepte und Lösungen. Beispielsweise wird in der industriellen Praxis die syntaktische Konsistenz eines Modells mit Hilfe von CASE-Tools validiert. Meist steht aber ein vollständiges Modell nicht zur Verfügung. Die Validierung der Konsistenz kann deshalb erst bei der Integration der einzelnen Komponenten durchgeführt werden.

Der Frage, ob ein weiter entwickeltes Modell den Anforderungen besser als das ursprüngliche Modell genügt, versucht man in der Praxis erst nach der Integration durch aufwendige Testverfahren gerecht zu werden. Diese Verfahren sind aber sehr kostspielig. Außerdem werden Fehler dadurch erst spät entdeckt, was wiederum zu zusätzlichen Kosten führt.

In der Forschung gibt es seit längerer Zeit Ansätze, diesen Zustand zu verbessern. Die Grundidee ist meist ähnlich: Mit Hilfe semantisch fundierter Beschreibungstechniken kann die syntaktische und semantische Konsistenz der Modelle garantiert werden. Formale Verfeinerungsbegriffe ermöglichen es, die Modelle syntaktisch zu verändern. Dabei bleiben die ursprünglichen semantischen Eigenschaften des Modells erhalten, und zusätzliche Eigenschaften können hinzugefügt werden. Das Problem dabei ist aber, dass die ursprünglichen Eigenschaften des Modells bewusst verändert werden sollen, da eben diese Eigenschaften teilweise stören. Wie oben ausgeführt, will ein Softwarearchitekt beim Entwurf nicht das Modell verfeinern, sondern er will es evolutionär verbessern und weiter entwickeln. Ein weiteres Problem ist, dass die Formalisierung der Beschreibungen und die damit verbundene Komplexität den Einsatz in der Praxis erheblich erschweren. So werden diese Verfahren für industrielle Softwaresysteme nur äußerst selten eingesetzt.

Der Architekt benötigt pragmatische Beschreibungstechniken und Methoden, mit denen er Softwarearchitekturen effizient und hinreichend präzise beschreiben kann. Diese Spezifikationen sollten möglichst ausführbar sein, oder es sollten zumindest große Teile ausführbarer Programme daraus generierbar sein. Damit wird ein frühes und effektives Testen ermöglicht. Werden Teile dieser Spezifikationen auf Grund des evolutionären Architekturentwurfs verändert und weiter entwickelt, so sollten entsprechende Mechanismen weitreichende Informationen über die Auswirkungen der Veränderungen bereitstellen. So lässt sich konsequent die Widerspruchsfreiheit und die Angemessenheit des Architekturmodells überprüfen.

Eine derartige Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme kann den Softwarearchitekten bei der täglichen Arbeit nachhaltig unterstützen. Allerdings verspricht diese Methodik noch weitreichendere Vorteile und Potentiale: „Time to Market“ ist einer der wesentlichen Erfolgsfaktoren heutiger Softwareprodukte. Der Konkurrenzdruck zwingt die Unternehmen, in immer kürzeren Abständen neue Produktversionen mit

zusätzlicher Funktionalität auf den Markt zu bringen. Deshalb ist es essentiell, eine effektive und pragmatische methodische Unterstützung der evolutionären Weiterentwicklung von Anwendungssystem zur Verfügung zu haben. Ein Unternehmen mit einem durchgängigen und methodisch unterstützten evolutionären Architekturentwurf besitzt entscheidende Wettbewerbsvorteile gegenüber seinen Konkurrenten.

Aber nicht nur Unternehmen mit hartem Konkurrenzdruck könnten mit Hilfe einer evolutionären Entwicklungsmethodik ihre Produktivität steigern. Wenn man sich vor Augen führt, dass ca. 80% der Kosten eines Softwaresystems in die Wartungs- und Erweiterungsphase fallen [IW99], dann erscheint es fast paradox, dass sich nahezu alle Software-Engineering Methoden hauptsächlich mit den verbleibenden 20% beschäftigen – also mit der initialen Entwicklung von Software. Bei einem Vorgehensmodell der evolutionären Softwareentwicklung existiert eine Wartungs- und Erweiterungsphase nicht. Es gibt keinen Unterschied zwischen der anfänglichen Softwareentwicklung und der Wartung und Erweiterung existierender Software. Im Zentrum steht immer die evolutionäre Weiterentwicklung und Verbesserung existierender Modelle und Implementierungen. Ein evolutionäres Vorgehensmodell bietet somit ein gewaltiges Potential für Produktivitäts- und Qualitätssteigerungen in der industriellen Softwareentwicklung.

1.2 Ziele und Ergebnisse

Kernziel dieser Arbeit ist es, wesentliche Bausteine einer durchgehenden Entwurfs- und Designmethodik zu konzipieren und anhand von Fallbeispielen und Prototypen die Tragfähigkeit dieser Konzepte zu zeigen. Die entwickelte Methodik umfasst insbesondere die Bereiche Vorgehensmodell, Beschreibungstechnik und Werkzeugunterstützung. Damit soll die Effizienz und Qualität von Architekturentwürfen nachhaltig verbessert werden.

Ein Vorgehensmodell für den evolutionären Architekturentwurf erlaubt es dem Softwarearchitekten, Architekturmodelle zu entwerfen und iterativ zu verbessern. Ein Satz pragmatischer und formal fundierter Beschreibungstechniken ermöglicht eine konsistente und adäquate Spezifikation des Architekturmodells. Wesentliche Merkmale dieser Spezifikationstechniken sind: Vollständige und unabhängige Spezifikationen für die Schnittstellen von Komponenten basierend auf Annahmen und Zusicherungen, Kompositions- und Instanzierungsspezifikationen für komponentenbasierte Systembeschreibungen, und schließlich explizite Beschreibungen der Abhängigkeiten zwischen den einzelnen Komponenten.

Mit Hilfe eines durchgängigen Werkzeugkonzepts kann der Architekt aus diesen Spezifikationen Prototypen erzeugen. Die Prototypen werden mit einer integrierten Testumgebung überprüft. Aufbauend auf den Testergebnissen kann der Designer dann die Architektur entsprechend weiter entwickeln. Mögliche Inkonsistenzen in dem Architekturmodell können durch eine frühzeitige, modellbasierte Integration erkannt werden. Somit ist es dem Architekten möglich, Sackgassen im Architekturentwurf zu erkennen und rechtzeitig Alternativen zu verfolgen.

Die wesentlichen Ergebnisse dieser Arbeit sind:

- Ein Vorgehensmodell für den evolutionären Architekturentwurf bildet den methodischen Rahmen der Arbeit und stellt so die Integrationsplattform für die restlichen Ergebnisse zur Verfügung.

- Ein wohldefinierter Satz von Beschreibungstechniken erlaubt eine pragmatische und hinreichend präzise Spezifikation von Architekturen auf Basis eines formalen Systemmodells.
- Die weitreichende Konzeption der Werkzeugunterstützung, von der evolutionären Spezifikation über die Programmgenerierung bis zur integrierten Testumgebung, ermöglicht die pragmatische Anwendung der Methodik in industriellen Projekten.
- Ein durchgängiges Fallbeispiel illustriert die praktische Anwendbarkeit und den Nutzen der Arbeit. Außerdem stellt es einen ersten Schritt in Richtung Umsetzung der Ergebnisse in der industriellen Praxis dar.

Diese Ergebnisse stellen grundlegende Konzepte und Ansätze bereit, die notwendig sind, um in Unternehmen nachhaltig eine Methodik des evolutionären Architekturentwurfs zu etablieren. Da ein wesentliches Ergebnis des Architekturentwurfs eine Reihe von identifizierten und spezifizierten Softwarekomponenten sind, ist dies ein weiterer Schritt auf dem Weg zu einer industriellen, kosteneffektiven Fertigung von Software basierend auf Komponenten.

Insgesamt sind die Ergebnisse der Arbeit für unterschiedliche Zielgruppen interessant:

- Methodenentwickler können ihre Methoden um die Konzepte eines evolutionären Architekturentwurfs erweitern. Außerdem kann die Arbeit in weiten Teilen als Fallstudie dienen, in der gezeigt wird, wie ein formale Fundierung von Beschreibungstechniken im Bereich Componentware und Softwarearchitekturen möglich ist.
- Architekten und Designer erhalten eine durchgängige Methode mit pragmatischen Spezifikationstechniken und einer weitreichenden Werkzeugunterstützung. Das Fallbeispiel illustriert die Anwendbarkeit der Methodik und bietet eine Fülle von Richtlinien und Hinweisen für den Einsatz der vorgestellten Konzepte.
- Die entwickelten Prototypen und Konzepte bieten eine Reihe von Anregungen für Werkzeughersteller. Insbesondere wird gezeigt, wie eine durchgängige Werkzeugunterstützung, die von der Modellierung über die Programmgenerierung bis zur Qualitätssicherung reicht, auf Grundlage einer Entwurfsmethodik mit formaler Fundierung erreicht werden kann.

1.3 Inhalt und Aufbau

Im folgenden wird kurz der Inhalt und der logischen Aufbau der Arbeit skizziert. Der Kern der Arbeit gliedert sich dabei in die unten beschriebenen Hauptteile:

Evolutionäre methodische Softwareentwicklung

Dieses Kapitel führt in die Thematik des Software-Engineering ein. Die fünf grundlegenden Bestandteile werden kurz erläutert: „Vorgehensmodell“, „Beschreibungstechniken und Systemmodell“, „Technologie und Architektur“, „Managementpraktiken“ sowie „Werkzeugunterstützung“. Dabei ist das Vorgehensmodell das methodische Fundament und die integrierende Plattform für alle anderen Bestandteile. Wir stellen ein entsprechendes Rahmenwerk vor, das die grundlegenden Konzepten einer methodische Softwareentwicklung vereint. Darauf aufbauend präsentieren wir ein Vorgehensmodell für den evolutionären Architekturentwurf. Dieses Vorgehensmodell bildet sowohl den Ausgangspunkt als auch die Integrationsplattform für die restliche Arbeit.

Evolutionärer Entwurf komponentenbasierter Systeme

In diesem Kapitel werden die grundlegendenden Konzepte einer formalen Methodik des evolutionären Architekturentwurfs erarbeitet. Der Ansatz einer systemmodellbasierten formalen Semantik, welche die Zusammenhänge zwischen Entwicklungsdokumenten und den entwickelten Systemen formalisiert, führt uns zu dem „Abstraction Refinement Model“. Da dieses Modell aber noch einige Schwächen im Bezug auf den evolutionären Entwurf aufweist, erarbeiten wir den neuen, verbesserten Ansatz einer prädikatenbasierten formalen Semantik. Dieser Ansatz ermöglicht es uns, auch Aussagen über inkonsistente Spezifikationen zu treffen, eine wesentliche Fragestellung während des evolutionären Architekturentwurfs. Abschließend zeigen wir noch die Zusammenhänge zwischen der prädikatenbasierten und der systemmodellbasierten formalen Semantik. So wird eine nahtlose Synergie dieser Ansätze erzielt.

Der Pausenplaner – Ein einfaches Anwendungsbeispiel

Die kommenden Kapitel sind teilweise verstärkt von theoretischer und abstrakter Natur. Damit wir nie den Bezug zur Praxis verlieren, führen wir in diesem Kapitel ein Anwendungsbeispiel ein. Dieses Anwendungsbeispiel, der Pausenplaner, ist die Entwicklung eines verteilten Editors für die Bearbeitung und Pflege von Pausenplänen an Schulen. Ausgehend von den initialen Kundenanforderungen präsentieren wir eine gründlichere Analyse der Anwendungsfälle des Pausenplaners. Die abschließend erarbeitete erste Aufteilung in fachliche Anwendungskomponenten dient als Ausgangspunkt und Illustrationsbeispiel für die Konzepte und Ansätze, die in den folgenden Kapiteln vorgestellt werden.

Grundlagen komponentenbasierter Systeme

In diesem Kapitel wird ein Systemmodell für komponentenbasierte Systeme definiert. Dieses formale Modell liefert ein präzises Verständnis der grundlegenden Begriffe von Componentware. Dabei werden zentrale Konzepte wie Komponente und Schnittstelle präzisiert. Insbesondere steht eine wohldefinierte Verhaltensbeschreibung von Komponenten und ihren Schnittstellen im Vordergrund. Die Verhaltensbeschreibung umfasst die Kommunikation zwischen Komponenten, die Zustandsänderungen von Komponenten und die Änderungen der Verbindungsstruktur in komponentenbasierten Systemen. Zusammen mit einem entsprechenden Kompositionsbegriff charakterisieren wir damit die Klasse aller Systeme, die in dieser Arbeit behandelt werden. Damit steht uns das formale Fundament für die restliche Arbeit zur Verfügung.

Architekturspezifikation komponentenbasierter Systeme

Im Brennpunkt dieses Kapitels stehen Beschreibungstechniken für eine vollständige Spezifikation von Softwarearchitekturen komponentenbasierter Systeme. Zentral dabei sind die Verhaltensbeschreibungen von Schnittstellen und deren Komponenten. Damit man aus diesen unabhängigen Komponentenspezifikationen eine vollständige Systembeschreibung erstellen kann, werden dem Entwickler Kompositions- und Instanzierungsspezifikationen zur Verfügung gestellt. Wir führen sowohl textbasierte als auch UML-basierte, grafische Varianten der Beschreibungstechniken ein. Auf Basis des Systemmodells werden diese Beschreibungstechniken mit Hilfe der prädikatenbasierten formalen Semantik formal fundiert. So ist es möglich, eine konstruktive Abbildung von der Spezifikation in eine Programmiersprache festzulegen. Dies ermöglicht ein frühes Prototyping und eine szenarienbasierte Validierung der modellierten Softwarearchitektur.

Erweiterte Spezifikationen für den evolutionären Entwurf

Mit den Beschreibungstechniken des vorhergehenden Kapitels lassen sich Softwarearchitekturen vollständig und präzise spezifizieren. Ändern sich einzelne Komponentenspezifikationen im Zuge des evolutionäre Entwurfs, so können noch keine spezifischeren Aussagen über die Auswirkungen auf das gesamte Architekturmodell getroffen werden. Aus diesem Grund erarbeiten wir in diesem Kapitel entsprechende Erweiterungen der Beschreibungstechniken aus dem vorhergehenden Kapitel. Mit Hilfe sogenannter Annahme-/Zusicherungsverträge werden die Abhängigkeiten zwischen den einzelnen Komponenten explizit spezifiziert. Evolutionäre Änderungen an einzelnen Komponenten und die damit verbundenen Auswirkungen am Architekturmodell können so dem Architekten frühzeitig dargestellt werden.

Werkzeugunterstützung

Eine Durchgängige und umfassende Werkzeugunterstützung ist ein essentieller Bestandteil einer erfolgreichen Softwareentwicklung. Dieses Kapitel liefert die grundlegenden Konzepte und Bausteine für eine adäquate Werkzeugunterstützung der erarbeiteten Ergebnisse: Angefangen bei Modellierung, Prototyping und Programmgenerierung, über Unterstützung der Konsistenz- und Integrationsüberprüfung nach einer evolutionären Weiterentwicklung, bis zu Test- und Migrationswerkzeugen. Design und Realisierung ausgewählter Werkzeuge auf Basis moderner Technologien werden diskutiert und prototypische Implementierungen vorgestellt.

1.4 Verwandte Arbeiten

Wie bereits erwähnt stehen im Zentrum der Arbeit drei aktuelle Themen, die in einer integrierten Methodik vereint werden: Componentware, Softwarearchitektur und ein evolutionäres Vorgehensmodell. Dementsprechend gibt es eine Vielzahl an Arbeiten, die sich in dem einen oder anderen Bereich besonders hervorheben. Eine Arbeit, die bereits diese drei Aspekte in einer Methodik vereinigt, existiert jedoch noch nicht. Im folgenden werden wir auf die Arbeiten eingehen, die bestimmte Aspekte der Dissertation besonders beeinflusst haben und somit einen natürlichen Bezug zu dieser Arbeit haben.

Die Grundprinzipien, die dieser Arbeit zu Grunde liegen, stammen aus dem Gebiet des Software-Engineering. Insbesondere die Arbeiten von Ernst Denert [Dene91], Gustav Pomberger und Günther Blaschek [PB96], Petra Kroha [Kroh97], Bernd Oestereich et al. [OHJ+99], Helmut Balzert [Balz00], Peter Brössler und Johannes Siedersleben [BS00], und Ian Sommerville [Somm00] sind hierbei zu nennen.

Viel Einfluss auf den methodischen Teil dieser Arbeit hatten nahezu alle modernen objektorientierten Softwareentwicklungsmethoden. Inzwischen gibt es hierzu eine Vielzahl guter Arbeiten. Im Kontext der vorliegenden Dissertation ist dabei insbesondere die Arbeit von Bertrand Meyer [Meye97] hervorzuheben.

Das Vorgehensmodell des evolutionären Architekturentwurfs wurde maßgeblich durch die iterativen und inkrementellen Vorgehensmodelle [DW98, DW99, JBR99, OHJ+99, OOSE01, Kruc00] sowie durch die Arbeiten aus der Process Patterns Community [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b] beeinflusst.

Darüber hinaus sind eine Reihe von Arbeiten im Umfeld von Design und Architektur Patterns [GHJV95, BMR+96, Fowl97, SSRB00] sowie neuere Werke auf dem Gebiet von Componentware [Szyp97, Grif98, HS99, ZL99 Alle00, Brow00, CD00, LS00] und Softwarearchitek-

turen [BCKB98, HNS99, BHH00, Bosh00, DKW00, JRLL00] entstanden. Diese Arbeiten haben die hier vorgestellten Konzepte von Componentware und Softwarearchitektur mit geprägt.

Ein zentrales Element des evolutionären Architekturentwurfs sind Beschreibungstechniken für komponentenbasierte Softwarearchitekturen [Raus01b]. Als Ausgangspunkte der entwickelten Notation diente hier natürlich die UML [BJR98, RJB98, OMG00a]. Für die Architekturspezifikation im speziellen waren aber insbesondere die Annahme-/Zusicherungskonzepte in Eiffel von Bertrand Meyer [Meye92, Meye97] und in der Java Modeling Language von Gary T. Leavens, Albert L. Baker und Clyde Ruby [LBR99] ausschlaggebend. Einige Ideen und Konzepte aus dem Bereich der Architectural Description Languages (ADL) [SG96] und der Arbeiten auf dem Gebiet von Spezifikationen basierend auf der Object Constraint Language (OCL) [WK98] flossen ebenfalls in diese Arbeit ein.

Für eine optimale Unterstützung des evolutionären Entwurfs sind insbesondere die entwickelten Annahme-/Zusicherungsverträge essentiell [Raus99, Raus00a, Raus00b, Raus00c]. Die Wurzeln dieser Ideen liegen in den Konzepten der Reuse Contracts [MSL98, MLS98] und Interaction Contracts [HHG90, Holl93].

Die Idee, diese Beschreibungstechniken auf Basis eines formalen Systemmodells zu fundieren, stammen aus dem Projekt Syslab [RKB95] und einigen Dissertation am Lehrstuhl von Professor Dr. Manfred Broy. Insbesondere die Dissertationen von Dr. Bernhard Rumpe [Rump96], Dr. Klaus Bergner [Berg97] und Dr. Ingolf Krüger [Krüg00] haben mich dabei begleitet.

Der Ursprung des vorliegenden formalen Systemmodells für Componentware stammt aus den Arbeiten des Forschungsprojektes FORSOFT A1 [BRS+99, BBR+00]. Dieses wiederum basiert auf Arbeiten im Bereich von FOCUS [BDD+92, BS01]. FOCUS ist eine mathematische Modellierungstechnik, die am Lehrstuhl von Professor Dr. Manfred Broy entwickelt wurde. Darüber hinaus sind einige Konzepte aus Communicating Sequential Processes [Hoar85], aus den Architectural Description Languages (ADL) [SG96] und aus dem Bereich der theoretischen Fundierung objektorientierter Sprachen [AC96] in die Arbeit mit eingeflossen.

Die Basis der Ergebnisse im Bereich der Werkzeugunterstützung und der Beispielarchitekturen stammt ebenfalls direkt aus den Arbeiten am Lehrstuhl von Professor Dr. Manfred Broy und von dem Forschungsprojekt FORSOFT A1 [BRS98a, BRSV99b, BRSV99c, BRSV00]. Hier sind insbesondere die Arbeiten an den Werkzeugen AutoFocus [BRS98b, BRS99], AutoMate [BKR98] und insbesondere DesignIt [Kivl00, Lave00, HP01] zu nennen. Darüber sind noch zusätzliche Aktivitäten bei den Werkzeugen und Frameworks SmartEvent [BRSV99a, BBR+99], ShapeShifter [FRS00] und Goal [BFR00] aufzuführen.

2 Evolutionäre methodische Softwareentwicklung

Softwareingenieure stehen vor der Aufgabe, nutzbringende Systeme zur maschinellen Verarbeitung von Informationen zu konzipieren, zu realisieren und bereit zu stellen. Die Entwicklung dieser Softwaresysteme ist eine echte Herausforderung, insbesondere auf Grund der geforderten Größe, Komplexität, Langlebigkeit und Flexibilität der Systeme. Für die Bewältigung der anstehenden Aufgaben benötigt der Softwareingenieur das entsprechende Rüstzeug.

Das Software-Engineering, ein Teilgebiet der Informatik, versucht, einen Baukasten von Methoden, Konzepten, Verfahren und Technologien für eine methodische Softwareentwicklung bereit zu stellen. Im einzelnen umfasst die methodische Softwareentwicklung folgende Teilgebiete, die in Kapitel 2.1 detaillierter diskutieren werden: Vorgehensmodell, Beschreibungstechniken und Systemmodell, Technologie und Architektur, Managementpraktiken und Werkzeugunterstützung.

Das Vorgehensmodell ist der zentrale Leitfaden einer methodischen Softwareentwicklung. Im Laufe der Entwicklung werden dabei verschiedene Modelle eines Systems erstellt, zum Beispiel ein Anwendungsfallmodell, ein Designmodell oder der Programmcode selbst. Der Detaillierungsgrad, die verwendeten Sichten und Beschreibungsmittel sowie die prozessspezifischen Abhängigkeiten zwischen den Modellen werden durch das Vorgehensmodell festgelegt.

Die Konzeption und Entwicklung einer vollständigen und umfassenden Methode für die Softwareentwicklung würde jedoch den Rahmen dieser Arbeit sprengen. Andererseits ist es auch nicht akzeptabel, eine isolierte Lösung für ein spezifisches Problem zu erarbeiten, ohne entsprechende Integrations- und Weiterentwicklungsmöglichkeiten vorzusehen.

Deshalb legen wir dieser Arbeit ein methodisches Rahmenwerk zu Grunde, das wir in Kapitel 2.2 vorstellen. Dieses Methodik-Framework stellt uns die grundlegenden Konzepte und Begriffe zur Verfügung, um in den kommenden Kapiteln den Teilausschnitt einer methodischen Softwareentwicklung zu beschreiben, den wir in dieser Arbeit untersuchen wollen – den evolutionären Architekturentwurf. Dabei bleiben wir offen für Erweiterungen und die Integration zusätzlicher methodischer Aspekte.

Wesentliche Elemente des methodischen Rahmenwerks sind ein integriertes Produktmodell und das Konzept der Prozessmuster. Zentrale Bestandteile des Produktmodells sind die verschiedenen Modelle eines Softwaresystems, die im Laufe der Softwareentwicklung entstehen. Im Kapitel 2.3 diskutieren wir die charakteristischen Beziehungen zwischen diesen Modellen. Unser besonderes Augenmerk gilt dabei dem Architekturmodell. Wir stellen die Sichten, die Struktur und die Beschreibungsmittel vor, die in einem Architekturmodell verwendet werden.

Schließlich präsentieren wir in Kapitel 2.4 das evolutionäre Vorgehensmodell des Architekturentwurfs in Form eines konkreten Prozessmusters. Dabei zeigen wir insbesondere die Zusammenhänge zwischen den spezifischen Aktivitäten des evolutionären Entwurfs und den einzelnen Bestandteilen des Architekturmodells auf. Dieses Vorgehensmodell und der zugehörige Begriff des Softwarearchitekturmodells stellen die zentrale Integrationsplattform für die restliche Arbeit dar. Somit können alle weiteren Ergebnisse dieser Arbeit zu einer integrierten Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme vereint werden.

2.1 Grundbausteine der methodischen Softwareentwicklung

Softwaresysteme gehören zu den komplexesten Gebilden, die von Menschen geschaffen werden. Bei der Entwicklung von Software stehen Softwareingenieure vor der Aufgabe bestimmte Anforderungen in einer vorgegebenen Zeit, im Rahmen eines festen Budgets und mit der geforderten Qualität umzusetzen. Die Beherrschung von Kosten, Zeit und Qualität ist eine der größten Herausforderungen des Software-Engineering.

Um dieses magische Dreieck besser meistern zu können, sind in den letzten Jahrzehnten eine Reihe von Methoden für das Software-Engineering entstanden: beispielsweise SA/SD [DeMa79, YC79], VDM [Jone86], SSADM [AG90], OMT [RBP+91], Objectory [Jaco92], HP's Fusion Method [Cole93], Booch Method [Booc94], SDL [OFM+94], IBM's OOTC [IBM97], Catalysis [DW98], V-Modell '97 [DW99], Object Engineering Process [OOSE01], oder der Unified Software Development Process [JBR99, Kruc00], um nur einige zu nennen[1].

Diese Methoden sind mehr oder weniger ausgereift und können in Form einer meist umfangreichen Dokumentation erstanden und angeeignet werden. Die Antwort auf die Frage, in wie weit diese Methoden uns aber bei den Herausforderungen des Software-Engineering tatsächlich helfen, bleibt dem Leser selbst überlassen. Oder, um es mit Tom DeMarco's Worten zu sagen:

> „There is a big difference between Methodology and methodology"
>
> (Zitat aus [DL99], Seite 115)

Die groß geschriebene „Methodology" ist der Versuch, ein generelles Verfahren zur Durchführung von komplexen Aufgaben, wie die eines Softwareingenieurs, zu etablieren. Alle wichtigen Entscheidungen sind bereits durch die Methode vorgegeben, wie zum Beispiel Vorgehen, Dokumentation, Technologie und Werkzeuge. Unter der klein geschriebenen „methodology" wird hingegen ein genereller Ansatz für eine geordnete Projektabwicklung verstanden. Dieser ist nicht in einem dicken Buch zu finden, sondern in den Köpfen der Menschen, die das Projekt durchführen.

Sowohl kleine wie auch große Methoden bestehen aus Konzepten und Techniken, die sich den folgenden Gruppen zuordnen lassen:

- Vorgehensmodell
- Beschreibungstechniken und Systemmodell
- Technologie und Architektur
- Managementpraktiken
- Werkzeugunterstützung

Abbildung 2.1 stellt diese Gruppen von Elementarmethoden nochmals grafisch im Zusammenhang dar. In den folgenden Abschnitten besprechen wir diese Grundbausteine einer methodischen Softwareentwicklung eingehender. Im Vordergrund steht dabei aber nicht eine umfassende und vollständige Diskussion. Vielmehr ist es unser Ziel, ein grundlegendes, gemeinsames Verständnis für diese Begriffe zu bekommen. Soweit es notwendig und hilfreich für die Arbeit ist, werden wir bestimmte Punkte detaillierter betrachten. Auf weiterführende Literatur wird an den geeigneten Stellen verwiesen. Allgemeine Literaturquellen zum The-

[1] Weitere Verweise auf die Methoden im Software-Engineering sind auf der WWW-Seite [Bott01] zu finden.

mengebiet des Software-Engineering, die wir für den folgenden Überblick verwendet haben, sind: [Dene91, PB96, Kroh97, OHJ+99, Balz00, BS00, Somm00]

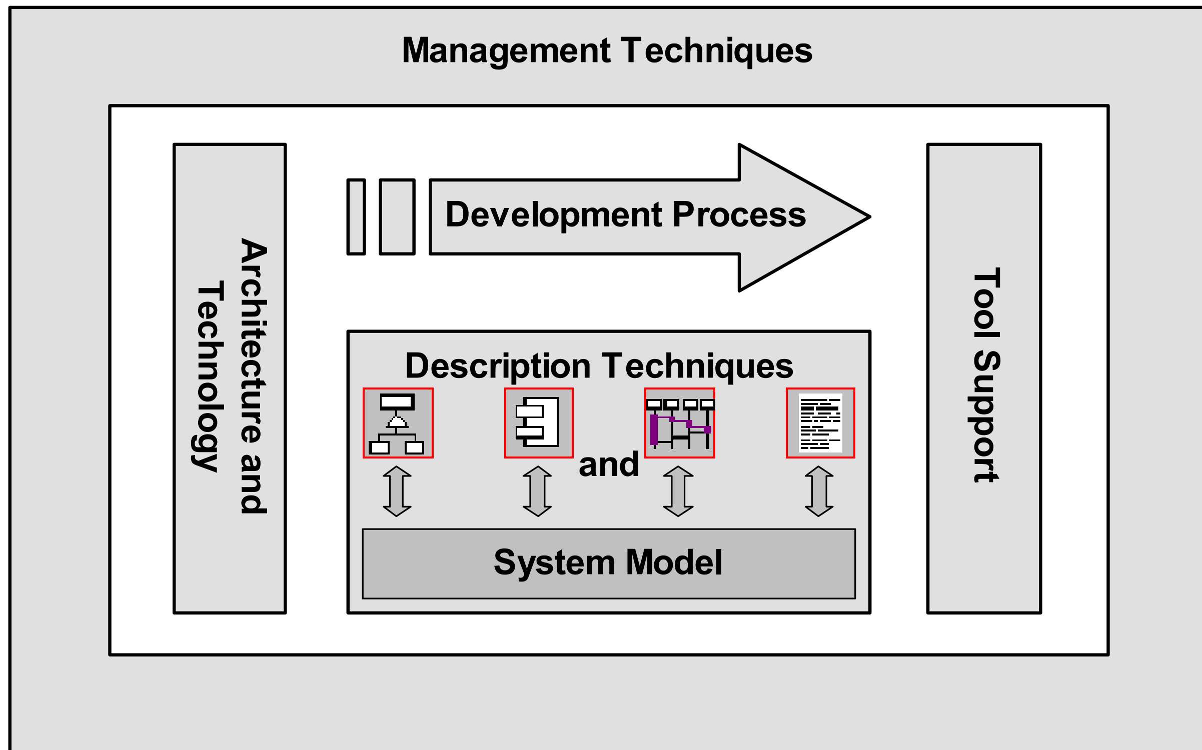

Abbildung 2.1: Grundbausteine der methodischen Softwareentwicklung

2.1.1 Vorgehensmodell

Das bekannteste und wohl auch älteste Prozessmodell ist das Wasserfallmodell [Royc70]. In den 70'er Jahren entstanden eine Reihe unterschiedlicher Phasenmodelle und kombinierte Phase-/Tätigkeitsmodelle. Die Planungseuphorie in dieser Zeit basierte auf der Vorstellung, dass man ein Problem nur detailliert genug planen müsse, um es bewältigen zu können. Die dabei vorherrschende Ausrichtung nach sequenziell abzuarbeitenden Phasen bewirkt, dass Planungs- und Entwicklungsfehler erst spät bemerkt und Risiken eventuell zu spät erkannt werden. Veränderungen in der Planung sind kostspielig und schwer zu bewerkstelligen. Insbesondere erfolgt die Rückkopplung aus der Implementierungsphase erst sehr spät (vgl. [Müll99]).

Ende der 80'er setzte sich dann vermehrt die Erkenntnis durch, dass es einen ganz natürlichen, nicht vollständig verhinderbaren Wandel bei der Planung und den Anforderungen gibt. Aus diesem Bewusstsein entstanden und entstehen immer noch eine Fülle neuer Prozessmodelle. Einen Meilenstein in dieser Entwicklung setzte das bekannte Spiralmodell von Barry Boehm [Boeh86]. Das Spiralmodell ist zwar immer noch rein sequentiell, jedoch beinhaltet es bereits eine iterative Planung mit integrierter Risikoanalyse.

Neuere Prozessmodelle, wie zum Beispiel die unterschiedlichen Formen von Prototyping [PB96, Kroh97], iterative und inkrementelle Modelle [DW98, DW99, HK99, JBR99, Müll99, OHJ+99, Kruc00], Process Pattern Ansätze [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b] oder eXtreme Programming [Beck99] als eine der neuesten Entwicklungen treiben diese Ideen weiter voran. Im Kern sind diese aber ähnlich: In mehreren Iterationen wird das System schrittweise entwickelt. Prototypen, technische Durchstiche, kur-

ze Integrationszyklen und anschließendes Testen sorgen dabei für eine frühzeitiges Rückkopplung. Problem und Risiken können so rechtzeitig erkannt, angegangen und aus dem Weg geräumt werden.

Unabhängig von einem bestimmten Vorgehensmodell finden sich in allen gewisse Konzepte wieder, wie zum Beispiel das Konzept von Aktivitäten und Rollen. Nach dem IEEE Software Engineering Standard ist die Definition des Entwicklungsprozess wie folgt:

> „A sequence of steps performed for a given purpose: for example, the software development process."
>
> (Zitat aus [IEEE99])

Ein Vorgehensmodell in unserem Sinne ist jedoch noch etwas enger gefasst: Es definiert eine eventuell zeitlich geordnete Menge von Aktivitäten (*Activity*), die anhand von methodischen Richtlinien und Vorgaben (*Process Pattern*) ausgeführt werden. Aktivitäten benötigen eine Menge von Produkten (*Work Product*) als Eingabe und manipulieren bzw. erzeugen eine Menge von weiteren Produkten (*Work Product*). Diese Aktivitäten werden von bestimmten Rollen (*Role*) ausgeführt. Die Rollen wiederum werden entsprechenden Personen (*Person*) zugeordnet.

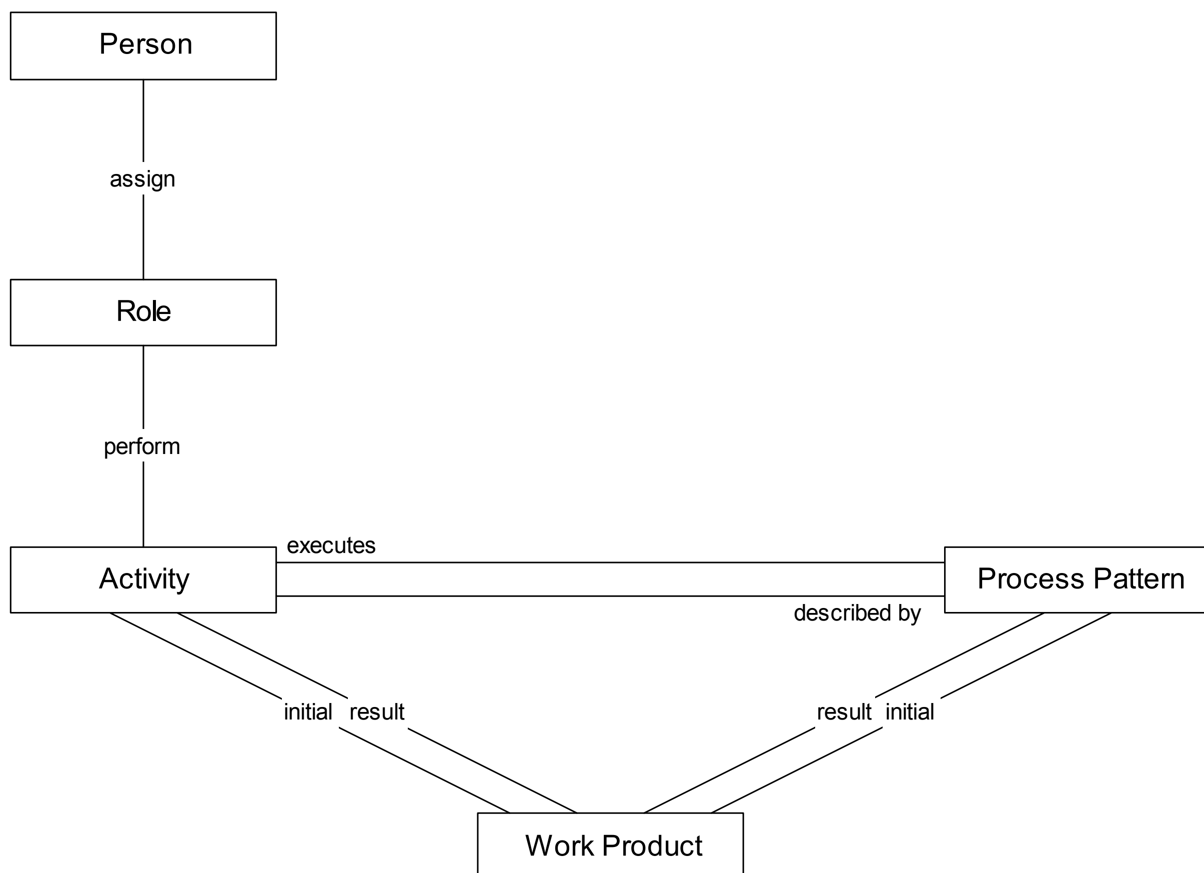

Abbildung 2.2: Grundlegende Konzepte in Vorgehensmodellen

Abbildung 2.2 illustriert dieses Modell eines Entwicklungsprozesses als UML Klassendiagramm. Besonderes Augenmerk gilt hierbei der Unterscheidung zwischen Prozessmustern und Aktivitäten. Prozessmuster beschreiben die methodischen Richtlinien und Vorgaben für die korrekte Ausführung (*executes*) einer Reihe von Aktivitäten. Die Durchführung einer Aktivität kann dabei wiederum durch weitere, eventuell alternative Prozessmuster beschrieben

werden (*described by*). Diese hierarchische Strukturierung von Prozessmustern und Aktivitäten führt zu einem modularen Vorgehensmodell, das es uns erlaubt, in den einzelnen Projekten spezifisch angepasste Folgen von Prozessmustern und somit auch Aktivitäten durchzuführen (siehe auch [GMP+01a, GMP+01b]).

Diese Modellierung und die zugehörige Begriffswelt spiegelt den heutigen Stand der Kunst wieder. Ähnliche Definitionen können den verschiedenen Prozessmodellen entnommen werden. Stellvertretend hierfür sei auf das V-Modell '97 verwiesen, das eine fast identische Sichtweise hat [DW99].

Manager und Projektverantwortliche verstehen aber unter einem Vorgehensmodell meist Prozessleitfäden, die auf die Belange des jeweiligen Unternehmens zugeschnitten sind und sich kochbuchartig zur Durchführung von Projekten einsetzen lassen. Derartige Kochbücher existieren aber nicht. Treffend und ehrlich wird dies in dem Buch „Succeding with Objects" von Adele Goldberg und Kenneth Rubin geschildert [GR95]: Hier beschreiben die Autoren, wie sie sich bei der Durchführung objektorientierter Projekte mit dem Gedanken getragen haben, eine allgemein brauchbare, kochbuchartige Anleitung zu schreiben. Nach langjähriger Erfahrung mit objektorientierter Projektarbeit sind sie aber zur Überzeugung gekommen sind, dass dies nicht möglich ist.

Deshalb sind alle der neueren Prozessmodelle verstärkt von generischer Bauart. Sie sind als Schablonen oder Universalbaukästen gedacht, zur Erstellung eines spezifisch angepassten Vorgehensmodells für ein bestimmtes Unternehmen, für verschiedene Projektarten oder für ein konkretes Projekt. Diese Anpassung ist im Regelfall mit größerem Aufwand verbunden. Eine Anpassung je Projekt lohnt sich deshalb nicht immer (vgl. hierzu [OHJ+99, GNR+00]).

Trotzdem sollten das Vorgehensmodell, die zu erzeugenden Produkte, die notwendigen Aktivitäten und die methodischen Richtlinien von dem Projektteam speziell für ihr Projekt ausgewählt und eventuell sogar noch angepasst werden. Nur wenn es vom ganzen Team gelebt wird, kann das Vorgehensmodell zu einer elementaren Stütze des Projektes werden: Denn das wichtigste ist und bleibt der Faktor „Peopleware" [DL99]: Menschen machen Projekte!

2.1.2 Beschreibungstechniken und Systemmodell

Modelle sind vereinfachte Abbildungen der Realität. Softwaresysteme sind sehr komplex. Meist können wir sie in ihrer Vollständigkeit nicht mehr verstehen. Deshalb ist die Modellbildung ein gängiges Mittel, um mit der Komplexität von Softwaresystemen umzugehen. Typischerweise beinhaltet ein Modell eines Softwaresystems eine umfassende Beschreibung des Systems aus bestimmten Perspektiven.

Grundlegende Idee der Modellbildung ist es, Aussagen im Modell so darstellen zu können, dass sie von allen Beteiligten gleich interpretiert werden. Das Modell selbst und die Aussagen über das Modell werden mit Hilfe von Beschreibungstechniken festgehalten. Speziell in der Objektorientierung, verstärkt aber auch auf dem Gebiet von Componentware und Softwarearchitekturen, hat sich hierfür die Unified Modeling Language (UML) als Standardmodellierungstechnik durchgesetzt [BJR98, RJB98, OMG00a].

Analog zu den meisten Beschreibungssprachen ist die UML eine Modellierungssprache, die sowohl aus textbasierten als auch aus grafischen Elementen besteht. Die Notation wird durch eine Menge festgelegter Symbole definiert, kann aber über entsprechende Mechanismen erweitert werden [DSB99, KRSW01]. Über den Grad der semantischen Fundierung der UML

lässt sich durchaus streiten. Hier gibt es Aussagen von „formal fundiert“ [OMG00a] über „semiformal“ [EK99, WK98] bis zu „nur Notation“ [BGH+98, EFLR99, HR00].

Unbestritten ist jedoch, dass Beschreibungstechniken unabhängig von einem spezifischen Vorgehensmodell betrachtet werden können. Beschreibungstechniken liefern beispielsweise keine Antwort auf die Fragestellung, ob und wie ein Sequenzdiagramm in der Analyse eingesetzt werden soll. Dies wird durch ein zu Grunde liegendes Vorgehensmodell beantwortet (vgl. Kapitel 2.1.1). Das Vorgehensmodell ordnet bestimmten Aktivitäten, zum Beispiel der „Projektplanung“, bestimmte Produkte, wie zum Beispiel den „Projektplan“ zu, wie in Abbildung 2.2 bereits illustriert wurde.

Während des gesamten Lebenszyklus eines Softwaresystems entstehen eine Vielzahl unterschiedlicher Produkte (*Work Product*). Diese Produkte werden unter der Verwendung von spezifischen Modellierungskonzepten (*Modeling Concept*) mit einer entsprechenden Notation (*Notation*) beschrieben (vgl. Abbildung 2.3).

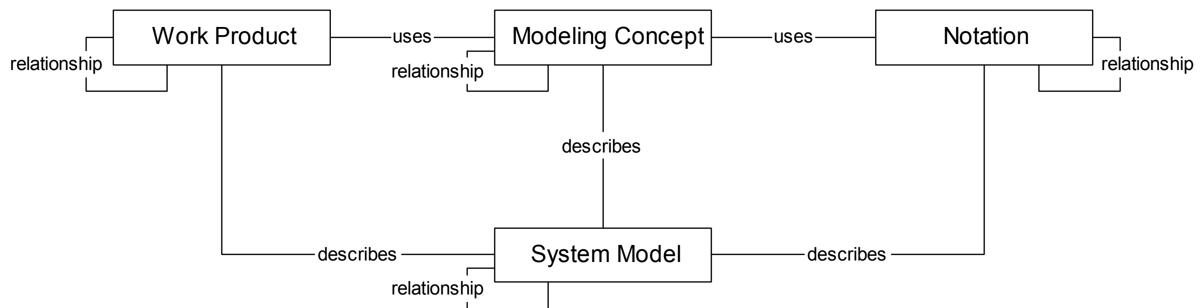

Abbildung 2.3: Zusammenhang zwischen Prozessmodellen und Beschreibungstechniken

So ist beispielsweise der „Projektplan“ ein Produkt, das im Laufe der Softwareentwicklung entsteht und verändert wird. Für die Beschreibung dieses Produktes wird meist das Modellierungskonzept der „Zeit/Aufgaben-Flussmodellierung“ verwendet, mit der Notation „Gantt-Diagramm“ oder „PERT-Diagramm“. Ein anderes Beispiel ist das Produkt „Datenmodell“. Hierfür wird unter anderem das Modellierungskonzept „Datenstrukturmodellierung“ verwendet, mit der Notation „Klassendiagramm“ oder „Entity/Relationship Diagramm“.

Je nach Art und Zielsetzung eines konkreten Softwareentwicklungsprojektes entstehen während der Entwicklung eine Vielzahl von Produkten. Diese stehen meist auf komplexe Weise miteinander in Verbindung. Die präzise Formulierung der Kontextbedingungen zwischen Produkten erhalten wir über das Konzept des Systemmodells (*System Model*). Ein Systemmodell charakterisiert dabei die Menge aller Systeme und stellt somit eine semantische Basis zur Verfügung. Die Kontextbedingungen zwischen Produkten, Modellierungskonzepten und Notationen werden so in einem gemeinsamen Systemmodell charakterisiert (*describes)*.

Im allgemeinen Fall wird es nicht immer möglich sein, alle Kontextbedingungen in einem Systemmodell zu beschreiben, da es teilweise sehr große Unterschiede zwischen den verschiedenen Produkten gibt. Beispielsweise erscheint es fragwürdig, ob es möglich und sinnvoll ist, in absehbarer Zeit ein gemeinsames Systemmodell für „Strukturdiagramme“ und „Projektpläne“ zu erstellen. Trotzdem existieren auch zwischen diesen Elementen Querbezüge. In einem Projektplan könnte gefordert sein, dass die Strukturmodellierung zu einem gewissen Zeitpunkt im Projektverlauf einen festgelegten Stand erreicht hat.

Produkte, Modellierungskonzepte, Notationen und Systemmodelle besitzen selbst komplexe, meist rekursive Strukturen. In Abbildung 2.3 wird dies durch die Beziehung *relationship* dar-

gestellt. Eine möglich Beziehung wäre zum Beispiel die „Ist-Teil-Von-Beziehung“ zwischen *Work Product*. So könnte beispielsweise ein Produkt „Anforderungsspezifikation“ aus den Teilprodukten „Systemvision“, „Benutzer-System-Interaktionsspezifikation“, „Anwendungsfallanalyse“, „Geschäftsprozessmodellierung“ und „Geschäftsentitätenmodell“ bestehen (vgl. [GMP+01a, GMP+01b]). Analoge Beziehungen lassen sich auch für die anderen Konzepte Systemmodell, Notation und Modellierungskonzept finden. Eine umfassendere Diskussion der Produkte in der Softwareentwicklung ist auch in den Arbeiten von Wolfgang Schwerin zu finden (siehe [DSV99]).

2.1.3 Technologie und Architektur

Das Systemmodell charakterisiert die Menge aller Systeme. Systeme werden durch eine Menge von Produkten beschrieben. (vgl. Kapitel 2.1.2). Dieser konzeptionellen Sichtweise kann man entsprechende Gegenstücke aus der industriellen Praxis zuordnen, wie in Abbildung 2.4 dargestellt.

In der pragmatischen Sichtweise sprechen wir nicht von Produkten, die ein System beschreiben, sondern von Spezifikationen und Modellen, die beispielsweise mit einem CASE-Tool angefertigt werden. Mit Hilfe von Generatoren werden aus den Spezifikationen mehr oder weniger vollständige Programme erzeugt, die in einer spezifischen Laufzeitumgebung ablauffähig sind.

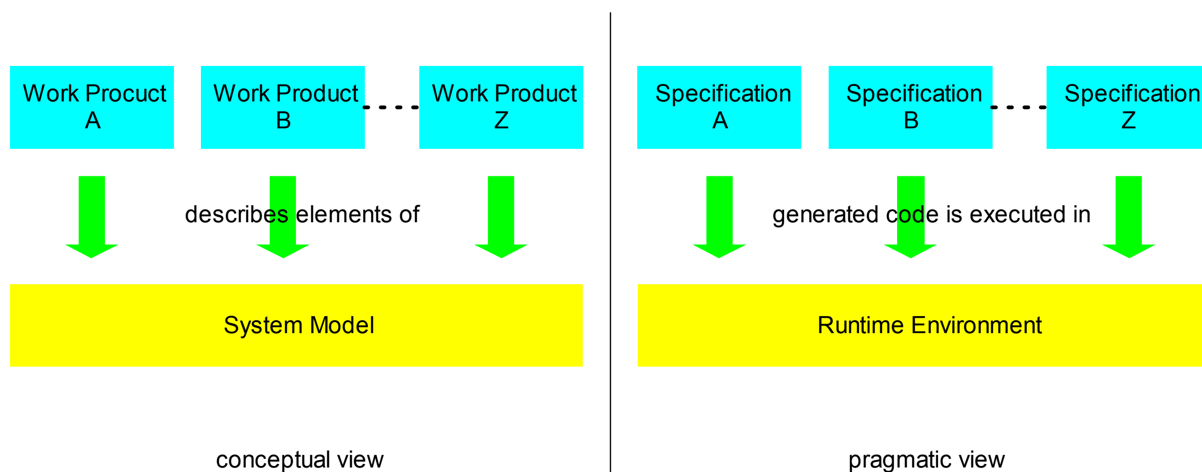

Abbildung 2.4: Softwarearchitektur und Laufzeitumgebung

Dieses Verständnis einer Laufzeitumgebung ist bereits seit längerem bekannt: In „Software-Engineering“ von Ernst Denert wird diese Laufzeitumgebung als die ideale Maschine bezeichnet [Dene91]. Mit dem Werkzeug AutoFocus [AF01] wurde am Lehrstuhl von Professor Dr. Manfred Broy eine Laufzeitumgebung für die formale Methodik FOCUS [BDD+92, BS01] entwickelt. Und schließlich, für das Systemmodell in Kapitel 5, das dieser Arbeit zu Grunde liegt, werden wir in Kapitel 8 eine entsprechende Laufzeitumgebung vorstellen.

Systemmodell und Laufzeitumgebung basieren dabei stets auf spezifischen Konzepten, wie zum Beispiel einem Komponentenmodell oder einer bestimmten Form der Kommunikation zwischen Komponenten. Technologien liefern Implementierungen dieser Konzepte und sind somit die Basis für die Realisierung von Laufzeitumgebungen. So sind in den existierenden komponentenbasierten Technologien, wie zum Beispiel in Java Enterprise Beans [SUN01], CORBA Components [OMG99] oder COM+ [Isem00], Komponentenmodelle vorhanden und verschiedene Formen der Kommunikation zwischen Komponenten realisiert.

Diese Technologien liefern uns die notwendige technische Basis um Laufzeitumgebungen für komponentenbasierte Systeme erstellen zu können. Unter einer Softwarearchitektur hingegen wollen wir eine konkrete Modellierung eines Systems auf Basis einer bestimmten Laufzeitumgebung bzw. Systemmodells verstehen:

> „A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system."

(Zitat aus [BMR+96], Seite 384)

Eine Softwarearchitektur ist – vereinfacht ausgedrückt – ein Bauplan für die strukturelle Softwareorganisation des Gesamtsystems oder exakter: eine Beschreibung der Subsysteme und Komponenten eines Softwaresystems und deren Beziehungen.

Die Zielsetzung des Softwarearchitekturentwurfs ist es, einen stabilen und gut dokumentierten Rahmen zu finden, der Flexibilität und Erweiterbarkeit eines Systems erlaubt und dabei Performanz und Zuverlässigkeit garantiert. Das Ergebnis des Entwurfs ist ein Architekturmodell des zu entwickelnden Systems. Die einzelnen Bestandteile eines Architekturmodells werden wir in Kapitel 2.3 vorstellen. Den evolutionären Prozess des Architekturentwurfs diskutieren wir in Kapitel 2.4.

Eine präzisere Definition einer Softwarearchitektur können wir am Ende von Kapitel 5 auf Basis des dort vorgestellten Systemmodells liefern. Die Beschreibungstechniken für die Modellierung derartiger Softwarearchitekturen werden in Kapitel 6 und 7 vorgestellt und anhand eines Anwendungsbeispiels praktisch angewandt und demonstriert.

2.1.4 Managementpraktiken

Das Vorgehensmodell leitet die Ingenieure bei der Erstellung des Softwaresystems, das mit entsprechenden Beschreibungstechniken modelliert wird. Eine möglichst weitgehende Abbildung von den Beschreibungstechniken in ein Systemmodell liefert ein fundiertes Verständnis der entwickelten Modelle. Ein Laufzeitsystem basierend auf modernen Technologien und einer Reihe von Architekturprinzipien garantiert eine effiziente Realisierung der Modellierung.

Managementpraktiken dagegen sorgen für ein gesichertes organisatorisches und betriebswirtschaftliches Umfeld während der gesamten Projektlaufzeit. Im wesentlichen unterscheiden wir vier verschiedene Gebiete des Managements (vgl. auch [DW99, Müll99]): Projektmanagement, Qualitätssicherung, Änderungsmanagement und Konfigurationsmanagement. Aus Gründen der Vollständigkeit gehen wir im folgenden kurz auf diese Aspekte ein. Für die restliche Arbeit sind diese Themen von geringerer Bedeutung.

Projektmanagement

Eine der wichtigsten Aufgaben des Projektmanagement ist die Planung und Überwachung des Projektverlaufes. Dazu muss man sich mindestens über die Zeitplanung, die Ressourcenplanung, den Entwicklungsprozess, die Teamstruktur und das Risikomanagement Gedanken machen. Diese Pläne können nicht unabhängig voneinander betrachtet werden, beispielsweise ist die Zeitplanung stark von der Ressourcenplanung abhängig (vgl. [OHJ+99]).

Allerdings sind Pläne nur dann von Nutzen, wenn sie die Realität widerspiegeln. Dementsprechend müssen die Pläne laufend aktualisiert werden. Nur so können sie sinnvoll zur Steuerung eines Projektes verwendet werden. Es ist deshalb wichtig laufend die Planung mit dem tatsächlichen Projektstand zu vergleichen. Die stetige Rückkopplung ist ausschlaggebend für eine effektive Projektsteuerung.

Planung, Messung und Projektsteuerung gehen dabei Hand in Hand. Aus dem Vergleich von Planung und Messung lassen sich Abweichungen erkennen. Eine Analyse der möglichen Ursachen für Abweichungen geben Aufschluss über sinnvolle Steuerungsmaßnahmen, um die Risiken zu minimieren. Durch ein evolutionäres Vorgehensmodell entstehen mehrere Messpunkte, die für eine feinere Planung verwendet werden können. Nach jedem Evolutionsschritt sollte ein Zyklus von Messung, Steuerung und Planung durchgeführt werden.

Wichtig bei der Steuerung und Führung von Projekten ist, dass die Projektziele klar kommuniziert werden. Es sollten nie zu viele Ziele sein. Die Entwicklung des Softwaresystems sollte dabei immer im Vordergrund stehen.

Das Management hat auch die Aufgabe das Team für das Projekt zusammenzustellen. Beim Teamaufbau und -abbau kommt es darauf an, die richtige Teamstruktur für die jeweilige Projektsituation zu finden. Dabei sollte die Motivation der Mitarbeiter immer ein bestimmender Faktor sein. Denn die Motivation hat den größten Einfluss auf die Produktivität. Das Management ist dafür verantwortlich eine produktive Umgebung zu schaffen. Dazu gehört nicht nur eine vernünftige Ausstattung mit moderner Hardware und Software, sondern auch Büros, in denen man ungestört arbeiten kann, genügend freie Besprechungsräume und eine angenehme Atmosphäre. Eine nachhaltig Diskussion dieses Themas und weiterführende Literatur ist unter [DeMa97, DL99, OHJ+99] zu finden.

Qualitätssicherung

Die Qualitätssicherung innerhalb eines Projektes ist eine Dienstleistung für das Projektmanagement. Die Prüfung der im Verlauf eines Projektes anfallenden Produkte ist für den Projektleiter eine der wichtigsten Informationen für die Steuerung des Projektablaufs. Der Projektplan muss daher genau festlegen, welche Artefakte zu welchem Zeitpunkt mit welchen Mitteln bzw. Methoden zu prüfen sind. Die Qualitätssicherung kennt dazu zwei zentrale Produkte: den Qualitätssicherungsplan und den Prüfplan. Der Qualitätssicherungsplan besagt, welche Projektergebnisse mit welchen Mitteln bzw. Methoden geprüft werden sollen. Der Prüfplan legt fest, wann eine Prüfung gemäß Vorgabe des Qualitätssicherungsplan durchzuführen ist und von welchen Personen bzw. Rollen sie durchgeführt werden soll. Dabei ist besonders darauf zu achten, dass die Qualitätssicherung rechtzeitig geplant wird und überprüfbare, realistische Qualitätsziele vereinbart werden.

Änderungsmanagement

Während des Projektverlaufes können von außen, beispielsweise in Form von Wünschen des Auftraggebers oder gesetzliche Änderungen, oder von innen, zum Beispiel durch neue Erkenntnisse über fachliche und technische Zusammenhänge und Begrenzungen, Anforderungen hinzukommen, entfallen oder sich ändern.

Unter Änderungsmanagement bzw. Change Management wird ein Prozess verstanden, durch den diese Änderungswünsche vorgeschlagen, aufgenommen, bewertet und angenommen oder abgelehnt werden. Dabei ist eine lückenlose Dokumentation und Verfolgung der Änderungs-

wünsche besonders wichtig. Das Änderungsmanagement ist insbesondere bei großen Projekten ein zentrales Thema.

Konfigurationsmanagement

Zum Konfigurationsmanagement gehören Aktivitäten zur Durchführung der technischen Systemintegration, der Versionierung und Konfiguration von Entwicklungsergebnissen und die Bereitstellung der dafür notwendigen Infrastruktur und Entwicklungsumgebung. Kleinere, vorwiegend projektspezifische Unterstützungsleistungen, wie zum Beispiel die Anpassung von Generierungsskripten oder Werkzeugen, sind ebenfalls Bestandteil des Konfigurationsmanagements.

Wesentliches Ziel ist dabei sicherzustellen, dass ein Produkt bzw. Teilprodukt der Softwareentwicklung jederzeit eindeutig identifizierbar ist. Es muss gewährleistet sein, dass man bei Bedarf auf verschiedene funktionsfähige Versionen eines Systems bzw. Teile des Systems zugreifen kann.

2.1.5 Werkzeugunterstützung

Vom Einsatz moderner Werkzeuge werden hohe Produktivitätssteigerungen erwartet. Die Hersteller unterstützten diese Erwartungen zusätzlich mit teilweise unrealistischen Verheißungen. In der Praxis muss man sich im klaren sein, dass viele Versprechungen nicht haltbar sind. Jedes Werkzeug erfordert eine Einarbeitungszeit, bevor man produktiv damit umgehen kann. Schulungen sind meist unerlässlich. Sie ermöglichen eine optimale Ausnutzung des Werkzeuges. Trotzdem hat jedes Werkzeug seine Einschränkungen.

Den übertriebenen Hoffnungen der Softwareingenieure ist meist eine pragmatische Sicht gewichen. Die Notwendigkeit des Einsatzes maschineller Unterstützung in der Softwareentwicklung ist heute jedoch mehr denn je unbestritten. Dabei sind Werkzeuge in allen Bereichen der Softwareentwicklung sinnvoll und auch kommerziell verfügbar, also in den Bereichen: Vorgehensmodell, Beschreibungstechniken und Systemmodell, Technologie und Architektur sowie Managementpraktiken.

So werden Werkzeuge beispielsweise verwendet für die Modellierung, zur Erstellung von Dokumentation, für das Konfigurationsmanagement, für das Änderungsmanagement, für die Programmgenerierung, für das Tailoring von Prozessmodellen, zur Unterstützung des Prozessmodells selbst, für die Projektplanung und für die Projektanalyse. Bei der Auswahl eines Werkzeuges sollte man nicht nur die funktionalen Anforderungen an das Werkzeug im Auge behalten, sondern auch technische und organisatorische Fragestellungen. So ist zum Beispiel die Marktverbreitung oder die Serviceleistungen ein durchaus wichtiger Faktor, den man bei der Kaufentscheidung mit betrachten muss.

Im Kapitel 8 werden wir grundlegende Konzepte einer umfassenden und durchgängigen Werkzeugunterstützung für den evolutionären Architekturentwurf eingehender diskutieren. Darauf aufbauend präsentieren wir Design- und Architekturvarianten ausgewählter Werkzeuge, die für den evolutionären Architekturentwurf von zentraler Bedeutung sind.

2.2 Methodisches Rahmenwerk dieser Arbeit

Die Entwicklung und Wartung von Software ist eine anspruchsvolle Aufgabe. Projektleiter müssen ihr Team sicher durch die verschiedenen Stufen der Softwareentwicklung führen. Als

Anleitung stehen den Projektleitern hierfür verschiedenste Methoden und Vorgehensmodell zur Verfügung, wie zum Beispiel, der Unified Software Development Process [JBR99, Kruc00] oder das V-Modell ’97 [DW99].

Methoden, Vorgehensmodelle und Softwareentwicklungsprozesse sind schwer zu verstehen und zu managen. Eine große „Methodology“ ist weder realistisch anwendbar noch praktikabel. Die individuellen Anforderungen und Randbedingungen in Projekten sind sehr unterschiedlich. Damit der Projekterfolg nicht gefährdet wird, müssen sie berücksichtigt werden.

In diesem Kapitel werden wir unsere Vorstellung einer kleinen „methodology“ präsentieren. Das Framework unserer Methodik ist nicht auf die komponentenbasierte Softwareentwicklung beschränkt sondern von genereller Natur. Es bietet eine Plattform, um alle Konzepte zu vereinen, die wir in den vorhergehenden Kapiteln diskutiert haben. Dabei ist es selbst modular aufgebaut, so dass es nahezu beliebig erweitert werden kann.

In den folgenden Kapiteln zeigen wir zuerst zwei unterschiedliche Sichtweisen auf Methoden und Vorgehensmodelle: die Sichtweise der Mitarbeiter in den Entwicklungsprojekten und die Sichtweise der Methodikgruppe in einem Unternehmen. Darauf aufbauend präsentieren wir dann die unterschiedlichen Modellebenen von Methoden und Vorgehensmodellen. Nach dieser „Ortsbestimmung“ stellen wir unser methodisches Rahmenwerk vor, das die Integrationsplattform für die restliche Arbeit zur Verfügung stellt.

2.2.1 Methoden und Vorgehensmodelle aus der Projektsicht

Heutzutage stehen in den Unternehmen typischerweise mehr oder weniger weit ausgearbeitete Kochbücher, Richtlinien und Leitfäden für die erfolgreiche Projektabwicklung zur Verfügung. Projektleiter und Mitglieder von Projektteams verwenden diese bei ihrer täglichen Arbeit. In den Kochbüchern, Richtlinien und Leitfäden ist dabei meist ein fest definiertes Produktmodell enthalten. Wie in Abbildung 2.5 dargestellt, kann man sich das Produktmodell als einen leeren Schrank vorstellen, den Projektschrank oder die Projektbibliothek (vgl. auch [Dene91, OHJ+99]). Auf jeder Schublade ist detailliert beschrieben, was die Entwickler in diese Schublade zu füllen haben, aber nicht wie.

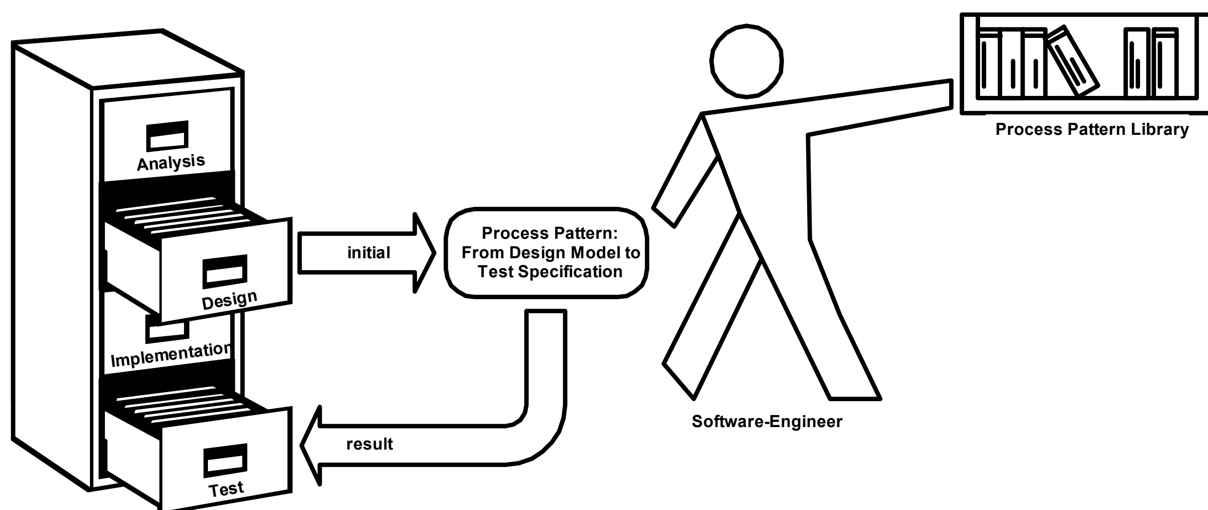

Abbildung 2.5: Die Projektsicht auf den prozessmusterbasierten Ansatz

Die initiale Definition des Schrankes erfolgt nicht immer von Grund auf neu. Vielmehr werden zu Beginn des Projektes, die für das Projekt relevanten Produkte ermittelt. Dies geschieht durch Auswahl aus einer Reihe von Vorlagen. Dieser Vorgang wird auch als Tailoring be-

zeichnet. Diese Art von Tailoring existiert bei allen generischen Prozessmodellen, wie zum Beispiel beim V-Modell ’97 [DW99] und beim Unified Software Development Process [JBR99, Kruc00]. Dort umfasst das Tailoring allerdings auch die Auswahl der Aktivitäten, also das „Wie“ der Produkterstellung.

Dies ist beim Prozessmusteransatz anders [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b]. Eine Vorauswahl, welche Prozessmuster in einem Projekt zur Anwendung kommen sollen, wird bewusst nicht getroffen. Das Tailoring des Vorgehensmodells bezüglich der Aktivitäten wird nicht zu Beginn des Projektes durchgeführt. Vielmehr soll dem Projektleiter und dem Projektteam die Möglichkeit gegeben werden, bedingt durch die konkrete Projektsituation bestimmte Prozessmuster auszuwählen.

Die Prozessmuster geben Tipps und Tricks zu den einzelnen Arbeitsschritten, die sich in der Praxis bewährt haben. Sie sind pragmatische Anleitungen für die Abwicklung einzelner Aktivitäten in einem Projekt. Die Reihenfolge der Ausführung von Prozessmustern wird aber bewusst offen gelassen. Beispielsweise könnte ein Prozessmuster beschreiben, wie man ausgehend von einem Designmodell konstruktiv Testfälle für die Testspezifikation erstellen kann (vgl. Abbildung 2.5). Ob und wann dieses Prozessmuster ausgeführt wird, ist aber nicht festgelegt, sondern wird vom Projektleiter zusammen mit dem Projektteam bei Bedarf entschieden.

Kochbücher, Richtlinien und Leitfäden, die in Unternehmen zur Verfügung stehen, enthalten bereits solche Prozessmuster, in der einen oder anderen Form. In ihrer täglichen Arbeit wenden Softwareingenieure diese Prozessmuster an und füllen so den Projektschrank. Allerdings unterscheiden diese Kochbücher, Richtlinien und Leitfäden nicht explizit die grundlegenden Konzepte von Vorgehensmodellen, wie Produkt und Prozessmuster. Im Vergleich zum prozessmusterbasierten Ansatz sind sie deshalb auch nicht ausreichend modular aufgebaut und bieten noch nicht die notwendige Flexibilität, um das Vorgehensmodell an die spezifischen Bedürfnisse konkreter Projektsituationen anpassen zu können.

2.2.2 Methoden und Vorgehensmodelle aus der Methodiksicht

Die in den Unternehmen verwendeten Kochbücher, Richtlinien und Leitfäden beinhalten sowohl Management- als auch Engineering-Aktivitäten, sind dokumentiert, standardisiert und in die anderen Unternehmensprozesse integriert. Meist werden sie im Rahmen eines speziellen Projektes einmalig definiert und festgelegt. So könnte ein Softwarehaus festlegen, dass es eine speziell, für die Bedürfnisse des Unternehmens, angepasste Version des V-Modells ’97 als Standardprozess verwenden will.

Firmen die eine standardisierte Methodik anwenden sind in dem Capability Maturity Model (CMM) auf Stufe 3 oder höher [PCCW93]. Da sich die Anforderungen an ein Unternehmen, das erfolgreich am Markt arbeitet stetig ändern, genügt eine einmalige Festlegung der Entwicklungsmethodik nicht. Ein kontinuierlicher Verbesserungsprozess ist im Unternehmen zu etablieren. Messbare Rückkopplung und die Pilotierung von innovativen Ideen und Techniken in der Entwicklungsmethodik sind ein essentieller Bestandteil eines Verbesserungsprozesses. Unternehmen mit einem derartigen Verbesserungsprozess sind auf der Stufe 5 im CMM.

Die Entwicklungsmethodik eines Unternehmens wird dann meist von einem speziellen Personkreis oder von einer spezifischen Abteilung, der Methodikgruppe, entwickelt, gepflegt und verbessert. Die Verbesserungs- und Erweiterungsvorschläge für die Methodik und das Vorgehensmodell kommen aus der Methodikgruppe selbst, aber auch von anderen Mitarbeitern im

Unternehmen, die neue Verfahren und Techniken einbringen, die sich in Projekten bewährt haben.

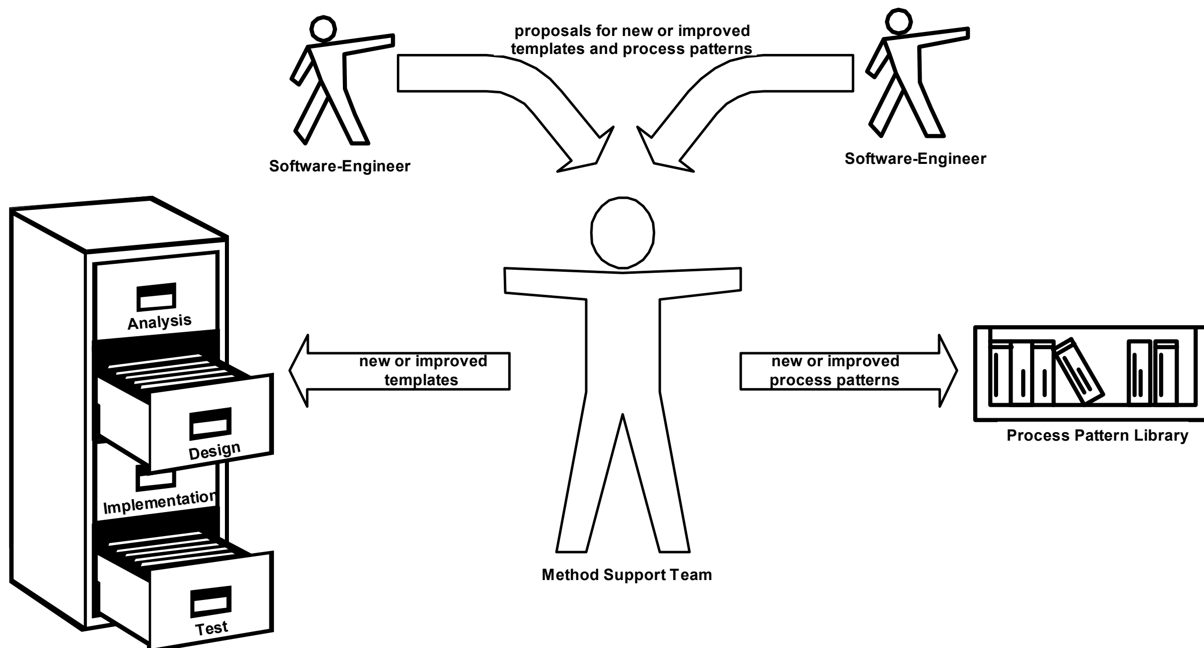

Abbildung 2.6: Die Methodiksicht auf den prozessmusterbasierten Ansatz

Abbildung 2.6 illustriert dieses Prinzip. Die Projekte liefern der Methodikgruppe neue Vorgehensweisen, in Form von Prozessmustern und Produktvorlagen, die sich in den Projekten bewährt haben und noch nicht im Unternehmensstandard integriert sind. Nach einer entsprechenden Qualitätssicherung und Verallgemeinerung durch die Methodikgruppe werden diese in die Standardmethodik und das Standardvorgehensmodell des Unternehmens integriert. Diese stetige Rückkopplung durch die Projekte ermöglicht eine adäquate und zielgerichtete Weiterentwicklung der Methodik und des Vorgehensmodells im Unternehmen.

2.2.3 Über Methoden, Modelle und Metamodelle

Methoden, Vorgehensmodelle und Softwareentwicklungsprozesse sind ein sehr komplexes Thema. Unterschiedliche Personen sind darin involviert. So steht beispielsweise für den Projektleiter und sein Projektteam das Vorgehen und die anstehenden Aufgaben ihres spezifischen Projektes im Vordergrund. Projektleiter konzentrieren sich dabei auf die Managementaufgaben in ihrem Projekt, beispielsweise auf die Fortschrittskontrolle oder auf die Planung der nächsten Entwicklungsiteration. Softwareentwickler beschäftigen sich mit der Programmierung einer Komponente oder dem Entwurf einer Klassenhierarchie für ihr Projekt. Die Methodikgruppe dagegen definiert, dokumentiert und pflegt die Methodik sowie das Vorgehensmodell und stellt es den Projekten zur Verfügung.

Alle diese unterschiedlichen Personen und Gruppen haben verschiedene Sichten auf ein Projekt und die zugrundeliegenden methodischen Rahmenbedingungen. Abbildung 2.7 illustriert diese verschiedenen Ebenen und Begriffswelten nochmals im Zusammenhang.

Die Ebene der Instanzen (*Instance Level*) ist identisch mit der eines konkreten Projektes. Das Analysedokument eines Projektes, ein Projektplan, die Durchführung eines Testlaufes und der Programmcode sind Beispiele von Elementen auf dieser Ebene.

Auf CMM Stufe 3 oder höher müssen diese Elemente einem standardisierten Entwicklungsprozess und einer standardisierten Entwicklungsmethodik folgen, die auf der Modellebene (*Model Level*) liegen. Die Festlegung der Struktur von Architekturdokumenten, die Beschreibung des Testverfahrens, Programmierrichtlinien und die Festlegung von Beschreibungstechniken liegen beispielsweise auf dieser Modellebene.

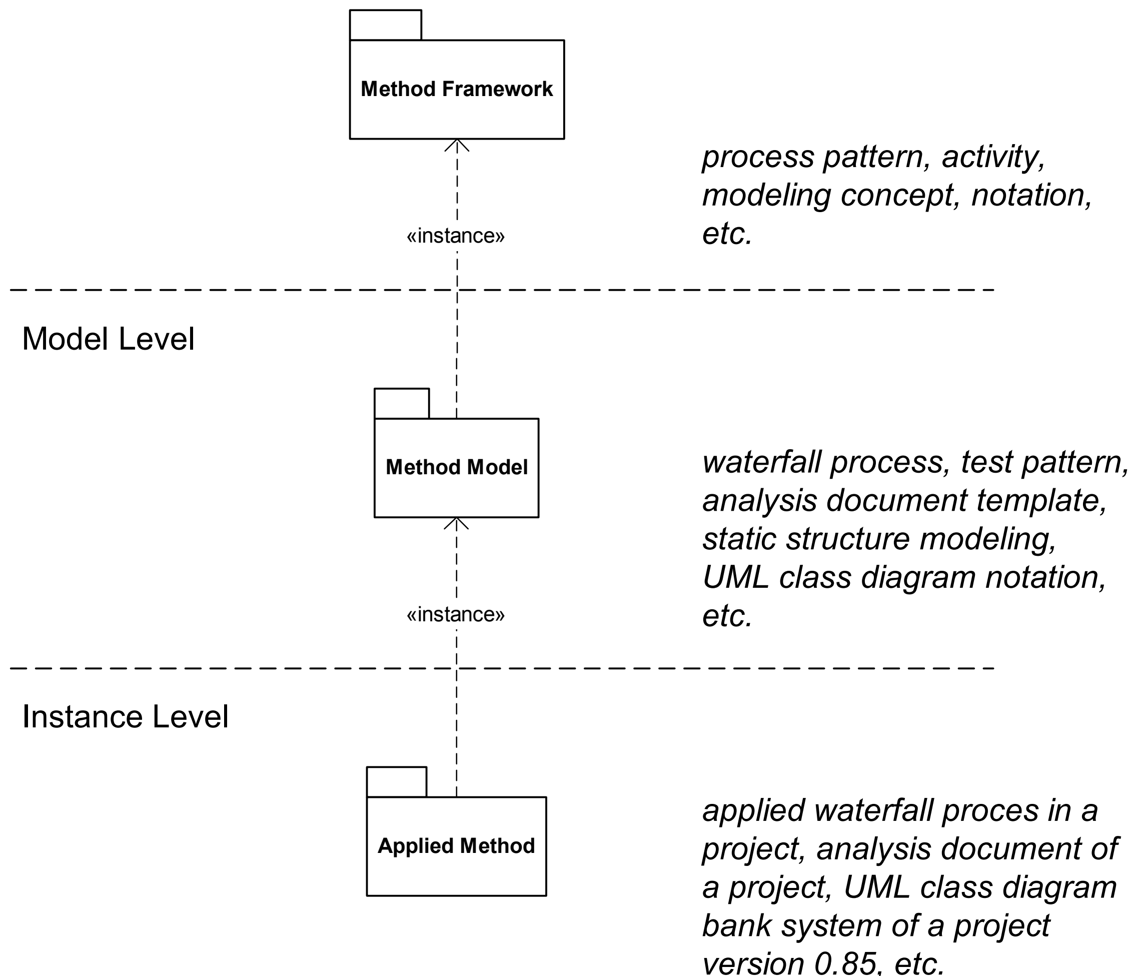

Abbildung 2.7: Die verschiedenen Ebenen der Methoden und Vorgehensmodelle

Unternehmen der CMM Stufe 5 müssen quantitative Aussagen über Veränderungen auf der Modellebene treffen können. Nur so ist es möglich festzustellen, ob eine bestimmte Veränderung in der Entwicklungsmethodik zu einer Verbesserung geführt hat. Dazu ist es notwendig eine Metamodellebene (*Metamodel Level*) einzuführen. Konzepte, wie zum Beispiel Aktivität, Notation, Modellierungskonzept und Systemmodell, liegt auf dieser Metamodellebene.

Anzumerken ist, dass diese Metamodellierung dem generellen Ansatz der Meta Object Facility (MOF) Spezifikation der Object Management Group (OMG) folgt [OMG00b]. Nach diesem Ansatz ist ein Metamodell auf der Stufe n stets eine Instanz des Metamodells der nächst höheren bzw. abstrakteren Stufe n+1.

2.2.4 Das prozessmusterbasierte Metamodell für Entwicklungsmethoden

In dieser Arbeit betrachten wir einen Teilaspekt einer umfassenden Entwicklungsmethodik, den evolutionären Architekturentwurf komponentenbasierter Systeme. Damit wir offen für die Integration weiterer methodischer Arbeiten und Konzepte sind, führen wir ein Methodik-Framework ein. Dieses methodische Rahmenwerk liegt in der Metamodellebene in Abbildung 2.7. Unsere (Teil-)Entwicklungsmethodik des evolutionäre Architekturentwurfs, die wir in den Kapiteln 2.3 und 2.4 vorstellen, basiert auf diesen Konzepten. Dementsprechend ist die Methodik der Modellebene in Abbildung 2.7 zuzuordnen. Dieser Ansatz ermöglicht es uns und anderen, die in dieser Arbeit vorgestellte Methodik zu erweitern und mit anderen Methoden zu integrieren.

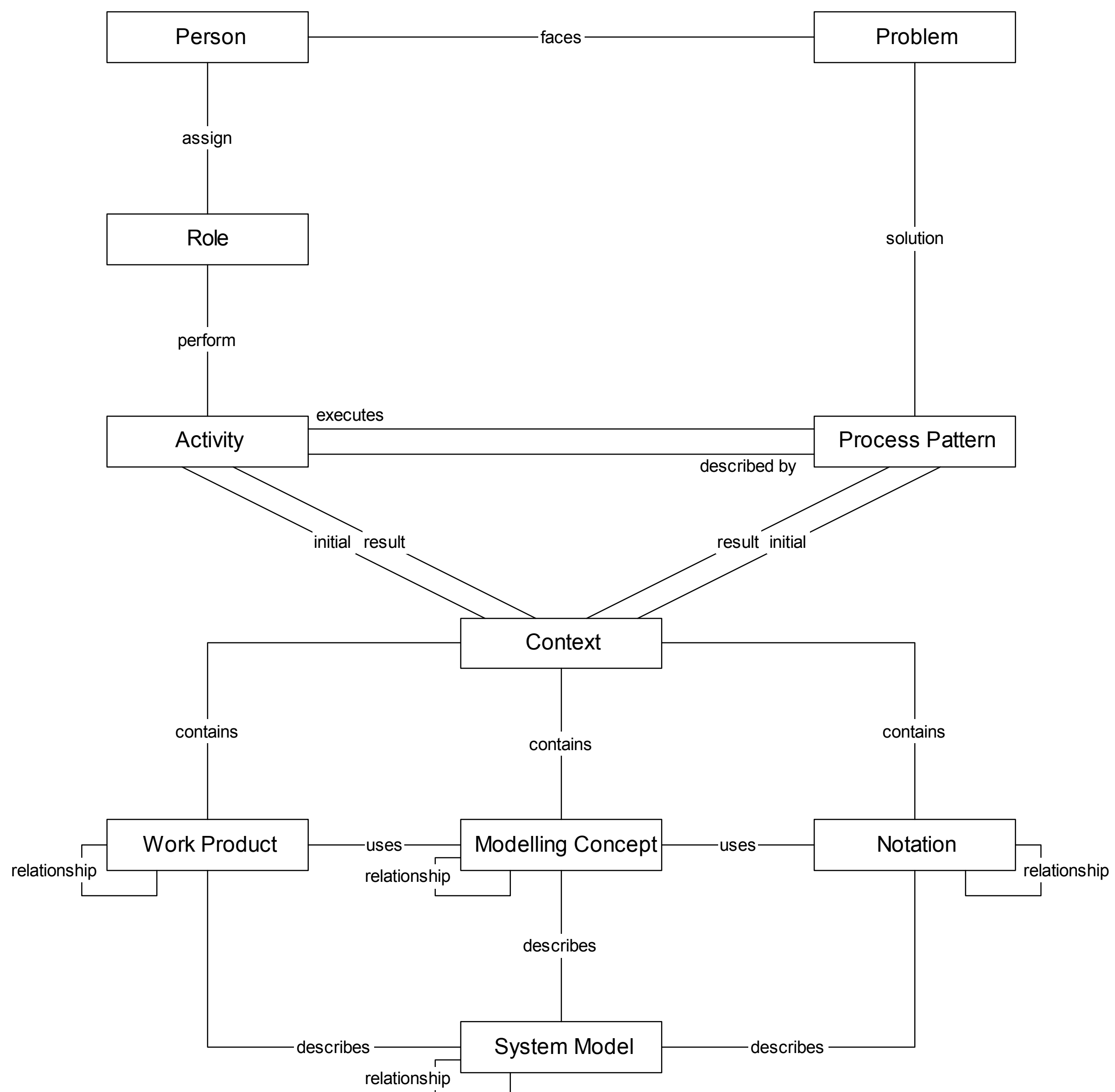

Abbildung 2.8: Prozessmusterbasiertes Metamodell für Entwicklungsmethoden

Abbildung 2.8 zeigt das prozessmusterbasierte Metamodell für Entwicklungsmethoden in Form eines UML Klassendiagramms. Es stellt die zentralen Konzepte und Begriffe im Zusammenhang dar, die notwendig sind, um prozessmusterbasierte Methoden und Vorgehensmodelle zu definieren.

Im Kern ist dieses Diagramm zusammengesetzt aus den zwei Klassendiagrammen in Abbildung 2.2 und Abbildung 2.3, die wir bereits in den vorhergehenden Kapiteln diskutiert haben. Es wurde nur eine neue Klasse eingeführt: *Context*. Der Kontext verfeinert die ursprüngliche Beziehung zwischen Aktivitäten und Produkten sowie zwischen Prozessmustern und Produkten aus Abbildung 2.2. Diese Klasse ermöglicht es bestimmte Kontextbedingungen über eine Menge von Produkten zu formulieren, die notwendig sind, damit eine Aktivität ausgeführt werden kann.

Beispielsweise könnte die Aktivität „Beschreibung der Benutzer-System-Interaktionen“ als Eingabe die Anwendungsfälle des Systems benötigen. Die „Anwendungsfallspezifikation“ muss aber bestimmten Bedingungen genügen, damit die Aktivität „Beschreibung der Benutzer-System-Interaktion“ durchgeführt werden kann: Es sollten möglichst viele Anwendungsfälle bereits identifiziert sein (> 50%), in Form eines Anwendungsfalldiagramm vorliegen (Anwendungsdiagramm existiert) und jeder Anwendungsfall zumindest in Prosa beschrieben sein (textbasierte Anwendungsfallspezifikation existiert). Derartige Kontextbedingungen können in den Instanzen der Klasse Kontext formuliert werden.

2.3 Modellbasierte Entwicklung und Softwarearchitekturmodelle

Während der Entwicklung eines Softwaresystems werden eine Vielzahl von unterschiedlichen Produkten erstellt, unabhängig von dem Entwicklungsprozess, der Programmiersprache, der Zielarchitektur, der technischen Infrastruktur oder irgend einem anderen Bestandteil einer methodischen Softwareentwicklung.

Entwicklungsdokumente sind spezielle Produkte, die in der Softwareentwicklung entstehen. Sie beschreiben das Softwaresystem auf einer bestimmten Abstraktionsebene oder Sicht. Inhaltlich und methodisch motiviert werden Entwicklungsdokumente zu Modellen zusammengefasst.

In diesem Kapitel untersuchen wir zuerst den Lebenszyklus dieser Modelle im Kontext einer modellbasierten Softwareentwicklung. Dabei lassen sich verschiedene Dimensionen der Modellveränderung festhalten. Im Rahmen dieser Arbeit sind wir besonders an der Evolutionsdimension des Architekturmodells interessiert. In den nächsten Kapiteln beschreiben wir die verschiedenen Sichten in einem Architekturmodell und die dabei verwendeten Modellierungskonzepte. Schließlich können wir dann die einzelnen Bestandteile eines Architekturmodells bestimmen und im Kontext eines übergreifenden Produktmodells der Softwareentwicklung, des Projektschrankes, einordnen.

2.3.1 Lebenszyklus von Softwaresystemen und deren Modelle

Die Modellbildung ist eine übliche Technik, um mit der Komplexität von Softwaresystemen umzugehen. Ein Modell eines Softwaresystems wird durch eine Menge von Entwicklungsdokumenten beschrieben. Beispiele solcher Entwicklungsdokumente sind das Systemarchitekturdokument, ein Klassendiagramm oder der Programmcode einer Klasse. Das Modell eines Softwaresystems ist somit der Teil des Projektschrankes, der das Softwaresystem selbst beschreibt.

Während der Entwicklung eines Systems wird das zugehörige Modell stetig verbessert. Es gibt bestimmte Zustände des Modells, die besonders wichtig sind und deshalb dauerhaft festgehalten werden. Wird beispielsweise ein Softwaresystem integriert und der Systemtest steht an, dann wird der aktuelle Stand des Modells archiviert und sicher aufbewahrt. Typischerwei-

se wird die Modellarchivierung immer dann vorgenommen, wenn ein Meilenstein erreicht worden ist. Darüber hinaus werden aber auch andere Zustände des Modells während des Projektverlaufes fixiert und dauerhaft gespeichert.

Wie in Abbildung 2.9 illustriert, kann man im Laufe des Lebenszyklus eines Softwaresystems Modellveränderungen in drei Dimensionen beobachten (siehe auch [TS00]). In der Entwicklungsdimension entsprechen die Modellversionen meist den Meilensteinen, die im Projektplan definiert sind und erreicht wurden, zum Beispiel die Fertigstellung des Analysemodells oder des Designmodells. Diese Modelle unterscheiden sich in der Abstraktionsebene.

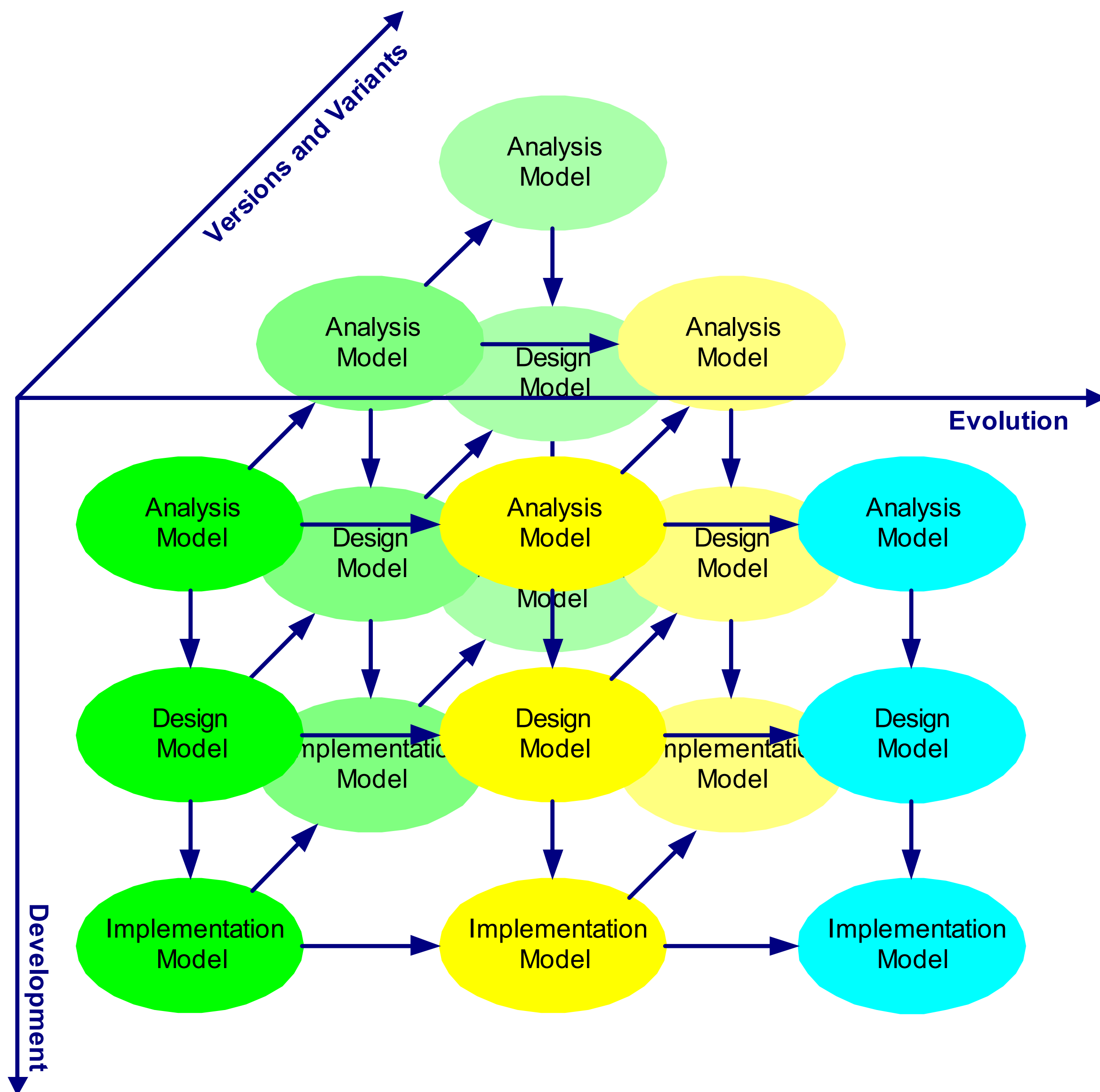

Abbildung 2.9: Die drei Dimensionen im Lebenszyklus der Modelle eines Systems

In der Evolutionsdimension betrachten wir die Modelländerungen, die sich im Laufe der iterativen Entwicklung eines Softwaresystems ergeben. Beispielsweise das Modell des ersten Prototypen, das Modell mit zusätzlicher Integration von COM+ und CORBA oder das Modell mit den neuen, veränderten Kundenanforderungen vom 26.6.2001. Diese Modelländerungen können auf allen Stufen in der Entwicklungsdimension auftauchen.

In der Versionen-/Varianten-Dimension betrachten wir die Modellversionen, die sich aus neuen Versionen oder Varianten eines Systems ergeben. Beispielsweise wird für die Entwicklung der Version 7.8 eines Systems das Modell der Version 7.0 kopiert und in einer neuen Modellversion angelegt. Typischerweise werden in dieser Dimension nicht so häufig Modelle erzeugt, wie in den anderen Dimensionen.

Wesentlich für diese Arbeit ist, dass zwischen diesen Modellen bestimmte Beziehungen bestehen. So sollte beispielsweise die Beziehung zwischen dem Analysemodell und dem Designmodell eine Verfeinerung sein. Bewegt man sich auf den anderen Dimensionen, so sind die Beziehungen zwischen den Modell meist keine Verfeinerungen. Zwei Modelle, die sich auf der Versionen-/Varianten-Dimension oder der Evolutionsdimension unterscheiden, unterliegen im Allgemeinen keiner Verfeinerungsbeziehung (vgl. auch Kapitel 3.2).

Eine eingehende Betrachtung der methodischen Aspekte in der Versionen-/Varianten-Dimension ist in den Arbeiten von Bernd Deifel zu finden (siehe auch [Deif01]). Mit der Entwicklungsdimension haben sich bereits eine Reihe von Dissertationen, die am Lehrstuhl von Professor Dr. Manfred Broy erstellt wurden, auseinandergesetzt. Stellvertretend sei hier auf die Dissertationen von Dr. Bernhard Rumpe [Rump96], Dr. Klaus Bergner [Berg97] und Dr. Ingolf Krüger [Krüg00] verwiesen.

Der Fokus dieser Arbeit liegt in der Evolutionsdimension. Dabei halten wir die Versionen-/ Varianten-Dimension fest und beschränken uns in der Entwicklungsdimension auf das Architekturmodell: Im Zentrum der vorliegenden Dissertation steht der evolutionäre Entwurf von Architekturmodellen komponentenbasierter Systeme.

2.3.2 Sichten in einem Softwarearchitekturmodell

Architekturmodelle beschreiben die Softwarearchitektur eines Systems. Bei kleinen Projekten ist die Softwarearchitektur häufig mehr oder weniger vorgegeben. Große Softwaresysteme und deren Softwarearchitekturen hingegen sind sehr komplexe Gebilde. Verschiedene Personengruppen mit unterschiedlichen Interessen sind an der Entwicklung des Architekturmodells beteiligt. Deshalb besteht eine Architekturmodell typischerweise aus unterschiedlichen Sichten auf die Architektur des Softwaresystems. Diese Sichten sind dabei auf die spezifischen Bedürfnisse bestimmter Personengruppen zugeschnitten (siehe auch MEH01).

So werden beispielsweise beim Rational Unified Process [Kruc00] fünf unterschiedliche Sichten des Architekturmodells unterschieden: Use-Case View, Logical View, Process View, Implementation View und der Deployment View. Eine anderes Beispiel ist die Arbeit von Christine Hofmeister, Robert Nord und Dilop Soni [HNS99]. Dort werden die Sichten Conceptual Architecture View, Module Architecture View, Execution Architecture View und Code Architecture View definiert.

Eine vereinheitlichte und allgemein anerkannte Definition der verschiedenen Sichten, die in einem Architekturmodell enthalten sein sollten, existiert aber noch nicht. Es lassen sich jedoch gewisse Parallelen in den verschiedenen Ansätzen finden. Im Zuge einer Vielzahl eigener Projekte haben sich hierbei die folgenden Sichten für Softwarearchitekten bewährt, die in Abbildung 2.10 im Zusammenhang dargestellt werden:

Logische Architektur (Logical Architecture)

Diese Sicht beschreibt die grundlegenden Architekturrichtlinien eines Systems. Die konzeptionellen Komponenten des Systems, deren Schnittstellen und die Beziehungen und Interaktionen der Komponenten werden dabei spezifiziert.

Die logische Architektur entspricht dem Logical View in [Kruc00] sowie dem Conceptual Architecture View und Module Architecture View in [HNS99].

Verteilungsarchitektur (Distribution Architecture)

Die Verteilungsarchitektur beschreibt die physische Verteilung der konzeptionellen Komponenten der logischen Architektur auf Ausführungseinheiten und Prozesse. Die logische Architektur legt somit die Verteilungsmöglichkeiten fest. Mit der Verteilungsarchitektur wird eine konkrete Variante daraus ausgewählt.

Die Verteilungsarchitektur entspricht dem Process View in [Kruc00] sowie dem Execution Architecture View in [HNS99].

Technische Architektur (Technical Architecture)

Bei der technischen Architektur werden für die konzeptionellen Komponenten der logischen Architektur spezifische Implementierungen ausgewählt. Diese Implementierungen können kommerziell verfügbare Standardkomponenten, eigene properitäre wiederverwendbare Komponenten, oder noch zu implementierende objektorientierte bzw. prozedurale Teilsysteme sein.

Die technische Architektur entspricht dem Implementation View und Deployment View in [Kruc00] sowie dem Code Architecture View in [HNS99].

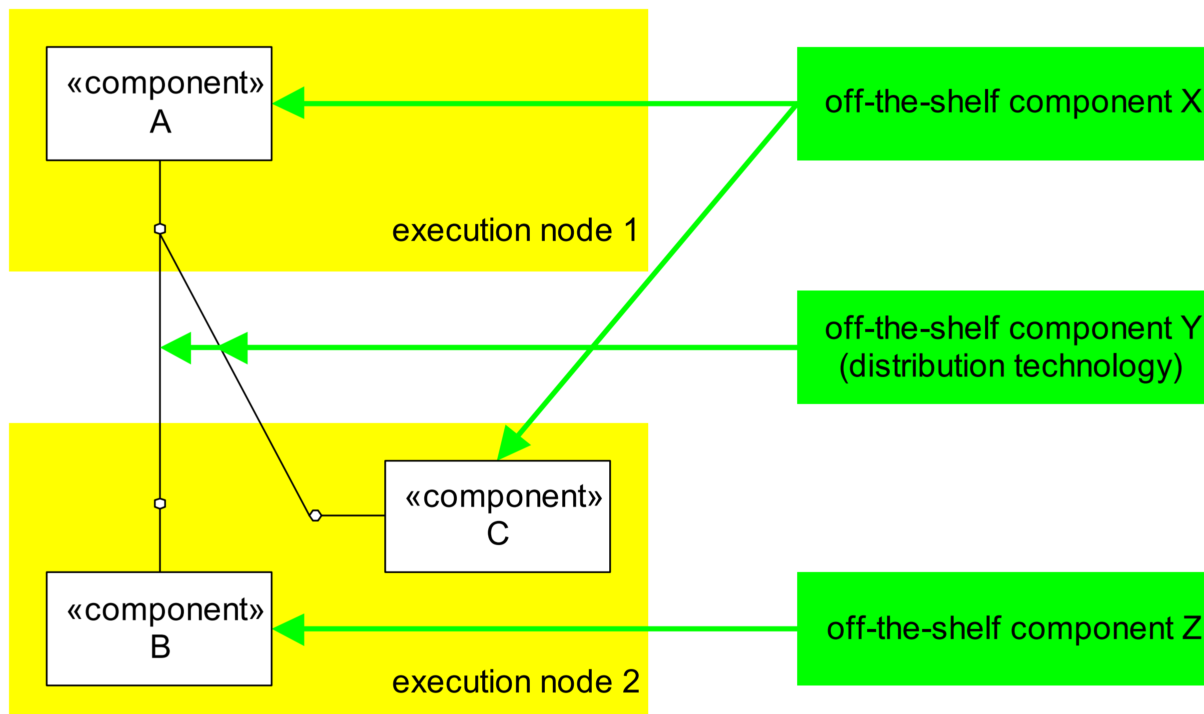

Abbildung 2.10: Die Verschiedenen Sichten eines Architekturmodells

Für die vorliegende Arbeit ist dabei aber wesentlich, dass die logische Architektur die Rahmenbedingungen für die anderen Architektursichten festlegt. Diese sind Verfeinerungen der

logischen Architektur. Deshalb werden wir uns im Rahmen dieser Arbeit hauptsächlich mit der logischen Architektur befassen.

2.3.3 Modellierungskonzepte in Softwarearchitekturmodellen

Sichten in einem Architekturmodell sind aus methodischen Gründen motiviert. Unterschiedliche Aspekte sollen in den Sichten hervorgehoben werden. Für die Beschreibung dieser Sichten eigenen sich ähnliche und teilweise sogar identische Modellierungskonzepte. Innerhalb eines Softwarearchitekturmodells unterscheiden wir die folgenden Modellierungskonzepte:

Komponentenbeschreibungen (Component Description)

Die Beschreibung der Komponenten in einem Architekturmodell beinhaltet eine informelle, meist testbasierte, Beschreibung der Funktionalitäten und Verantwortlichkeiten der einzelnen Komponenten. Ziel dieser Komponentenbeschreibung ist es, alle Komponenten zu identifizieren sowie ein Gefühl und erstes Verständnis für den Aufgabenbereich der Komponenten zu erhalten. Dabei werden insbesondere die funktionalen und nicht funktionalen Anforderungen aus dem Analysemodell den einzelnen Komponenten zugeordnet. Somit ist die Verfolgbarkeit der Anforderungen von dem Analysemodell zum Architekturmodell möglich.

Strukturspezifikationen (Structure Specification)

Die Beschreibung der Komponentenstruktur von Softwarearchitekturen erfolgt mit Hilfe von Strukturspezifikationen. Die Struktur besteht aus den Komponenten und Schnittstellen des Systems sowie die Beziehungen zwischen den Komponenten. Beispiele für solche Beziehungen sind Vererbung, Aggregation oder einfache Verbindungen. Die wohl bekannteste Notation dieses Modellierungskonzeptes sind UML Klassendiagramme [BJR98, RJB98, OMG00a]. Aber auch viele der Architectural Description Languages (ADL) [SG96] oder die Familie der klassischen Entity/Relationship Diagramme [Chen76] gehören in diese Kategorie.

Schnittstellenspezifikationen (Interface Specification)

Schnittstellenspezifikationen ermöglichen es dem Architekten Schnittstellen von Komponenten möglichst präzise und vollständig zu spezifizieren. Die meisten Techniken für die Schnittstellenspezifikation, die in der Praxis eingesetzt werden, erlauben nur eine Spezifikation der syntaktischen Schnittstelle, wie zum Beispiel CORBA IDL [OMG01b] oder UML Schnittstellen [BJR98, RJB98, OMG00a]. Weiterführende Spezifikationstechniken basieren meist auf Konzepten für Vor- und Nachbedingungen, wie zum Beispiel in Eiffel [Meye97] oder in der Java Modeling Language [LBR99]. Diese bieten eine präzisere Spezifikation des Verhaltens von Schnittstellenmethoden. Das Verhalten von Komponenten kann so besser beschrieben werden. Eine vollständige Beschreibung des Verhaltens ist aber mit diesen Spezifikationstechniken noch nicht möglich (siehe auch [BW97]).

Interaktionsspezifikationen (Interaction Specification)

Mit Hilfe von Interaktionsbeschreibungen kann der Architekt die Interaktion zwischen den Schnittstellen von Komponenten beschreiben. Typische Interaktionen zwischen Komponenten ist der Austausch von Nachrichten oder der Aufruf von Diensten. Beispiele für Notationen basierend auf diesem Modellierungskonzept sind UML Sequenzdiagramme [BJR98, RJB98, OMG00a], Extended Event Traces [BHK+97] oder Message Sequence Charts

[ITU94]. Der methodische Einsatz dieser Beschreibungen, sowie deren semantische Bedeutung ist aber bei diesem Modellierungskonzept noch relativ unklar. So kann man Protokoll Beschreibungen als rein exemplarische Beschreibungen verstehen, wie zum Beispiel in der UML [BJR98, RJB98, OMG00a], oder man kann sie auch zur vollständigen Spezifikation des Komponentenverhaltens verwenden, wie beispielsweise in der Dissertation von Dr. Ingolf Krüger (siehe [Krüg00]).

Implementierungsspezifikation (Implementation Specification)

Die Beschreibung der Realisierung von Komponenten hat im Kontext von Componentware einen besonderen Stellenwert. Die Implementierung einer Komponente kann und soll hierbei durch andere komponentenbasierte Systeme beschrieben werden. Dementsprechend kann die Realisierungsspezifikation einer Komponente aus einem weiteren Architekturmodell bestehen. Die Beschreibung der Implementierung von atomaren Komponenten, den kleinsten Einheiten, kann natürlich durch Programmcode erfolgen. Ebenso sind auch abstraktere Beschreibungen wie zum Beispiel Automaten (siehe auch [Rump96]) oder Interaktionsdiagramme denkbar (siehe auch [Krüg00]).

2.3.4 Aufbau und Struktur eines Architekturmodells

Im Laufe der Entwicklung eines Softwaresystems werden unterschiedliche Produkte erstellt. Abbildung 2.11 zeigt die Struktur und den Zusammenhang dieser Produkte in einem UML Klassendiagramm. Beinhaltet ein Produkt ein anderes so wird dies jeweils durch die gerichtete Beziehung *consists* dargestellt.

Abbildung 2.11 entsprechend beinhaltet der Projektschrank aus Kapitel 2.2 eine Menge von untergeordneten Produkten. Modelle sind spezielle Produkte, die das Softwaresystem beschreiben. Das Architekturmodell ist eine wesentliche Ausprägung eines solchen Modells. Es besteht selbst wieder aus Teilprodukten, die spezielle Sichten auf die Architektur beinhalten. Die logische Architektur ist eine der zentralen Architektursichten.

In Abbildung 2.11 sind die Struktur und die einzelnen Bestandteile der Beschreibung logischer Architekturen detaillierter aufgeschlüsselt. Dementsprechend besteht die Spezifikation einer logischen Architektur aus einer Menge von Architekturskizzen, Komponenten- und Schnittstellenspezifikationen, Testspezifikationen, Architekturprototypen, Testberichten und Realisierungsspezifikationen. Letztere können dabei rekursiv weitere Spezifikationen logischer Architekturen enthalten.

Für die Beschreibung der einzelnen Bestandteile einer logischen Architekturspezifikation werden unterschiedliche Modellierungskonzepte verwendet. In Abbildung 2.11 ist über die Beziehung *uses* jedem dieser Bestandteile ein entsprechendes Modellierungskonzept zugeordnet. Abbildung 2.11 legt somit nicht nur die Struktur und den Aufbau von Architekturspezifikationen fest, sondern auch die dabei verwendeten Modellierungskonzepte.

Das Klassendiagramm in Abbildung 2.11 ist der Modellebene in Abbildung 2.7 zuzuordnen. Instanzen der Klassen repräsentieren Teile von Architekturspezifikationen eines konkreten Projektes und liegen somit auf der Instanzebene in Abbildung 2.7. Beispielsweise ist die Architekturspezifikation eines Projektes mit der Bezeichnung *ArchSpec_V_0_8* eine Instanz der Klasse *Logical Architecture*.

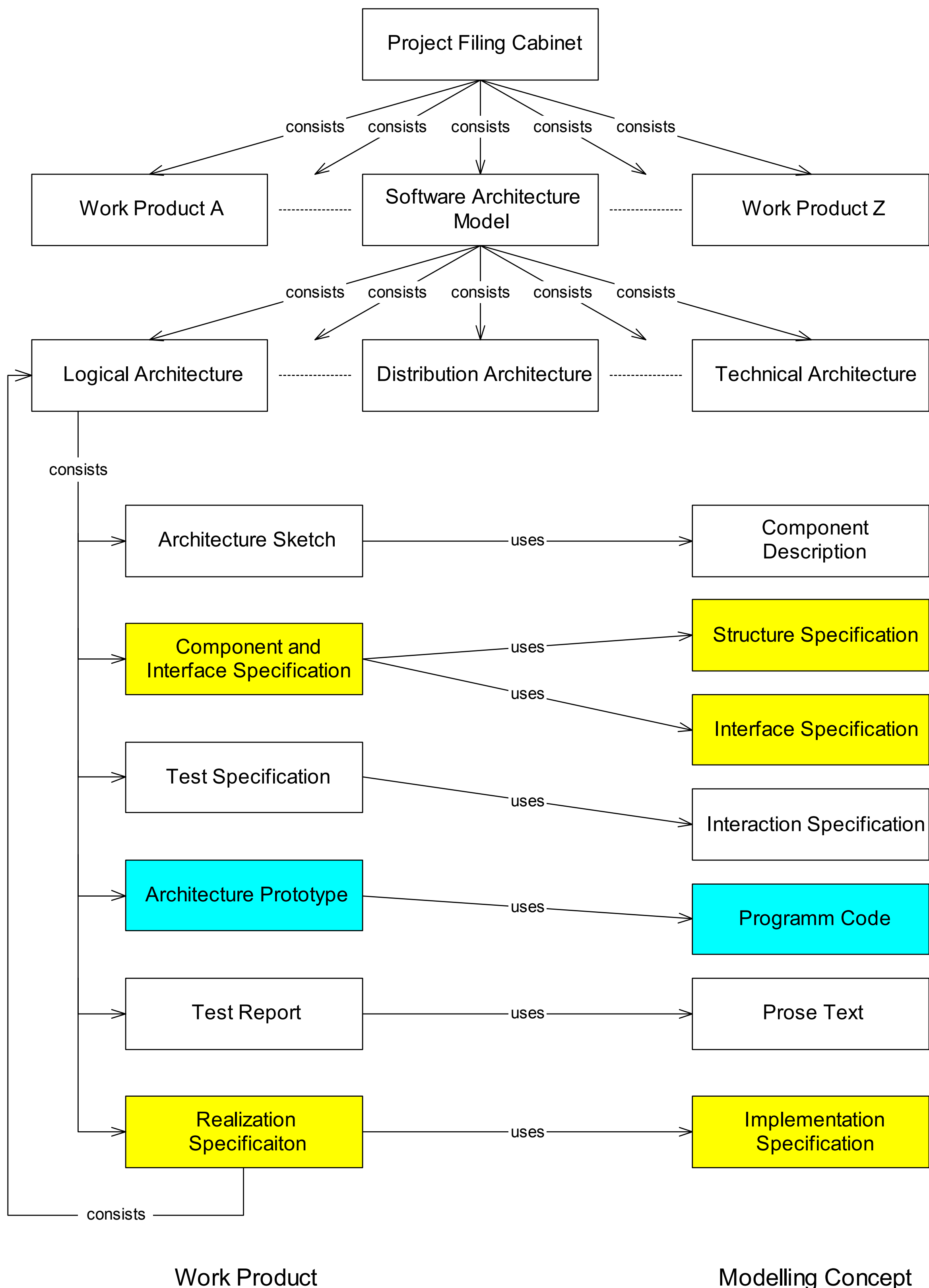

Abbildung 2.11: UML Klassendiagramm des Architekturmodells im Projektschrank

Andererseits sind die Elemente in dem Klassendiagramm selbst Instanzen von Klassen des Methodik-Frameworks, das der Metamodellebene in Abbildung 2.7 zugeordnet ist. So ist beispielsweise die Klasse *Logical Architecture* eine Instanz der Klasse *Work Product* aus Abbildung 2.8 und die Beziehung *consists* ist eine Instanz der Beziehung *relationship* aus Abbildung 2.8. Mit diesem Ansatz kann die Struktur und der Aufbau eines umfassenden Pro-

jektschrankes, der in Teilen bereits in Abbildung 2.11 dargestellt ist, auf Basis der Konzepte des Methodik-Frameworks sukzessive weiter ausgebaut und festgelegt werden.

In der vorliegenden Arbeit konzentrieren wir uns aber auf den Ausschnitt des Projektschrankes der die logische Architekturspezifikation enthält. Im Zentrum steht dabei eine möglichst präzise und vollständige Spezifikationstechnik für Komponenten- und Schnittstellenspezifikationen sowie zugehörige Realisierungsspezifikationen zu entwickeln. Aus diesen Spezifikationen können dann ablauffähige Architekturprototypen generiert werden (vgl. Abbildung 2.11). Deshalb verwenden wir in der restlichen Arbeit die Begriffe der Architekturspezifikation und des Architekturmodells als Synonyme für eine Menge von Komponenten- und Schnittstellenspezifikationen sowie zugehörige Realisierungsspezifikationen, außer der Kontext erfordert eine Unterscheidung.

2.4 Prozessmusterbasierter Ansatz und der evolutionäre Entwurf

Die Softwarearchitektur eines Systems und die zugehörigen Teilprodukte werden im Rahmen eines projektspezifischen Vorgehens- und Verfahrensplan entworfen. Verschiedene Techniken werden dabei eingesetzt, um diesen Plan effektiv umzusetzen. Im Rahmen dieser Arbeit verwenden wir Prozessmuster für die Beschreibung des evolutionären Architekturentwurfs.

Bevor wir das spezifische Prozessmuster des evolutionären Architekturentwurfs vorstellen, diskutieren wir im nächsten Kapitel zuerst den generellen Ansatz eines prozessmusterbasierten Vorgehensmodells. Im darauffolgenden Kapitel präsentieren wir die Schablone für die Beschreibung von Prozessmustern. Schließlich stellen wir dann das Prozessmuster des evolutionären Architekturentwurfs vor. Dabei zeigen wir insbesondere die Zusammenhänge zwischen den einzelnen Aktivitäten des evolutionären Architekturentwurfs und den benötigten, veränderten sowie erzeugten Teilprodukten des Architekturmodells auf.

2.4.1 Prozessmusterbasierte Ansatz für Vorgehensmodelle

Ein allumfassendes Vorgehensmodell ist entweder sehr restriktiv, so dass es mehr bei der Arbeit behindert, wie zum Beispiel das Wasserfallmodell [Royc70], oder es ist so generisch, dass der Anpassungsaufwand so groß ist, dass kein messbarer Nutzen zurückbleibt. Moderne Prozessmodelle, wie zum Beispiel das V-Modell ’97 [DW99] oder der Unified Software Development Process [JBR99, Kruc00] sind verstärkt von dieser generischen Bauart.

Da diese Modelle noch sehr neu sind, kann über Erfolg oder Misserfolg noch keine Aussage getroffen werden. Es ist aber bereits jetzt erkennbar, dass insbesondere bei kleineren Projekten die Verwendung eines Prozessmodells auf Grund der Komplexität meistens nicht rentabel ist [OHJ+99, GNR+00]. Da generische Prozessmodelle komplexer sind und vor dem Einsatz in einem Projekt noch angepasst werden müssen, kann sich dieser Trend auch auf mittlere Projekte ausdehnen. Speziell bei Componentware werden große Projekte in kleinere Unterprojekte aufgeteilt, die jeweils eine geringe Anzahl von Komponenten realisieren. Der Erfolg generischer Prozessmodelle in einem komponentenbasierten Entwicklungsumfeld erscheint daher eher unwahrscheinlich.

Die Vermutung liegt nahe, dass generische Prozessmodelle in der Theorie eine Hilfe darstellen können, allerdings für die Masse der Projekte nicht relevant sind. Da Prozessmodelle ein breites Feedback benötigen, ist es fraglich, ob diese Prozessmodelle ein großes Anwendungsspektrum finden werden. Eine endgültige Antwort auf diese Frage kann man aber vom augenblicklichen Standpunkt aus nicht geben.

Ungeachtet dessen wurden in den vergangenen Jahrzehnten einige Methoden und Vorgehensmodelle entwickelt und teilweise erfolgreich eingesetzt. Das eigentliche wertvolle methodische Wissen ist aber nicht diesen Vorgehensmodellen verborgen, sondern in den Köpfen der Mitarbeiter. Ausschlaggebend für eine erfolgreiche Projektdurchführung ist nicht die Methodik, die Technologie, die Beschreibungstechnik oder die Programmiersprache sondern die Mitarbeiter, deren Wissen und die Fähigkeit dieses Wissen einzubringen und umzusetzen. Alle anderen Bestandteile einer methodischen Softwareentwicklung sind nur Statisten, die unterstützend eingreifen und dafür Sorge tragen können, dass Produktivität und Erfolg nicht behindert wird (siehe auch [DL99]).

Wissen ist eine Mischung aus Erfahrungen, Wertvorstellungen, Kontextinformationen und Fachkenntnissen, die in ihrer Gesamtheit einen Rahmen zur Beurteilung und Eingliederung neuer Erfahrungen und Informationen bildet. In der Praxis ertrinken wir in Informationen, benötigen aber eigentlich Wissen. Rund ein Drittel der gesamten Arbeitszeit wird allein für das Suchen bereits vorhandenen Wissens verwendet [RKP00].

Es gibt nur wenige erfolgreiche Versuche wertvolles methodisches Wissen, das in den Köpfen der Mitarbeiter steckt, zu identifizieren, aufzubereiten, zu dokumentieren und zu kommunizieren. In der Vergangenheit haben sich hierbei sogenannte „pragmatische goldene Regeln“ [Broo75], Heuristiken [Your89, Ould95] und als neuste Erscheinung Muster [GHJV95, BMR+96, BRSV98a] bewährt.

Die Idee ist dabei immer ähnlich: Wissensartefakte werden in mehr oder weniger strukturierter Prosa mit eventuell zusätzlichen Diagrammen dokumentiert. Diese Wissensartefakte werden in einer möglichst prägnanten Form beschrieben, abstrahiert von einen konkreten Kontext. Zusätzlich wird die Anwendung des Wissens anhand möglichst stichhaltiger Beispiele demonstriert. Die Essenz des Wissens wird auf diese Weise verständlich dokumentiert. Die Anwendbarkeit wird plastisch illustriert. Und, es besteht die begründete Hoffnung, dass es kognitiv, für neue Anwendungsgebiete, adaptiert werden kann.

Mit dem Konzept der Prozessmuster, das in unserem Metamodell für Entwicklungsmethoden integriert ist (vgl. Kapitel 2.2.4, Abbildung 2.8), haben wir bereits eine Möglichkeit geschaffen methodisches Wissen festzuhalten. Im wesentlichen basiert das Modell der Prozessmuster auf einer sehr einfachen Annahme: Wir gehen davon aus, dass wir am Anfang eines Projektes relativ genau sagen können, was wir alles an Produkten bzw. Ergebnissen erwarten. Jedoch können wir noch nicht sagen, wie und in was für einer Reihenfolge wir diese Ergebnisse erzeugen werden.

Am Anfang des Projektes wird also ein konkretes Produktmodell definiert, der leere Projektschrank. Durch den Schrank wird aber nicht das „Wie“ der Produkterstellung festgelegt. Dies wird von den Prozessmustern übernommen. Diese Muster sind pragmatische Anleitungen für die Abwicklung einzelner Aktivitäten in einem Projekt. Dabei wird insbesondere die Reihenfolge der Aktivitäten bewusst offen gelassen.

Man kann sich Prozessmuster als mikro-codierte Handlungsanweisungen im Gehirn der Systementwickler vorstellen. Der Code kommt immer dann zur Ausführung, wenn man im realen Projekt bestimmte Situationen erkennt oder beobachtet. Dann weiß man, entsprechend des Prozessmusters, wie man darauf zu reagieren hat. Man kann jedoch nicht vorhersagen, welche Situationen in welcher Reihenfolge im Projekt auftreten werden.

Die Auswahl des anzuwendenden Musters wird anhand der konkreten Problemstellung durch das Team bzw. den Projektleiter vorgenommen. Dazu sollten in einer sogenannten Musterbibliothek eine Reihe von Prozessmustern bereit stehen. Getrieben durch eine konkrete Problemstellung und eingeschränkt durch den Projektkontext können so aus der Musterbibliothek passende Muster ausgewählt werden.

Die vorgeschlagene Lösung kann im Team diskutiert, eventuell an die aktuelle Situation angepasst und schließlich angewendet werden. Eine konkrete Problemstellung könnte zum Beispiel sein, dass das im Projekt sehr moderne, teilweise unausgereifte Technologien mit einem unerfahrenen Team eingesetzt werden sollen. Die Füllung des Projektschrankes, der Projektkontext, ist nicht weit fortgeschritten. Die Anforderungsspezifikation ist noch nicht abgeschlossen.

Bei diesem Kontext und bei der gegebenen Problemstellung könnte ein Prozessmuster, das sinnvoll anwendbar ist, folgendes Vorgehen vorschlagen: Unabhängig vom Analyseteam wird ein technischer Prototyp von einem neu zu bildenden Architekturteam erstellt. Damit lassen sich die technologischen Risiken minimieren, das technische Know-how kann frühzeitig aufgebaut werden. Die Arbeiten an der Anforderungsspezifikation und das zugehörige Analyseteam werden dadurch nicht beeinträchtigt.

Diese und ähnliche Prozessmuster finden sich in ausführlicherer Form in [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b]. Im Rahmen dieser Arbeit werden wir ein Prozessmuster für den evolutionären Architekturentwurfs vorstellen. Zuvor präsentieren wir im folgenden Kapitel die Beschreibungstechnik für Prozessmuster und Musterlandkarten.

2.4.2 Beschreibung von Prozessmustern und Musterlandkarten

Prozessmuster – wie alle Muster – sollten in einer entsprechenden einheitlichen Form präsentiert werden. Dies erleichtert die Verständlichkeit und man kann mit anderen besser darüber diskutieren. Eine gute Beschreibungsform hilft dabei die Essenz eines Muster möglichst schnell deutlich zu machen. Darüber hinaus bietet eine gute Beschreibung eines Muster aber auch alle Einzelheiten die notwendig sind, um ein Muster anzuwenden und die Konsequenzen sowie Vor- und Nachteile des Muster zu verstehen.

Außerdem erleichtert eine einheitliche Beschreibung der Muster die Vergleichbarkeit. Die Suche nach den verschiedenen, möglichen Lösungen für bestimmte Kontext- und Problemstellungen wird verbessert. In „The Timeless Way of Building“ definiert der Architekt Christopher Alexander den Begriff des Muster wie folgt:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
>
> As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system forces, wherever the context makes it relevant.

> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.
>
> (Zitat aus [Alex79], Seite 247)

In „Pattern-Oriented Software Architecture" von Frank Buschmann et al. werden Patterns wie folgt beschrieben:

> The discussion in the previous section leads us to adopt a three-part schema that underlies every pattern:
>
> Context: a situation giving rise to a problem.
>
> Problem: the recurring problem arising in that context.
>
> Solution: a proven resolution of the problem.
>
> (Zitat aus [BMR+96], Seite 8)

Gleichgültig, welches Verständnis von Mustern man zu Grunde legt, Muster bestehen stets aus dem Trio von Kontext, Problem und Lösung [GHJV95, BMR+96, BRSV98a]. Dabei liegt das Hauptaugenmerk auf einer sehr detaillierten Diskussion des Kontextes, der zu einer bestimmten Problemstellung führt, und den treibenden Kräften, die mit der gewählten Lösung optimal ausbalanciert werden sollen.

Diese rudimentäre Strukturierung eines Muster in Kontext, Problem und Lösung bietet einen guten Ausgangspunkt, eine detailliertere Beschreibung für Prozessmuster festzulegen. Tabelle 2.1 gibt das Beschreibungsformat für Prozessmuster an, das wir in dieser Arbeit verwenden werden. Dieses Beschreibungsformat ist angelehnt an die bekannten Musterbeschreibungen aus [GHJV95, BMR+96, BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b].

Abschnittsbezeichnung	Beschreibung des Inhaltes des Abschnitts
Name	Name des Musters
Intent	Eine kurze Zusammenfassung der Motivation, Ziele und Hintergründe des Musters. Das zentrale Problem und die methodischen Aspekte dieses Musters werden hier beschrieben.
Also Known As	Andere Namen unter denen dieses Muster bekannt ist.
Problem	Eine Beschreibung der spezifischen Entwicklungsaufgabe oder Problemstellung, die von diesem Muster adressiert wird. Dabei sollte ein anwendungsnahes Beispiel für die Motivation des Musters enthalten sein.
Work Products	Eine Liste der Produkte, die als Eingabe benötigt, neu erzeugt und verändert werden.
Context	Die Beschreibung des Kontextes in dem dieses Muster angewendet werden kann. Der Kontext ist in erster Linie die Eingabeprodukte, die das Muster benötigt, und die Ausgabeprodukte, die das Muster erzeugt bzw. manipuliert. Hierbei werden insbesondere

	auch Kontextbedingungen zwischen den Produkten festgehalten.
Solution	Der grundlegende Lösungsansatz des Musters ist in diesem Abschnitt dokumentiert. Besondere Aufmerksamkeit kommt dabei den methodischen Richtlinien und Rahmenbedingungen zu.
Structure	Eine textbasierte und grafische Repräsentation des Lösungsweges des Musters. Der Eingabe- als auch der Ausgabekontext und die Abfolge der einzelnen Aktivitäten werden festgehalten. Wir verwenden hierfür vorzugsweise UML Aktivitätsdiagramme.
Activities	Eine Liste und Beschreibung der Aktivitäten, die im Rahmen dieses Musters ausgeführt werden sollen. Die Aktivitäten können dabei bereits bestimmten Rollen zugewiesen werden.
Application Guidelines	Praktische Hinweise, Tipps und Tricks sowie wichtige Techniken, die bei der Anwendung des Musters hilfreich und sinnvoll sind. Unterstützende Verfahren, Techniken und Werkzeuge sollten ebenfalls erwähnt werden.
Application Examples	Bekannte Anwendungsbeispiele des Musters in konkreten Projekten und Entwicklungsszenarien. Diese Beispiele illustrieren die Anwendbarkeit und das Potential des Musters. Darüber hinaus können aber auch kontraproduktive Anwendungsszenarien sowie bekannte Fehler bei der Anwendung beschrieben werden.
Consequences	Die Vor- und Nachteile des Musters werden hier diskutiert. Dies erleichtert die Evaluierung der Anwendbarkeit des Musters.
Related Patterns	Eine Liste von Mustern die entweder alternativ, begleitend, im Vorfeld oder als nächste Schritte durchgeführt werden können oder mit einer ähnlichen Menge von Produkten arbeiten.

Tabelle 2.1: Vorlage für die Beschreibung von Prozessmustern

Mit der in Tabelle 2.1 dargestellten Struktur können einzelne Prozessmuster beschrieben werden. In einer Prozessmusterbibliothek sind eine Fülle derartiger Prozessmustern enthalten. Diese Prozessmuster unterscheiden sich in der Granularität und der Abstraktionsebene. Prozessmuster in einer Prozessmusterbibliothek müssen deshalb organisiert und adäquat verwaltet werden, beispielsweise mit Hilfe sogenannter Prozessmusterlandkarten.

Dabei sind die verschiedensten Strukturierungsmöglichkeiten vorstellbar, wie zum Beispiel ein thematischer Katalog oder eine Einteilung anhand der Abstraktionsebenen. Vor dem Hintergrund unseres Metamodells für Entwicklungsmethoden (vgl. Kapitel 2.2.4, Abbildung 2.8) erscheint eine Strukturierung sowohl anhand von Aktivitäten als auch anhand von Produkten besonders sinnvoll und hilfreich.

Eine Strukturierung anhand der Produkte kann man sich als einen gerichteten Graphen vorstellen, dessen Knoten jeweils Mengen von Produkten in einem bestimmten Kontext repräsentieren. Die gerichteten Kanten zwischen diesen Kontexten sind die Prozessmuster, die jeweils angeben wie man von einem bestimmten Eingabekontext zu einem gewünschten Ausgabekontext kommt. Wir bezeichnen diese Art von Musterlandkarten als Kontextlandkarten.

Produkte sind hierarchisch organisiert. Der Projektschrank besteht aus mehreren Schubladen. Eine Schublade ist das Architekturmodell, das wiederum aus Unterprodukten besteht. Auf jeder dieser Ebene können derartige Kontextlandkarten erarbeitet werden.

Die zweite Strukturierungsmöglichkeit, anhand der Aktivitäten, bezeichnen wir als Aktivitätslandkarten. Ein Prozessmuster beschreibt, wie eine Folge von Aktivitäten durchgeführt werden soll. Die Durchführung jeder dieser Aktivitäten kann wiederum durch weitere, alternative Prozessmuster beschrieben werden.

Aktivitätslandkarten werden ebenfalls durch gerichtete Graphen dargestellt. Die Knoten repräsentieren jetzt aber die Prozessmuster und die Kanten sind die Aktivitäten. Kanten verweisen jeweils von einem übergeordneten Prozessmuster auf andere Prozessmuster, die zur Durchführung einer bestimmten Aktivität des übergeordneten Prozessmusters angewendet werden können.

Wie Produkte, werden Aktivitäten auch hierarchisch strukturiert. Deshalb können wir, analog zu den Kontextlandkarten, auch Aktivitätslandkarten auf den verschiedenen Ebenen angeben. Eine weiterführende und tiefergreifende Diskussion des prozessmusterbasierten Ansatzes ist in [BRSV98a, BRSV98b, BRSV98c, GMP+01a, GMP+01b] zu finden.

2.4.3 Prozessmuster des evolutionären Architekturentwurfs

Beim evolutionären Architekturentwurf wird im Rahmen eines Evolutionsschrittes das Architekturmodell verändert. Am Ende dieses Schrittes stellt sich die zentrale Frage, ob das veränderte Modell konsistent ist und eine Implementierung existiert (Widerspruchsfreiheit), und, ob die Anforderungen besser erfüllt sind als das im vorhergehenden Modell der Fall war (Angemessenheit).

Im folgenden stellen wir das Prozessmuster des evolutionären Architekturentwurfs vor, entsprechend der Schablone für Prozessmuster aus Tabelle 2.1:

Name:

Evolutionärer Architekturentwurf

Intent:

Ziel des Softwarearchitekturentwurfs ist es, eine stabile und gut dokumentierte Softwarearchitektur zu entwerfen, die Flexibilität und Erweiterbarkeit des Systems unterstützt und dabei Performanz und Zuverlässigkeit garantiert.

Anforderungen, Technologien und Rahmenbedingungen ändern sich aber sehr schnell in den Projekten, meist noch während der Entwicklung des Systems. Eine stabile und erweiterbare Softwarearchitektur die mit einem evolutionären Entwurfsprozess entwickelt wird, minimiert die Risiken, die mit diesem schnellen Wandel verbundenen sind.

Darüber hinaus wird beim evolutionären Architekturentwurf durch die frühzeitige Rückkopplung über Prototyping eine beständige Qualitätssicherung erreicht. Im Vergleich zum explorativen und experimentellen Prototyping bleibt aber beim evolutionären Prototyping gewährleistet, dass ein stabiles, erweiterbares und konsistentes Architekturmodell entwickelt wird, auf dessen Basis das endgültige System erstellt werden kann.

Also Known As:

Evolutionary Software Architecture Modeling

Problem:

Neue Technologien, wie zum Beispiel das Internet oder UMTS, eröffnen unter Umständen völlig neue Geschäftsfelder. Die Erstellung eines Softwaresystems zur Unterstützung eines neuen Geschäftsfeldes unter Einbeziehung einer neuen Technologie birgt zwei zentrale Risiken:

Auf der einen Seite sind die fachlichen Anforderungen an das System nicht vollständig verstanden und durchdacht, da es noch kein vergleichbares System oder Erfahrungen in diesem Geschäftsfeld gibt. Auf der anderen Seite muss das System auf Basis einer neuen, noch unbekannten Technologie realisiert werden. Denn nur diese Technologie ermöglicht überhaupt den Zugang in den neuen Markt. Es stehen aber weder Referenzprojekte auf Basis dieser Technologie zur Verfügung, noch gibt es Erfahrungen mit dem Einsatz der Technologie.

Gerade bei neuen Märkten ist „Time to Market“ aber ein wichtiger Erfolgsfaktor. Deshalb ist es essentiell, möglichst frühzeitig ein produktives System am Markt zu platzieren. Somit steht man vor der Herausforderung unter starkem Zeitdruck ein System zu realisieren, das möglichst bald eine stabile Architektur bietet und somit die technische Machbarkeit zeigt. Gleichzeitig soll diese Architektur ein softwaretechnisches Gerüst bieten, in das man die notwendige Funktionalität einbetten, sukzessive weiter entwickeln und verändern kann.

Work Products:

Eingabe: Logische Architektur bestehend aus: Architekturskizze, Testspezifikation, Komponenten- und Schnittstellenspezifikationen (optional), Realisierungsspezifikationen (optional), Architekturprototyp (optional), Testbericht (optional)

Verändert: Komponenten- und Schnittstellenspezifikationen, Realisierungsspezifikationen, Architekturprototyp, Testbericht

Ausgabe: Logische Architektur bestehend aus: Architekturskizze, Testspezifikation, Komponenten- und Schnittstellenspezifikationen, Realisierungsspezifikationen, Architekturprototyp, Testbericht

Context:

Der evolutionäre Architekturentwurf wird iterativ im Projekt durchgeführt. Eine Iteration wird gestartet, wenn das Architekturmodell noch unvollständig ist oder sich die Anforderungen, Technologien, oder Rahmenbedingungen im Projekt geändert haben.

Die einzigen Anforderungen des evolutionären Architekturentwurfs an die benötigten Eingabeprodukte und den Projektkontext sind, dass eine erste grobe Architekturskizze existiert, die eine Liste von Komponenten und eine informelle Beschreibung dieser Komponenten beinhaltet. Außerdem muss eine Testspezifikation zur Verfügung stehen. Diese Teilprodukte des Architekturmodells sind meist Ergebnisse eines vorhergehenden Requirementsengineering- und Analyseprozesses.

Solution:

Es gibt einen nicht vollständig verhinderbaren Wandel der Anforderungen eines Softwaresystems, den Technologien auf deren Basis das Softwaresystem erstellt wird, sowie der Rahmenbedingungen des Projektes. Der evolutionäre Architekturentwurf versucht den damit verbundenen Risiken vorzubeugen, in dem frühzeitig eine stabile Softwarearchitektur erarbeitet wird, die es ermöglicht die Auswirkungen der Veränderung von Anforderungen, Technologien und Rahmenbedingungen abzufedern.

Die Softwarearchitektur wird in mehreren Iterationen entworfen. Am Ende jeder Iteration steht ein konsistentes und in sich vollständiges Architekturmodell zur Verfügung. Dieses Architekturmodell wird einer entsprechenden Qualitätssicherung unterzogen. Das Architekturmodell erlaubt eine möglichst einfache Realisierung, eventuell sogar eine automatische Generierung eines Prototypen. Der Prototyp ermöglicht am Ende jeder Iteration eine Überprüfung und Rückkopplung bezüglich der Erfüllung der Anforderungen und der technischen Realisierbarkeit. Das bestehende Architekturmodell wird in der nächsten Iteration anhand der dabei erzielten Erkenntnisse weiter entwickelt und verbessert.

Damit die iterative Evolution des Architekturmodells in einem geordnetem und zielgerichtetem Rahmen umgesetzt wird, ist es besonders hilfreich, wenn die Abhängigkeiten zwischen den einzelnen Komponenten explizit spezifiziert werden. Wird in einem Evolutionsschritt eine Komponente weiter entwickelt, so kann man die Auswirkungen auf die anderen Elemente des Architekturmodells wesentlich besser einschätzen. Probleme und mögliche Fehlerquellen, die sonst erst bei der Integration des Gesamtsystems auftreten, werden so frühzeitig bereits auf der Modellebene erkannt und können behoben werden.

Structure:

Abbildung 2.12 zeigt das Vorgehen des evolutionären Architekturentwurfs als UML Aktivitätsdiagramm. Eingabe des evolutionären Architekturentwurfs ist ein eventuell unvollständiges Modell einer logischen Architektur, bestehend aus einer Architekturskizze und einer Testspezifikation. Der evolutionäre Architekturentwurf wird insbesondere auch bei bereits vorhandenen Architekturmodellen angewendet. Deshalb können optional auch die anderen Teile der logischen Architektur bereits vorhanden sein: Komponenten- und Schnittstellenspezifikationen, Realisierungsspezifikationen, ein Architekturprototyp und ein Testbericht.

In der ersten Aktivität, der Spezifikation der Softwarearchitektur (*Software Architecture Specification*), erzeugt oder verändert der Softwarearchitekt die Architekturspezifikation, die aus Komponenten- und Schnittstellenspezifikationen sowie Realisierungsspezifikationen für hierarchische Komponenten besteht.

Im nächsten Schritt muss der Softwarearchitekt die Änderungen überprüfen und validieren (*Validate Evolutionary Model Changes*). Vorraussetzung hierfür ist, dass die Spezifikationstechnik es ermöglicht, Komponenten und Abhängigkeiten zwischen Komponenten präzise und explizit zu beschreiben. Die Auswirkungen der Veränderung einzelner Komponenten auf das Gesamtmodell können dann dem Architekten transparent dargestellt werden. Die Integrationsproblematik kann so bereits auf der Architekturmodellebene angegangen werden. Ist das Architekturmodell nicht mehr konsistent, so werden dem Architekten detailliertere Informationen über die inkonsistenten Teile der Spezifikation zur Verfügung gestellt. Dies ermöglicht eine zielgerichtete Verbesserung der Architekturspezifikation.

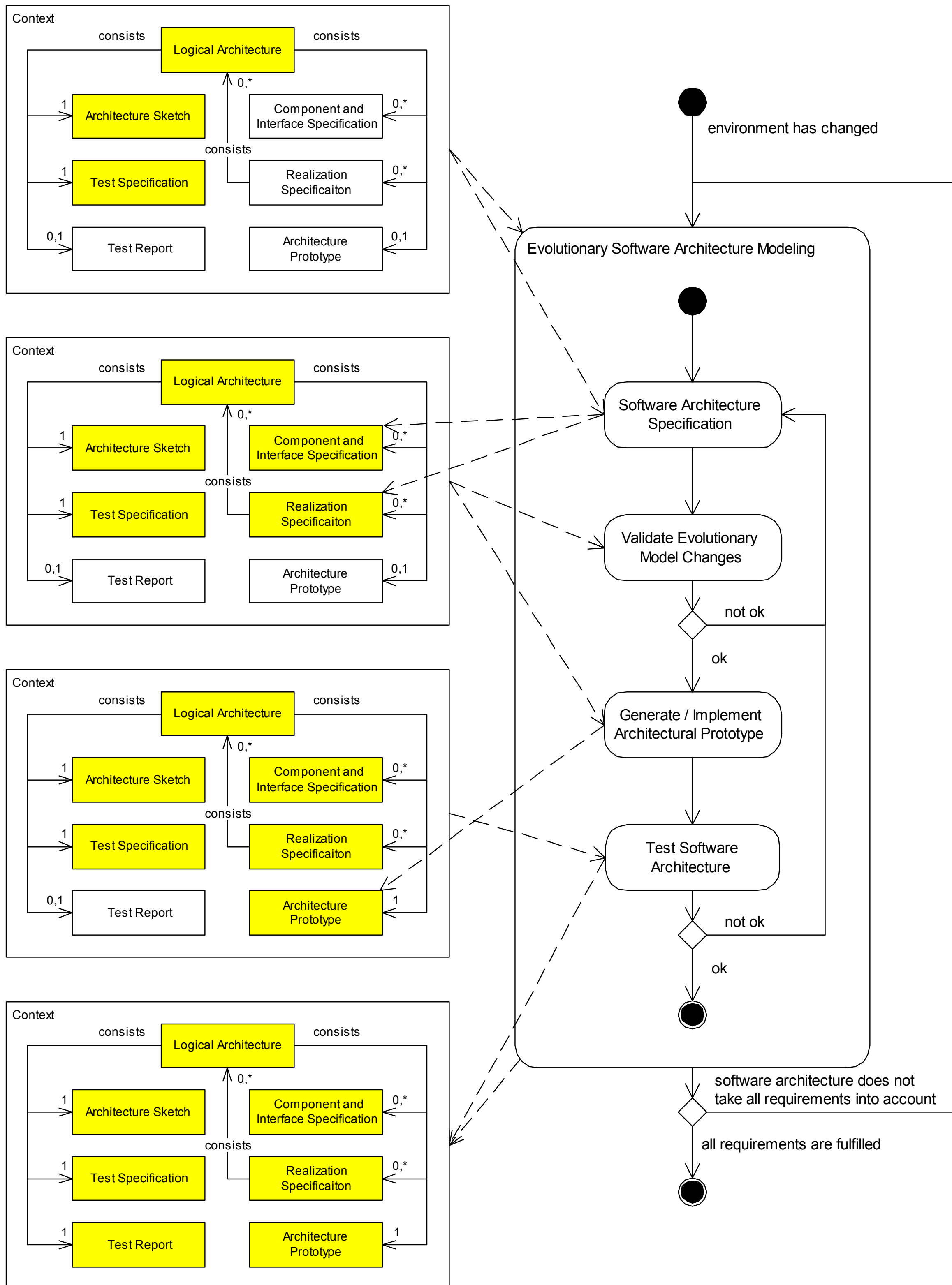

Abbildung 2.12: Vorgehen beim evolutionären Architekturentwurf

Ist das Architekturmodell konsistent, so kann ein Architekturprototyp generiert bzw. implementiert werden (*Generate / Implement Architectural Prototype*). Nun kann die Qualität des erzeugten Architekturmodells überprüft werden. Anhand der Testspezifikation wird der Architekturprototyp getestet (*Test Software Architecture*). Das Ergebnis der Tests wird in einem Testbericht festgehalten. Wurde der Test nicht erfolgreich absolviert, so wird die Architektur-

spezifikation entsprechend verbessert. Besteht der Architekturprototyp den Test, so ist diese Iteration der Architekturmodellierung beendet.

Jetzt kann die Realisierung der atomaren Komponenten des entworfenen Architekturmodells erfolgen, beispielsweise durch eine objektorientierte Modellierung und Implementierung oder durch einen weiteren evolutionären Architekturentwurf für jede Komponente. Unabhängig davon kann es aber sein, dass noch nicht alle Anforderungen im Architekturmodell berücksichtigt sind oder Anforderungen, Technologien, sowie Randbedingungen sich im Projekt ändern. Dann wird die nächste Iteration des evolutionären Architekturentwurfs gestartet.

Activities:

Software Architecture Specification: In der Architekturskizze sind bereits eine Liste von Komponenten und deren Verantwortlichkeiten beschrieben. Ziel dieser Aktivität ist es, eine möglichst präzise Spezifikation dieser Komponenten im Rahmen einer konsistenten und vollständigen Architekturspezifikation zu erarbeiten. Diese besteht aus einer Reihe von Komponenten- und Schnittstellenspezifikationen. Für Komponenten, die selbst wieder aus anderen Komponenten zusammengesetzt sind, ist eine Realisierungsspezifikation zu erstellen, die aus einer weiteren vollständigen Architekturspezifikation besteht.

Validate Evolutionary Model Changes: Dem Softwarearchitekten werden die Auswirkungen der Modellevolution transparent dargestellt. Insbesondere wird dabei bereits auf der Modellebene die Integration der einzelnen Komponentenspezifikationen zu einem Gesamtsystem vorgenommen. Ist das neue Modell inkonsistent, so werden dem Architekten die Komponenten präsentiert, die zu dem inkonsistenten Modell geführt haben. So kann er zielgerichteter das Architekturmodell modifizieren, so dass es konsistent wird und spätere Integrationsprobleme bereits im Vorfeld vermieden werden.

Generate / Implement Architectural Prototype: In dieser Aktivität wird aus der Spezifikation des Architekturmodells ein Prototyp erstellt. Wird für die Spezifikation eine weitgehend formalisierte Beschreibungstechnik verwendet, so kann ein ablauffähiger Prototyp generiert werden. Ist dies nicht der Fall, muss er von Hand implementiert werden.

Test Software Architecture: Im Rahmen dieser Aktivität wird das Architekturmodell, das zuvor erarbeitet wurde, möglichst von unabhängiger Seite getestet. Anhand der exemplarischen Abläufe, die in der Testspezifikation beschrieben sind, wird die Funktionsfähigkeit des Modells überprüft. Außerdem muss festgehalten werden, welchen Anforderungen das Architekturmodell genügt und welchen noch nicht.

Application Guidelines:

Eine Iteration des evolutionären Architekturentwurfs wird dann gestartet, wenn noch nicht alle Anforderungen erfüllt sind oder sich Anforderungen, Technologien oder Rahmenbedingungen verändert haben. Der Softwarearchitekt steht dann zuerst vor der Aufgabe eine neue, verbesserte Architekturspezifikation zu entwerfen, die den Anforderungen besser gerecht wird als das ursprüngliche Modell.

Ein sinnvolles generelles methodisches Verfahren, das den Architekten beim Entwurf des verbesserten Architekturmodells anleitet, kann nicht existieren, da es den spezifischen Bedürfnissen der konkreten Projektsituation nicht gerecht werden würde. Es stehen aber eine

Fülle von einzelnen Vorgehensweisen zur Verfügung, die jeweils abhängig von dem tatsächlichen Projektkontext sinnvoll angewendet werden können oder nicht.

Jedes dieser Verfahren könnte als eigenständiges Prozessmuster beschrieben werden, um die erste Aktivität des evolutionären Architekturentwurfs, die Spezifikation der verbesserten Softwarearchitektur, umzusetzen. Hierzu zählen die in der Literatur unter dem Schlagwort „Refactoring“ bekannten Techniken zur Evolution von objektorientierten Modellen (siehe [Opdy92, Fowl99]), die Anwendung von spezifischen Entwurfstechniken und Entwurfsmustern für die Evolution von Softwarearchitekturen (siehe [FO95, GHJV95, BMR+96]) und eine Reihe von formalen Entwicklungskalkülen (siehe [BDD+92, Rump96, Krüg00, BS01]). Letztere sind im Rahmen des evolutionären Architekturentwurfs nur bedingt anwendbar, da diesen Konzepten ein starker Verfeinerungsbegriff zu Grunde liegt, der mit dem Anspruch der evolutionären Modellierung nicht immer harmoniert (vgl. auch Kapitel 3.2).

Speziell hinter den ersten zwei Verfahren – Refactoring und der musterbasierten Entwurf – verbirgt sich eine gemeinsame Strategie, die generell angewendet werden kann: Zuerst wird eine neue, verbesserte Strukturierung des Systems in Komponenten erarbeitet. Im nächsten Schritt werden den Komponenten in dieser Struktur die entsprechenden Datenanteile zugeordnet. Im letzten Schritt werden dann die dynamischen Interaktionen zwischen den Komponenten festgelegt.

Soll beispielsweise im Rahmen eines evolutionären Entwicklungsschrittes ein prozeduraler Entwurf in einen objektorientierten Entwurf verändert werden, so werden zuerst aus den bestehenden Datenentitäten und Funktionen des prozeduralen Entwurfs die neuen Objekte identifiziert. Im nächsten Schritt werden diesen Objekten die einzelnen Attribute von den ursprünglichen Datenentitäten zugewiesen. Und schließlich werden dann die ursprünglichen Funktionen den neuen Objekten zugeordnet. So wird entsprechend dem voranstehenden Schema der evolutionäre Entwurf einer verbesserten Architekturspezifikation methodisch schrittweise entwickelt (siehe auch [Fowl99]).

Speziell in den anfänglichen Iterationen des evolutionären Architekturentwurfs sollte nicht zuviel Aufwand in eine möglichst umfangreiche Spezifikation investiert werden. Es kann hilfreich sein, die Abhängigkeiten zwischen den einzelnen Komponenten zu Beginn noch nicht vollständig und explizit zu dokumentieren. Für die ersten Architekturmodelle genügt es die Beschreibungstechniken aus Kapitel 6 zu verwenden. Diese erlauben bereits eine Generierung eines vollständigen Architekturprototypen, die Abhängigkeiten zwischen den einzelnen Komponenten werden aber noch nicht explizit spezifiziert.

Hat sich das Projektumfeld stabilisiert, werden die zeitlichen Abstände zwischen den Iterationen länger, dann sollte der Fokus auch auf der Spezifikation der Abhängigkeiten zwischen den Komponenten in einem Architekturmodell liegen. Hierzu können dann die erweiterten Beschreibungstechniken aus Kapitel 7 verwendet werden.

Aus dem Architekturmodell sollte möglichst effizient ein Prototyp erstellt werden können. Basiert das Modell auf einer präzisen Spezifikationstechnik, die auf wohl fundierte Notationen zurückgreift, so steht einer automatisierten Generierung eines ablauffähigen Prototypen nichts im Wege. Es können Werkzeuge erstellt werden, die einen Prototyp vollständig generieren können (vgl. Kapitel 8).

Eine Iterationen des evolutionären Architekturentwurfs sollte nicht zu viel Zeit in Anspruch nehmen. Ein frühes Feedback ist meist wichtiger, als die Umsetzung möglichst vieler Anforderungen.

Application Examples:

Die vorliegende Arbeit

Consequences:

Anforderungen, Technologien und Rahmenbedingungen verändern sich heutzutage sehr schnell in Projekten, meist noch während der Entwicklung eines Systems. Die Vorteile des evolutionären Architekturentwurfs liegen in einer frühen Minimierung des damit verbundenen Risikos. Der evolutionäre Architekturentwurf unterstützt so die Entwicklung einer stabilen und erweiterbaren Softwarearchitektur, die einen langfristigen softwaretechnischen Rahmen bietet für die sukzessive Entwicklung der eigentlichen Funktionalität des Systems.

Die mit dem evolutionären Architekturentwurf verbundene Spezifikationstechnik erlaubt eine präzise und vollständige Spezifikation der Softwarearchitektur. Darüber hinaus werden insbesondere die Abhängigkeiten zwischen den Komponenten der Softwarearchitektur explizit spezifiziert. Die Probleme, die typischerweise bei der Integration der Softwarekomponenten auftauchen, können hierdurch bereits im Softwarearchitekturmodell erkannt und behoben werden.

Die mit dem evolutionären Architekturentwurf verbundene präzise Architekturspezifikation ermöglicht eine einfache und zeitnahe Umsetzung der Spezifikation in einem Architekturprototypen. Durch dieses frühe Prototyping ist eine weitreichende und frühzeitige Rückkopplung für den Architekten selbst aber auch zusammen mit den Anwender möglich. So wird die Qualität der Softwarearchitektur und des Softwaresystems nachhaltig verbessert.

Die wesentlichen Nachteile sind ein erhöhter Aufwand bei der Spezifikation und beim Testen der Softwarearchitektur. Für Kunden und Anwender ist ebenfalls ein erhöhter Aufwand zu erwarten, da sie häufiger mit Prototypen konfrontiert werden.

Mit jedem neuen Prototyp besteht dabei die Gefahr, dass Anforderungen die bereits als sicher galten, wieder in Frage gestellt werden. Dies kann zu unnötigen Iterationen führen. Deshalb sollte die Diskussion mit dem Kunden anhand von Prototypen gut vorbereitet und wohl überlegt sein. Nicht jeder Prototyp muss dem Kunden vorgeführt werden, insbesondere wenn er nur dazu dient, technische Konzepte und deren Realisierbarkeit zu zeigen.

Related Patterns:

Siehe [Ambl98, Ambl99, BRSV98a, BRSV98b, BRSV98c, Fowl99, GMP+01a, GMP+01b]

2.5 Zusammenfassung

Die Konzeption, Entwicklung, Realisierung und der Betrieb von Softwaresystemen ist eine herausfordernde Aufgabe. Das Software-Engineering versucht dem Softwareingenieur in allen Teilgebieten – Vorgehensmodell, Beschreibungstechniken und Systemmodell, Technologie und Architektur, Managementpraktiken und Werkzeugunterstützung – Bausteine für eine erfolgreiche Softwareentwicklung zur Verfügung zu stellen.

Eine allumfassende Methode für die Softwareentwicklung existiert jedoch noch nicht. Die bestehenden Fragmente einer methodischen Softwareentwicklung werden ständig verbessert und neue Bestandteile kommen laufend hinzu. Deshalb haben wir in diesem Kapitel ein Rahmenwerk für die methodische Softwareentwicklung erarbeitet und vorgestellt.

Dieses Methodik-Framework besteht aus einem integrierten Produktmodell und einer Reihe von Prozessmustern. Durch das Produktmodell werden die Ergebnisse des Entwicklungsprozesses im Vorfeld sehr präzise definiert. Die zeitliche Reihenfolge, in der die Ergebnisse erstellt werden, und die einzelnen Arbeitsschritte zur Erstellung der Ergebnisse sind aber nicht fest vorgegeben. Prozessmuster geben denkbare Wege und Teillösungen an, um methodisch zu bestimmten Produkten und Ergebnissen zu gelangen.

Zentraler Bestandteil des Produktmodells sind die einzelnen Entwicklungsdokumente, die im Rahmen der Softwareentwicklung erstellt werden. Entwicklungsdokumente werden zu Modellen zusammengefasst, die das System auf einer bestimmten Abstraktionsebene beschreiben. Zwischen den verschiedenen Modellen, die im Laufe der Entwicklung eines Systems entstehen, existieren charakteristische Beziehungen. Beispielsweise sollte das Designmodell eine Verfeinerung des Analysemodells sein. Dagegen ist ein durch geänderte Anforderungen notwendiges, neues Architekturmodell meist keine Verfeinerung des ursprünglichen Modells, sondern eine Weiterentwicklung.

Im Zentrum der vorliegenden Dissertation steht die evolutionäre Weiterentwicklung des Architekturmodells. Ein Architekturmodell besteht dabei aus verschiedenen Sichten auf die Softwarearchitektur des Systems. Im Rahmen dieser Arbeit konzentrieren wir uns auf die logische Architektur. Diese beschreibt möglichst präzise die einzelnen Komponenten des Systems, deren Schnittstellen und die Beziehungen zwischen diesen Schnittstellen.

In einem Prozessmuster haben wir die einzelnen Teilschritte des evolutionären Architekturentwurfs festgehalten. Der Entwurf findet dabei in mehreren Iterationen statt, an deren Ende jeweils ein evolutionär weiter entwickeltes Architekturmodell steht. Startpunkte einer Iteration sind ein noch unvollständiges Architekturmodell oder Veränderungen der Anforderungen, Technologien oder Rahmenbedingungen des Projektes.

In einem ersten Schritt werden dann die Komponenten und Schnittstellen einer verbesserten Architektur vollständig und präzise spezifiziert. Insbesondere sollten die Abhängigkeiten zwischen den einzelnen Komponenten explizit dokumentiert werden. So kann die Integration der einzelnen Komponentenspezifikationen zu einer konsistenten Architekturspezifikation bereits auf der Modellebene durchgeführt werden.

Im nächsten Schritt wird aus der integrierten Architekturspezifikation ein Prototyp generiert oder implementiert. Der Prototyp kann dann anhand der Testspezifikation validiert werden. Erfüllt er die Anforderungen, so ist diese Iteration des evolutionären Architekturentwurfs erfolgreich abgeschlossen.

Eine effektive Umsetzung dieses Vorgehens basiert auf formal fundierten Beschreibungstechniken. Mit Hilfe dieser Beschreibungstechniken kann eine Softwarearchitektur so spezifiziert werden, dass ablauffähige Prototypen daraus generiert werden können. Darüber hinaus können die Abhängigkeiten zwischen den einzelnen Teilen der Spezifikation explizit beschrieben werden, so dass die Auswirkungen des evolutionären Entwurfs und die damit verbundenen Integrationsfragestellungen frühzeitig erkannt und angegangen werden können. In den folgenden Kapiteln werden wir die grundlegende formale Konzeption vorstellen und die entsprechenden Beschreibungstechniken erarbeiten.

3 Evolutionärer Entwurf komponentenbasierter Systeme

Ziel dieser Arbeit ist es, eine Methodik für den evolutionären Architekturentwurf komponentenbasierter Systeme bereit zu stellen. Die zentralen Elemente dieser Methodik sind die evolutionäre Modellierung von Softwarearchitekturen und die Generierung von Architekturprototypen aus Architekturmodellen. Nach jedem Evolutionsschritt stellt sich die Frage nach der Widerspruchsfreiheit und der Angemessenheit des neuen, weiter entwickelten Architekturmodells (vgl. auch Kapitel 1 und 2).

Vor dem Hintergrund dieser Fragestellungen weisen die gängigen formalen Methoden erhebliche Defizite auf. Im Zentrum dieser Ansätze stehen semantisch fundierte Beschreibungstechniken, die eine weitgehend vollständige Generierung der Implementierung aus konsistenten Beschreibungen ermöglichen. Sind die Beschreibungen jedoch inkonsistent, so sind spezifischere Aussagen in der Regel nicht möglich. Insbesondere bei komponentenbasierten Systemen, die evolutionär entworfen werden, sind aber weitergehende Fragestellungen von Bedeutung, wie zum Beispiel welche Veränderungen an einzelnen Komponentenbeschreibungen zu einer inkonsistenten Systemspezifikation geführt haben.

In diesem Kapitel werden die grundlegenden Konzepte einer formalen Methodik erarbeitet, die den besonderen Anforderungen des evolutionären Architekturenwurfs komponentenbasierter Systeme gerecht wird.

Kapitel 3.1 stellt die Grundzüge gängiger formaler Methoden vor, die systemmodellbasierte formale Semantik von Dokumentenmengen. Eine formale Semantik definiert dabei die Zusammenhänge zwischen Dokumentenmengen und der Menge von Systemen, die betrachtet werden sollen. Die evolutionäre Veränderung von Dokumentenmengen wird in Kapitel 3.2 eingeführt. In der Konsequenz führt dies zu dem in der Literatur bekannten „Abstraction Refinement Model“. In Kapitel 3.3 diskutieren wir diesen Ansatz und die damit verbundenen Schwächen im Hinblick auf die spezifischen Anforderungen des evolutionären Architekturentwurfs.

In Kapitel 3.4 wird eine weiter entwickelte, neuartige formale Semantik für Architekturmodelle präsentiert, die prädikatenbasierte formale Semantik von Spezifikationen. Diese ist speziell an die Bedürfnisse der evolutionären Modellierung komponentenbasierter Systeme angepasst. Das darauffolgende Kapitel 3.5 zeigt, dass die Fragestellungen der Widerspruchsfreiheit und der Angemessenheit mit diesem Ansatz besser beantwortet werden können als mit dem Abstraction Refinement Model, das auf einer systemmodellbasierten formalen Semantik beruht.

Abschließend präzisieren wir in Kapitel 3.6 die unterschiedlichen Varianten der Modellevolution. Damit lassen sich Querbezüge zwischen der systemmodellbasierten und der prädikatenbasierten formalen Semantik aufzeigen. Somit wird eine nahtlose Synergie dieser Ansätze erreicht.

3.1 Systemmodellbasierte formale Semantik

Im Gegensatz zu Hardware ist Software immateriell und abstrakt. Deshalb ist die Modellbildung ein zentrales Element der Softwareentwicklung. Selbst die Programmierung ist letztlich nur das Erstellen eines Modells des zu realisierenden Systems. Offensichtlich ist dieses Modell bereits sehr konkret und operationell. Es kann entweder direkt von einem Interpreter ausgeführt, oder zuerst durch einen Compiler übersetzt und dann ausgeführt werden.

Ziel einer methodischen Softwareentwicklung ist es, ein effektives sowie möglichst effizientes Verfahren für die Modellbildung festzulegen. Bei der formalen methodischen Softwareentwicklung beruht dieses Verfahren auf Konzepten der Mathematik.

Viele formale Ansätze einer methodischen Softwareentwicklung, wie zum Beispiel FOCUS [BDD+92, BS01], Temporal Logic [Lamp89] oder Architectural Description Languages (ADL) [SG96], basieren auf einer sehr ähnlichen Grundidee: Formal fundierte Beschreibungstechniken garantieren die syntaktische und semantische Konsistenz der Modelle, die im Laufe der Softwareentwicklung erstellt werden. Mathematische Kalküle ermöglichen es, die Modelle syntaktisch zu verändern und zu transformieren, ohne dabei die semantische Konsistenz zu beeinträchtigen.

Diese formalen Ansätze basieren auf zwei grundlegenden Konzepten:

- Trennung von Syntax und Semantik sowie
- präzise Formulierung der Zusammenhänge zwischen Syntax und Semantik.

Ein syntaktisches Modell legt die Beschreibungstechniken fest, die dem Softwareentwickler zur Verfügung gestellt werden. Das Systemmodell liefert die Interpretationsplattform für das syntaktische Modell. Eine systemmodellbasierte formale Semantik definiert die Zusammenhänge zwischen dem syntaktischen Modell und dem Systemmodell. Die folgenden Definitionen geben diese Zusammenhänge formal wieder:

Definition 3.1: Systemmodell

Ein Systemmodell charakterisiert die Menge aller gültigen Systeme SYSTEM. *Eine Menge von Systemen* $\mathsf{System} \subseteq \mathsf{SYSTEM}$ *bezeichnen wir auch als Implementierungen eines Softwaresystems.*

Definition 3.2: Syntaktisches Modell

Ein syntaktisches Modell charakterisiert die Menge aller gültigen Entwicklungsdokumente $\mathsf{DOCUMENT}$. *Eine Menge von Entwicklungsdokumenten* $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ *bezeichnen wir auch als Modell eines Softwaresystems oder als Spezifikation eines Softwaresystems.*

Definition 3.3: Systemmodellbasierte formale Semantik von Dokumentenmengen

Eine systemmodellbasierte formale Semantik einer Dokumentenmenge $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ *ist durch die folgende Funktion gegeben:*

$$\mathsf{sem} : \mathrm{P}(\mathsf{DOCUMENT}) \rightarrow \mathrm{P}(\mathsf{SYSTEM})$$

Die Semantik der leeren Dokumentenmenge ist wie folgt definiert:

$$\mathsf{sem}(\varnothing) =_{\mathsf{def}} \mathrm{P}(\mathsf{SYSTEM})$$

$\mathrm{P}(\mathsf{A})$ *steht für die Potenzmenge, die Menge aller Teilmengen der Menge* A.

Die formale Methodik FOCUS [BDD+92, BS01] und viele der Arbeiten rund um FOCUS, wie zum Beispiel [Berg97], [Krüg00] und [Rump96], beruhen auf diesen Grundlagen. Im Rahmen dieser Arbeiten werden die komplexen Beziehungen zwischen einzelnen Dokumenten sowie Dokumentenmengen, die während der Systemerstellung entstehen, präzisiert und formal fundiert.

Informelle Aussagen über Modelle eines Systems, die durch Dokumentenmengen beschrieben werden, können durch exakte mathematische Aussagen über das Systemmodell formalisiert werden. Beispielsweise wird die Konsistenz einer Dokumentenmenge $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ wie folgt festgelegt:

$$\mathsf{consistent}(\mathsf{Document}) \Leftrightarrow_{\mathsf{def}} \mathsf{sem}(\mathsf{Document}) \neq \emptyset$$

Die Verfeinerungsbeziehung zwischen dem Analysemodell $\mathsf{Document}_{\mathsf{analysis}} \subseteq \mathsf{DOCUMENT}$ und dem Designmodell $\mathsf{Document}_{\mathsf{design}} \subseteq \mathsf{DOCUMENT}$ ist charakterisiert durch:

$$\mathsf{sem}\left(\mathsf{Document}_{\mathsf{analysis}}\right) \supseteq \mathsf{sem}\left(\mathsf{Document}_{\mathsf{design}}\right)$$

Darüber hinaus ist die systemmodellbasierte formale Semantik von Dokumentenmengen gleichzeitig die Spezifikation der semantikerhaltenden Übersetzung der Dokumente, das heißt die Spezifikation der korrekten Generierung von Programmcode aus einem Modell eines Systems (vgl. [Berg97] oder [Rump96]).

3.2 Evolution von Dokumentenmengen und Spezifikationen

Die Fähigkeit, Softwaresysteme in einer kontrollierten Art und Weise weiter zu entwickeln, ist ein zentrales Charakteristikum der evolutionären methodischen Softwareentwicklung. Basiert der evolutionäre Ansatz auf formalen Konzepten, so muss die Evolution von Spezifikationen und Modellen mathematisch beschrieben werden können.

Unter einem Evolutionsschritt verstehen wir eine Veränderung der Entwicklungsdokumente des Softwaresystems. Ein Softwareentwickler, der beispielsweise am Anfang seines Arbeitstages eine Menge von Programmdateien aus dem gemeinsamen Projektarchiv herausnimmt, diese tagsüber verändert, und abends dann wieder in das gemeinsame Archiv zurückspeichert, hat in diesem Sinne einen Evolutionsschritt durchgeführt. Formal kann die Evolution von Dokumentenmengen wie folgt charakterisiert werden:

Definition 3.4: Evolution von Dokumentenmengen

Die Funktion evolve *modelliert die Veränderung von Dokumenten und ist wie folgt definiert:*

$$\mathsf{evolve} : \mathrm{P}(\mathsf{DOCUMENT}) \to \mathrm{P}(\mathsf{DOCUMENT})$$

$\mathsf{evolve}(\mathsf{Document})$ *bezeichnet eine Menge von veränderten Entwicklungsdokumenten, die durch einen Evolutionsschritt aus der ursprünglichen Dokumentenmenge* $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ *entstanden sind.*

Mit diesem Begriff eines Evolutionsschrittes lassen sich Verfeinerungsbeziehungen zwischen einem Modell $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ und dem evolutionär weiter entwickelten Modell $\mathsf{evolve}(\mathsf{Document})$ wiederum sehr einfach charakterisieren:

$$\mathsf{sem}(\mathsf{Document}) \supseteq \mathsf{sem}\left(\mathsf{evolve}(\mathsf{Document})\right)$$

Dabei wäre es natürlich wünschenswert, dass zum Beispiel ein Implementierungsmodell eines Systems eine Verfeinerung des Designmodells ist und das Designmodell eine Verfeinerung

des Analysemodells. In der industriellen Praxis ist dies aber meist nicht durchzuhalten. Angenommen, neue Anforderungen an das System ergeben sich während der Implementierung, so werden diese im Implementierungsmodell realisiert. Die anderen Modelle, wie beispielsweise das Designmodell oder das Analysemodell, werden meist aus Zeitgründen nicht nachbearbeitet (vgl. auch Kapitel 2.3). David L. Parnas bezeichnet dieses Phänomen als „software aging" [Parn94].

In [Parn94] legt er dar, dass während der Entwicklung eines Systems Verfeinerungsbeziehungen zwischen Modellen nicht zielgerecht sind. Der Zusatzaufwand für die laufende Sicherung und Erhaltung der Verfeinerungsbeziehung zwischen den Modellen würde den Entwicklungsfortschritt wesentlich beeinträchtigen. Für die abschließende Dokumentation des Systems sind mehrere Modelle auf unterschiedlichen Abstraktionsebenen, die entsprechenden Verfeinerungsbeziehungen genügen, sehr wertvoll. In der Praxis ist selbst dies aus Zeit- und Kostengründen nicht immer umsetzbar.

Deshalb sind, unter methodischen sowie praktischen Gesichtspunkten betrachtet, Verfeinerungsbeziehungen zwischen den verschiedenen Entwicklungs- bzw. Evolutionsstufen von Modellen nicht relevant. Im Rahmen des evolutionären Entwurfs von Architekturmodellen wäre die zwingende Forderung nach Verfeinerungsbeziehungen sogar kontraproduktiv. Ein klassischer Verfeinerungsbegriff würde dem Softwarearchitekten nur erlauben, zusätzliche Eigenschaften hinzuzufügen, die ursprünglichen semantischen Eigenschaften des Modells blieben erhalten. Beim evolutionären Entwurf will man aber nicht selten einige der ursprünglichen Eigenschaften des Modells bewusst verändern, um beispielsweise den Anforderungen besser zu genügen.

Beim Architekturentwurf will der Softwarearchitekt in erster Line das Modell evolutionär verbessern sowie weiter entwickeln, und nicht verfeinern. Nach jedem Evolutionsschritt steht die Frage nach der Widerspruchsfreiheit und der Angemessenheit des Architekturmodells im Vordergrund (vgl. Kapitel 1 und 2):

- Widerspruchsfreiheit: Ist das weiter entwickelte Modell syntaktisch und semantisch in sich konsistent und kann eine effiziente Implementierung realisiert werden?
- Angemessenheit: Erfüllt das weiter entwickelte Modell die Anforderungen an das System besser als das ursprüngliche Modell?

Die erste Frage ist im Prinzip einfach zu beantworten: Verwenden wir formal fundierte Beschreibungstechniken, so kann man aus einem Modell eine Implementierung generieren, wenn es syntaktisch und semantisch konsistent ist. Eine erfolgreiche und korrekte Generierung zeigt somit die Widerspruchsfreiheit des Modells.

Ist das Modell nicht widerspruchsfrei, so stellt sich speziell bei komponentenbasierten Systemen die weiterführende Frage, welche Veränderungen an einzelnen Komponentenspezifikationen zu den Inkonsistenzen im Modell geführt haben. Die Fragestellung der Widerspruchsfreiheit von Modellen komponentenbasierter Systeme lässt sich dementsprechend präzisieren:

1. Existiert eine Implementierung des Modells und kann sie generiert werden?
2. Welche Eigenschaften der im Modell beschriebenen Komponenten sind inkonsistent?

Die Frage der Angemessenheit ist wesentlich schwieriger. Im Kern handelt es sich dabei um eine qualitative Bewertung von Architekturmodellen gegenüber den Anforderungen. Natürlich gibt es Möglichkeiten, diese Bewertung zu quantifizieren, wie zum Beispiel die Anzahl der Testfälle, die erfolgreich von den Modellen absolviert werden. Diese Quantifizierung

kann die Bewertung aber nur unterstützen. Letztlich muss die Entscheidung, welches Modell besser ist, dem Softwarearchitekten überlassen werden. Denn beim Entwurf muss der Architekt auch die Möglichkeit haben, einen Evolutionsschritt durchzuführen, der keine sichtbaren Vorteile bringt, aber die Grundlagen für kommende Entwicklungsschritte schafft, von denen sich der Architekt dann wesentliche Verbesserungen erhofft.

Aufgabe der evolutionären Entwurfsmethodik bleibt es aber, dem Architekten eine adäquate Informationsbasis für die Modellbewertung zur Verfügung zu stellen. Die Methodik muss die Auswirkungen eines Evolutionsschrittes transparent darstellen können. Im Zentrum steht die Fragestellung, welche Eigenschaften des Modells im Rahmen eines Evolutionsschrittes stabil geblieben, neu hinzugekommen oder weggefallen sind, sowie sich verändert haben[1]. Da ein komponentenbasiertes System aus einzelnen Softwarekomponenten besteht, ist die Angemessenheit des weiter entwickelten Modells differenzierter zu betrachten:

3. Welche Eigenschaften der im Modell beschriebenen Komponenten sind stabil geblieben?
4. Welche Eigenschaften der im Modell beschriebenen Komponenten wurden ergänzt, entfernt oder verändert?

Eine der zentralen Aufgaben des evolutionären methodischen Architekturentwurfs komponentenbasierter Systeme ist es, den Softwarearchitekten bei der Beantwortung dieser vier Fragen zu unterstützen. Die existierenden formalen Ansätze konzentrieren sich meist nur auf die erste Fragestellung, die Konsistenz des Modells und die damit verbundene Generierung einer Implementierung (vgl. Kapitel 3.1). In Kapitel 8 werden wir ebenfalls ein entsprechendes Werkzeug vorstellen, das es uns erlaubt, aus unseren Beschreibungen eine Implementierung zu generieren.

Im Zentrum der anderen Fragestellungen, Frage 2 bis 4, stehen Eigenschaften von Komponenten und deren Veränderungen. Da die existierenden formalen Ansätze nicht explizit über Eigenschaften von Komponenten und deren Abhängigkeiten sprechen, können diese Fragestellungen nicht adäquat beantwortet werden. Im nächsten Kapitel werden wir diese Defizite noch eingehender diskutieren.

3.3 Evolution als Vergröberung und Verfeinerung

Wie im vorhergehenden Kapitel und bei [Parn94] beschrieben, sind Evolution und Verfeinerung methodisch unterschiedlich zu betrachten. Im Allgemeinen existiert keine Verfeinerungsbeziehung zwischen dem ursprünglichen Modell $\mathsf{Document} \subseteq \mathsf{DOCUMENT}$ und dem evolutionär weiter entwickelten Modell $\mathsf{evolve}(\mathsf{Document})$. Trotzdem kann eine systemmodellbasierte formale Semantik und das mathematische Konzept der Verfeinerung verwendet werden, um die Evolution von Modellen formal zu charakterisieren.

Dieser Ansatz ist in der Literatur auch als das „Abstraction Refinement Model" bekannt (siehe [KN93]). Im folgenden werden die Grundzüge dieses Ansatzes kurz skizziert. Dabei wird deutlich, dass die wesentlichen Fragestellungen des evolutionären Architekturentwurfs (vgl. Kapitel 3.2) durch eine systemmodellbasierte formale Semantik und das Abstraction Refinement Model nicht ausreichend beantwortet werden können.

[1] Die Fragestellung, ob eine Eigenschaft verändert wurde, oder ob sie weggefallen ist und eine neue hinzugekommen ist, spielt in einigen Fällen eine entscheidende Rolle, beispielsweise bei der Evolution von persistenten Daten. Allerdings ist diese Frage nicht ohne weitere Informationen entscheidbar. Siehe dazu auch Kapitel 8.

Das grundlegende Konzept des Abstraction Refinement Model beruht auf dem mathematischen Begriff des vollständigen Verbandes. Bei einem vollständigen Verband existiert zu zwei oder mehreren Elementen stets ein Maximum und ein Minimum. Da mit jeder Potenzmenge $P(A)$ einer beliebigen Menge A auch ein vollständiger Verband $(P(A),\cup,\cap)$ mit der zugehörigen Halbordnung $\supseteq$ (bzw. $\subseteq$) gegeben ist (vgl. [Goos95]), ist die Potenzmenge der Menge aller Systeme $P(\text{SYSTEM})$ ebenfalls ein vollständiger Verband. Der Wertebereich der Abbildung sem, der systemmodellbasierten formalen Semantik von Dokumentenmengen, ist $P(\text{SYSTEM})$. Zusammen mit der Verfeinerungsrelation $\supseteq$ ist $P(\text{SYSTEM})$ somit ein vollständiger Verband.

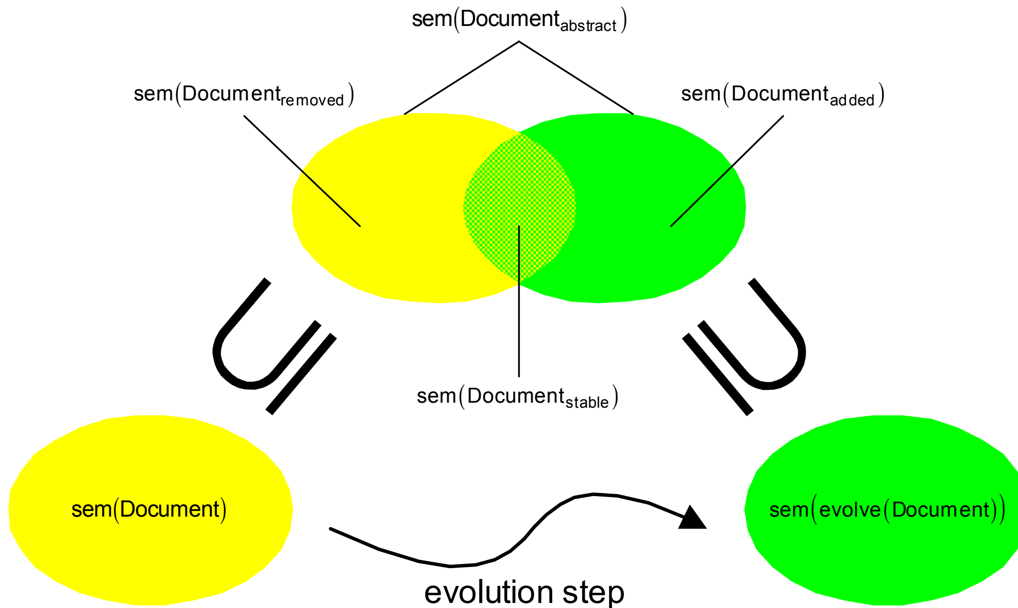

Abbildung 3.1: Evolution als Vergröberung und Verfeinerung

Demzufolge existiert zu den zwei Systemmengen $\text{sem}(\text{Document})$ und $\text{sem}(\text{evolve}(\text{Document}))$, die den Modellen $\text{Document} \subseteq \text{DOCUMENT}$ und $\text{evolve}(\text{Document})$ zugeordnet sind, stets ein Maximum, die Vereinigungsmenge:

$$\text{SystemMax} = \text{sem}(\text{Document}) \cup \text{sem}(\text{evolve}(\text{Document}))$$

Werden nun geeignete Beschreibungstechniken mit einer entsprechenden formalen Semantik verwendet, so existiert für das Maximum SystemMax sogar eine Beschreibung und sie kann angegeben werden:

$$\text{sem}(\text{Document}_{\text{abstract}}) = \text{SystemMax} = \text{sem}(\text{Document}) \cup \text{sem}(\text{evolve}(\text{Document}))$$

Wie in Abbildung 3.1 dargestellt, steht $\text{Document}_{\text{abstract}} \subseteq \text{DOCUMENT}$ dabei in einer Verfeinerungsbeziehung zu den anderen zwei Modellen und stellt somit eine Vergröberung dieser Modelle dar. Es gilt:

$$\text{sem}(\text{Document}_{\text{abstract}}) \supseteq \text{sem}(\text{Document}) \wedge \text{sem}(\text{Document}_{\text{abstract}}) \supseteq \text{sem}(\text{evolve}(\text{Document}))$$

Basierend auf diesem Ansatz des Abstraction Refinement Models lassen sich Modelländerungen, die mit einem Evolutionsschritt verbunden sind, wie folgt formal charakterisieren (vgl. auch Abbildung 3.1):

$\text{Document}_{\text{stable}} \subseteq \text{DOCUMENT}$ ist genau dann das Modell der Systeme mit den Eigenschaften, die stabil geblieben sind, wenn folgende Bedingung gilt:

$$\text{sem}(\text{Document}_{\text{stable}}) = \text{sem}(\text{Document}) \cap \text{sem}(\text{evolve}(\text{Document}))$$

$\text{Document}_{\text{removed}} \subseteq \text{DOCUMENT}$ ist genau dann das Modell der Systeme mit den Eigenschaften, die verworfen wurden, wenn folgende Bedingung gilt:

$$\text{sem}(\text{Document}_{\text{removed}}) = \text{sem}(\text{Document}) \setminus \text{sem}(\text{evolve}(\text{Document}))$$

Und schließlich, $\text{Document}_{\text{added}} \subseteq \text{DOCUMENT}$ ist genau dann das Modell der Systeme mit den Eigenschaften, die neu hinzugenommen werden, wenn folgende Bedingung gilt:

$$\text{sem}(\text{Document}_{\text{added}}) = \text{sem}(\text{evolve}(\text{Document})) \setminus \text{sem}(\text{Document})$$

Wie bereits angesprochen, können die zusätzlichen Modelle $\text{Document}_{\text{abstract}}$, $\text{Document}_{\text{stable}}$, $\text{Document}_{\text{removed}}$ und $\text{Document}_{\text{added}}$ sogar konstruktiv angegeben werden, wenn geeignete Beschreibungstechniken mit einer entsprechenden formalen Semantik verwendet werden, wie das folgende Beispiel illustriert:

Angenommen, wir würden Automatenmodelle verwenden [HU88, Rump96], dann beschreibt $\text{Document}_{\text{abstract}}$ einen Automaten, der einen Startzustand mit zwei ε-Übergängen hat: Einen ε-Übergang zum Startzustand des Automaten des ursprünglichen Modells Document und einen anderen ε-Übergang zum Startzustand des Automaten des weiter entwickelten Modells $\text{evolve}(\text{Document})$. Somit ist der Automat in $\text{Document}_{\text{abstract}}$ die Disjunktion der zwei Automaten in Document und $\text{evolve}(\text{Document})$.

In $\text{Document}_{\text{stable}}$ ist ein Automat beschrieben, der die Automaten aus Document und $\text{evolve}(\text{Document})$ simuliert und genau dann eine Ausgabe x erzeugt, wenn die beiden simulierten Automaten ebenfalls die Ausgabe x erzeugen. Der Automat in $\text{Document}_{\text{stable}}$ ist die Konjunktion der zwei Automaten in Document und $\text{evolve}(\text{Document})$.

$\text{Document}_{\text{removed}}$ beschreibt einen Automaten, der die Automaten aus Document und $\text{evolve}(\text{Document})$ simuliert und genau dann eine Ausgabe x erzeugt, wenn der Automat aus Document die Ausgabe x erzeugt und der Automat aus $\text{evolve}(\text{Document})$ die Ausgabe x nicht erzeugt. Der Automat in $\text{Document}_{\text{removed}}$ ist die Konjunktion des Automaten in Document und dem Negat des Automaten in $\text{evolve}(\text{Document})$.

Und schließlich, in $\text{Document}_{\text{added}}$ ist ein Automat beschrieben, der die Automaten Document und $\text{evolve}(\text{Document})$ simuliert und genau dann eine Ausgabe x erzeugt, wenn der Automat aus Document die Ausgabe x nicht erzeugt und der Automat aus $\text{evolve}(\text{Document})$ die Ausgabe x erzeugt. Der Automat in $\text{Document}_{\text{added}}$ ist die Konjunktion des Negats des Automaten in Document und dem Automaten in $\text{evolve}(\text{Document})$.

Zusammenfassend kann man festhalten: Wird ein Modell $\text{Document} \subseteq \text{DOCUMENT}$ weiter entwickelt zu $\text{evolve}(\text{Document})$, so liefert eine systemmodellbasierte formale Semantik unter der Verwendung von geeigneten Beschreibungstechniken zusammen mit dem darauf aufbauenden Abstraction Refinement Model vier zusätzliche Modelle: $\text{Document}_{\text{abstract}}$, $\text{Document}_{\text{stable}}$, $\text{Document}_{\text{removed}}$ und $\text{Document}_{\text{added}}$. Es stellt sich somit die Frage, ob dieser Ansatz genügt, um die Fragestellungen des evolutionären Entwurfs aus Kapitel 3.2 adäquat zu beantworten.

Die erste Frage, die Existenz einer Implementierung des Modells und deren effektive Generierung, wird unabhängig davon durch die Festlegung einer geeigneten systemmodellbasierten formalen Semantik für Dokumentenmengen beantwortet (siehe [Rump96, Berg97, Krüg00]).

Welche Veränderungen von Eigenschaften der im Modell beschriebenen Komponenten zu einem inkonsistenten Modell geführt haben – die zweite Frage – ist speziell bei der evolutio-

nären Entwicklung komponentenbasierter Systeme von zentraler Bedeutung. Mit dem vorgestellten Ansätzen können hierzu aber keine Aussagen gemacht werden. Dies ist eine inhärentes Problem des Ansatzes einer systemmodellbasierten formalen Semantik von Dokumentenmengen. Denn, nichttriviale Aussagen über inkonsistente Modelle sind nicht möglich, da die Semantik eines inkonsistenten Modells laut Definition die leere Menge von Systemen ist (vgl. Kapitel 3.1: $\text{consistent}(\text{Document}) \Leftrightarrow_{\text{def}} \text{sem}(\text{Document}) \neq \emptyset$). Darüber hinaus sind die geforderten spezifischeren Aussagen über Komponenten auf dieser Ebene auch nicht möglich, da die notwendige Strukturierung von Systemen in Komponenten nicht vorhanden ist.

Für die dritte und vierte Frage, welche Eigenschaften sowohl im alten als auch im neuen Architekturmodell stabil geblieben, neu hinzugekommen, weggefallen sind oder sich verändert haben, bietet das Abstraction Refinement Model dem Architekten vier zusätzliche Modelle. Diese Modelle beschreiben jedoch vollständige Systeme. Die Eigenschaften des neuen Modells, die sich durch den Evolutionsschritt verändert haben, muss sich der Architekt selbst aus diesen Modellen erarbeiten. Insbesondere bei größeren, komplexen Systemen erscheint es auf Grund der Größe illusorisch, dass man diese zusätzlichen Modelle durchdringen und davon Eigenschaften ableiten kann. Außerdem sind detailliertere Aussagen über Komponenten auf dieser Ebene wiederum nicht möglich, da eine Strukturierung von Systemen nicht existiert.

Der Ansatz, der mit einer systemmodellbasierten formalen Semantik und dem Abstraction Refinement Model verfolgt wird, liefert somit noch keine ausreichende Grundlage für die formale Konzeption des evolutionären Entwurfs komponentenbasierter Systeme. Die grundsätzliche Problematik dieses Ansatzes ist es, dass

- keine spezifischen Aussagen über inkonsistente Modelle getroffen werden können,
- eine Strukturierung des Systemmodells und des syntaktischen Modells in deren Bestandteile nicht vorhanden ist, und
- eine präzise Beschreibung und Formalisierung der Abhängigkeiten zwischen diesen Bestandteilen nicht möglich ist.

Im folgenden Kapitel werden wir einen neuen, verbesserten Ansatz vorstellen, der diese Defizite nicht beinhaltet und damit die formale Grundlage für den evolutionären Architekturentwurf bereit stellt.

3.4 Prädikatenbasierte formale Semantik

Wie wir bereits im vorhergehenden Kapitel argumentiert haben, ist eine systemmodellbasierte formale Semantik und das darauf aufbauende Abstraction Refinement Model keine ausreichende Basis für den evolutionären Entwurf komponentenbasierter Systeme. Die grundlegenden Fragestellungen aus Kapitel 3.2 konnten nicht zufriedenstellend behandelt werden.

Deshalb erarbeiten wir in diesem Kapitel einen neuen Ansatz, das Konzept einer prädikatenbasierten formalen Semantik. Wir führen diesen Ansatz zuerst unabhängig zu dem Konzept einer systemmodellbasierten formalen Semantik ein. Abschließend, am Ende dieses Kapitels, werden wir dann zeigen, wie diese zwei konkurrierenden Ansätze integriert werden können.

Vor dem Hintergrund der Defizite einer systemmodellbasierten formalen Semantik führen wir im folgenden zuerst eine Strukturierung des Systemmodells sowie des syntaktischen Modells in deren Bestandteile ein:

- Die Menge der Instanzen $\mathsf{INSTANCE}$ beschreibt dabei alle Bestandteile eines Systems zur Laufzeit, wie zum Beispiel Systeminstanzen, Komponenteninstanzen, Klasseninstanzen, Attributeinstanzen von Klasseninstanzen oder die Werte von Attributen.
- Die Menge der Spezifikationsbezeichner $\mathsf{SPECIFIER}$ charakterisiert alle identifizierbaren Einheiten einer Beschreibung eines Systems zur Entwicklungszeit, wie zum Beispiel Systembeschreibungen, Komponentenbeschreibungen, Klassenbeschreibungen oder Attributbeschreibungen. Von diesen Spezifikationsbezeichner werden dann zur Laufzeit Instanzen erzeugt.

Die Zuordnung zwischen den Instanzen und den Spezifikationsbezeichnern modelliert die Funktion $\mathsf{specified}$.

Definition 3.5: Instanzen und Spezifikationsbezeichner eines semantischen Modells

Ein semantisches Modell charakterisiert die Menge aller gültigen Instanzen $\mathsf{INSTANCE}$ *und die Menge aller gültigen Spezifikationsbezeichner* $\mathsf{SPECIFIER}$. *Über die Funktion* $\mathsf{specified}$ *wird jeder Instanz eindeutig ein entsprechender Spezifikationsbezeichner zugeordnet:*

$$\mathsf{specified} : \mathsf{INSTANCE} \rightarrow \mathsf{SPECIFIER}$$

Diese Definition liefert uns eine erste, feinere Unterteilung des semantischen Modells in Instanzen und Spezifikationsbezeichner[1]. Die semantische Fundierung der Spezifikationsbezeichner basiert aber nicht, wie bei der systemmodellbasierten formalen Semantik und im Abstraction Refinement Model, auf einer Abbildung von der Menge der Spezifikationsbezeichner in die Menge der Instanzen (vgl. Definition 3.3). Dieser Ansatz würde wiederum dazu führen, dass keine weitergehenden Aussagen über ein inkonsistentes Modell möglich sind, für das keine Implementierung existiert.

Deshalb benötigen wir eine andere semantische Basis als die betrachteten Systeme selbst. Hierfür verwenden wir die Menge aller logischen Ausdrücke der ersten Stufe mit freien Variablen, die Menge TERM. Beispielsweise könnte ein Ausdruck $\mathsf{t} \in \mathsf{TERM}$ festhalten, dass alle Attributinstanzen in einem System, die Instanzen des Spezifikationsbezeichners $\mathsf{AttributeSpecifierWithConstantValueFive} \in \mathsf{SPECIFIER}$ sind, immer den Wert 5 haben. Das betrachtete System wird in dem Term durch die Variable v gekennzeichnet[2]:

$$\forall \mathsf{a} \in \mathsf{Attribute}_{\mathsf{v}} . \mathsf{specified}(\mathsf{a}) = \mathsf{AttributeSpecifierWithConstantValueFive} \Rightarrow (\mathsf{a},5) \in \mathsf{valuation}_{\mathsf{v}}$$

In diesem Beispiel ist die einzige freie Variable die in dem Term t vorkommt v. Alle anderen Parameter in dem Term werden durch diese einzige Variable bestimmt. Für unser semantische Modell sind Terme mit nur einer einzigen freien Variablen ausreichend. Diese Variable kann dann mit einer konkreten Systeminstanz $\mathsf{s} \in \mathsf{INSTANCE}$ belegt werden. Ist der logische Ausdruck wahr, so ist die Systeminstanz eine gültige Implementierung bzw. Belegung, es gilt die Modellbeziehung (siehe dazu auch [EFT92]).

Dieser Ansatz der semantischen Fundierung, den wir prädikatenbasierte formale Semantik von Spezifikationsbezeichnern nennen, ersetzt den ursprünglichen Begriff der systemmodellbasierten formalen Semantik von Dokumentenmengen aus Definition 3.3:

---

[1] Eine weitergehende Strukturierung und die Definition der komplexen Beziehungen zwischen Instanzen sowie zwischen Spezifikationsbezeichnern werden in den Kapiteln 5, 6 und 7 vorgenommen.

[2] Einige Bezeichner, die in diesem Term verwendet werden, werden erst im Kapitel 5 eingeführt. Zum Verständnis des Konzeptes der prädikatenbasierten formalen Semantik genügt bereits die vorgestellte informelle Erklärung des Termes.

Definition 3.6: Prädikatenbasierte formale Semantik von Spezifikationsbezeichnern

Sei TERM^v *die Menge aller logischen Ausdrücke der ersten Stufe mit der einzigen freien Variablen* v. *Eine Instanz* $i \in \mathsf{INSTANCE}$ *ist eine gültige Belegung bzw. Interpretation eines logischen Ausdruck* $t \in \mathsf{TERM}^v$, *wenn die Modellbeziehung* $t[i]$ *gilt:*

$$.[.] : \mathsf{TERM}^v \times \mathsf{INSTANCE} \rightarrow \mathsf{BOOLEAN}$$

Gilt bei einer Instanz i *die Modellbeziehung* $t[i]$, *so sagen wir auch* i *implementiert die Eigenschaft* t *oder das Prädikat* t *gilt bei* i.

Anhand dieser Definition können wir nun die Gültigkeit von Eigenschaften bei Instanzen überprüfen. Die Eigenschaften sind dabei in den Beschreibungen der Spezifikationsbezeichnern formuliert. Dabei unterscheiden wir zwei Kategorien von Eigenschaften, die Spezifikationsbezeichnern, insbesondere Komponenten, zugeordnet werden: Die Menge der angenommenen Eigenschaften und die Menge der zugesicherten Eigenschaften.

Definition 3.7: Eigenschaften von Spezifikationsbezeichnern

Sei TERM^v *die Menge aller Eigenschaften, präziser die Menge aller logischen Ausdrücke der ersten Stufe mit der einzigen freien Variablen* v. *Jedem Spezifikationsbezeichner* $sp \in \mathsf{SPECIFIER}$ *werden durch die folgenden Funktionen die angenommen und zugesicherten Eigenschaften zugeordnet:*

$$\mathsf{assured} : \mathsf{SPECIFIER} \rightarrow \mathcal{P}\left(\mathsf{TERM}^v\right)$$
$$\mathsf{required} : \mathsf{SPECIFIER} \rightarrow \mathcal{P}\left(\mathsf{TERM}^v\right)$$

Dabei gilt:

$$\forall a_1, a_2 \in \mathsf{assured}(sp) \Rightarrow a_1 \wedge a_2 \in \mathsf{assured}(sp)$$
$$\forall r_1, r_2 \in \mathsf{required}(sp) \Rightarrow r_1 \wedge r_2 \in \mathsf{required}(sp)$$

Die Funktion $\mathsf{required}$ *charakterisiert das Axiomensystem eines Spezifikationsbezeichners* $sp \in \mathsf{SPECIFIER}$, *eine Menge von Eigenschaften. Die Funktion* $\mathsf{assured}$ *beschreibt die Eigenschaften des Spezifikationsbezeichners* sp, *die bei der Belegung mit einer Instanz* $i \in \mathsf{INSTANCE}$ *genau dann gültig sind, wenn alle Eigenschaften des Axiomensystems bei der Belegung mit der Instanz* i *gelten. Es gilt:*

$$\forall a \in \mathsf{assured}(sp), i \in \mathsf{INSTANCE} . \left(\bigwedge_{\forall r \in \mathsf{required}(sp)} r \right)[i] \Rightarrow a[i]$$

Diese Charakterisierung ermöglicht es, die Abhängigkeiten zwischen den Spezifikationsbezeichner eines Systems sehr detailliert zu modellieren, bis auf die Ebene einzelner Eigenschaften. Hierfür wird für jeden Spezifikationsbezeichner ein Annahme-/Zusicherungsvertrag definiert:

Definition 3.8: Annahme-/Zusicherungsvertrag (Requirement Assurance Contract)

Ein Annahme-/Zusicherungsvertrag $\mathsf{Contract} \subseteq \mathsf{CONTRACT}$ *ist eine Relation zwischen Eigenschaften, die jeweils Spezifikationsbezeichnern zugeordnet sind:*

$$\mathsf{CONTRACT} =_{def} \mathsf{TERM}^v \times \mathsf{SPECIFIER} \times \mathsf{TERM}^v$$

Die Funktion $\mathsf{fulfilled}$ *charakterisiert, ob ein Annahme-/Zusicherungsvertrag in einer Menge von Spezifikationsbezeichnern gültig ist. Dies ist genau dann der Fall, wenn alle Eigenschaf-*

ten, die ein bestimmter Spezifikationsbezeichner aus dieser Menge annimmt, aus zugesicherten Eigenschaften von Spezifikationsbezeichnern aus dieser Menge abgeleitet werden können:

$$\text{fulfilled} : \text{CONTRACT} \times \mathrm{P}(\text{SPECIFIER}) \rightarrow \text{BOOLEAN}$$

Sei $\text{Specifier} \subseteq \text{SPECIFIER}$ *eine beliebige Menge von Spezifikationsbezeichnern und sei* $\text{Contract}_{sp} \subseteq \text{CONTRACT}$ *der Annahme-/Zusicherungsvertrag des Spezifikationsbezeichners* $sp \in \text{Specifier}$ *, so gilt:*

$$\text{fulfilled}(\text{Contract}_{sp}, \text{Specifier}) \Leftrightarrow_{def}$$
$$\forall r \in \text{required}(sp) \Rightarrow \exists (r,x,a) \in \text{Contract}_{sp} \,.\, x \in \text{Specifier} \wedge a \in \text{assured}(x) \wedge \text{holds}(a,r)$$

Dabei ist $\text{holds}(a,r)$ *ein Prädikat, das auf Eigenschaften definiert ist. Gilt* $\text{holds}(a,r)$ *so impliziert die Eigenschaft* a *die Eigenschaft* r *.*

$$\text{holds} : \text{TERM}^{V} \times \text{TERM}^{V} \rightarrow \text{BOOLEAN}$$

Dabei gilt:

$$\text{holds}(a,r) \Leftrightarrow_{def} (a[i] \Rightarrow r[i])$$

Das Prädikat $\text{holds}(a,r)$ ist genau dann gültig, wenn die Eigenschaft a bei der Belegung mit der Instanz $i \in \text{INSTANCE}$ die Eigenschaft r ebenfalls bei der Belegung mit der Instanz i impliziert. Die Berechnung des Prädikates $\text{holds}(a,r)$ hängt dabei von der konkreten Syntax ab, mit der die Eigenschaften dargestellt werden. In Kapitel 7.4 wird ein einfaches Schema für die Berechnung des Prädikates vorgestellt.

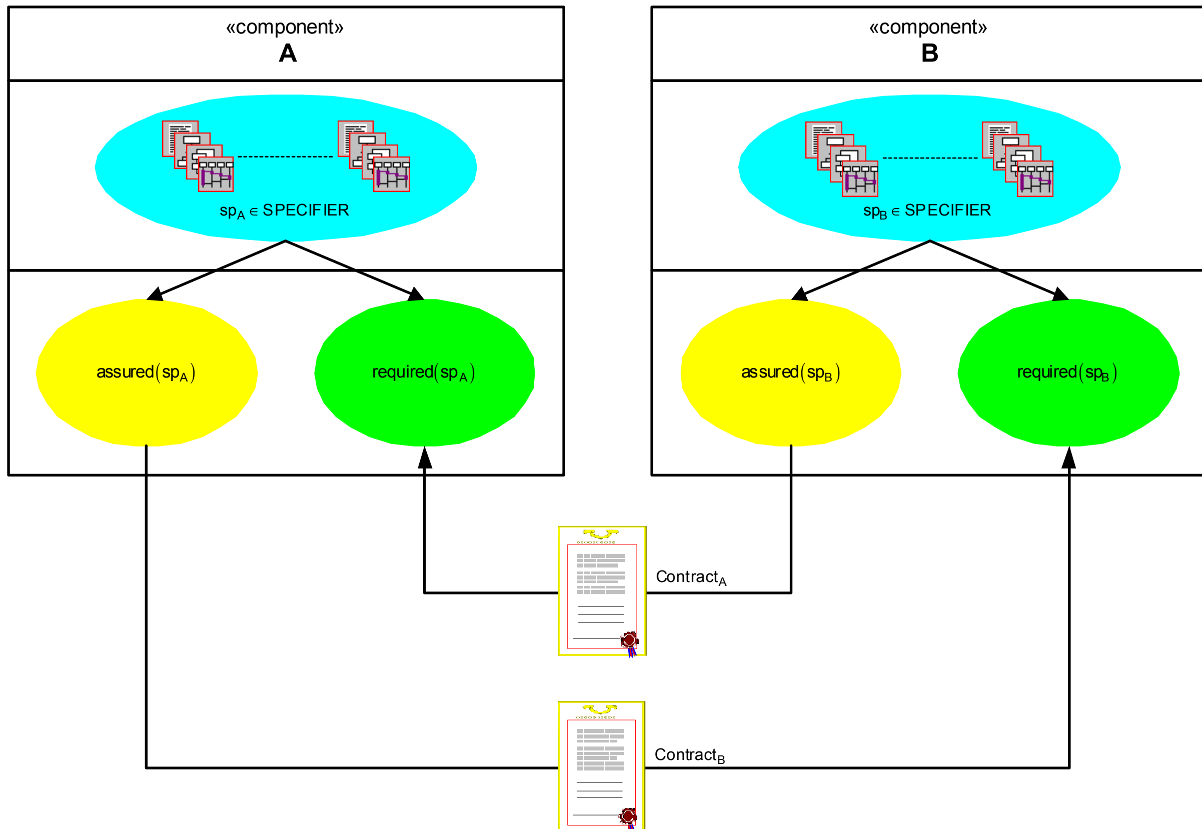

Abbildung 3.2: Prädikatenbasierte Semantik und Annahme-/Zusicherungsverträge

Wie in Abbildung 3.2 beispielhaft dargestellt, sind die Annahmen eines Spezifikationsbezeichners, insbesondere einer Komponente, genau dann erfüllt, wenn alle geforderten Eigen-

schaften in dem zugehörigen Annahme-/Zusicherungsvertrag enthalten und Eigenschaften zugeordnet sind, die von anderen Spezifikationsbezeichnern zugesichert werden. Außerdem gilt eine Folgerungsbeziehung zwischen der zugesicherten und den benötigten Eigenschaften.

Mit dem Konzept der Annahme-/Zusicherungsverträge können die Abhängigkeiten zwischen Spezifikationsbezeichnern explizit modelliert werden. Diese Abhängigkeiten können mit der Funktion $\mathsf{fulfilled}$ bei Bedarf auf Konsistenz überprüft werden.

In den gängigen Programmiersprachen, CASE-Tools oder formalen Ansätzen ist es bereits jetzt möglich, bestimmte syntaktische Abhängigkeiten auf Konsistenz zu überprüfen. Im wesentlichen wird dabei aber nur validiert, ob jeder Bezeichner, der verwendet wird, auch definiert ist,. Aus der Funktion $\mathsf{fulfilled}$ lässt sich diese rein syntaktische Konsistenz der Abhängigkeiten direkt ableiten, denn es gilt:

$$\begin{aligned}&\mathsf{fulfilled}\left(\mathsf{Contract}_{sp},\mathsf{Specifier}\right)\Rightarrow\\&\quad\forall r\in \mathsf{required}(sp)\Rightarrow \exists(r,x,a)\in \mathsf{Contract}_{sp}\,.\,x\in \mathsf{Specifier}\wedge a\in \mathsf{assured}(x)\end{aligned}$$

Darüber hinaus bildet das vorliegende Modell aber über die weitere Bedingung $\mathsf{holds}(a,r)$ zusätzlich die semantische Konsistenz der Abhängigkeiten ab. Verwendete Bezeichner sind somit nicht nur definiert, sondern sie leisten auch das, was von ihnen erwartet wird. Diese entscheidende Verbesserung ermöglicht es, Abhängigkeiten zwischen Komponenten nicht nur auf syntaktischer Ebene, sondern auch auf semantischer Ebene zu modellieren und zu spezifizieren.

Werden im Rahmen eines Evolutionsschrittes einzelne Komponentenspezifikationen verändert, so kann die Konsistenz der Systemspezifikation über die Funktion $\mathsf{fulfilled}$ ermittelt werden. Eine Systemspezifikation eines komponentenbasierten Softwaresystems besteht dabei aus einer Menge von Spezifikationsbezeichnern und den zugehörigen Annahme-/Zusicherungsverträgen, wie in Abbildung 3.2 bereits illustriert wurde:

Definition 3.9: Spezifikation eines komponentenbasierten Softwaresystems

$\mathsf{SPECIFICATION}$ *sei die Menge aller Beschreibungen komponentenbasierter Systeme. Eine Spezifikation* $\mathsf{spec}\in\mathsf{SPECIFICATION}$ *besteht aus*

a) $\mathsf{Specifier}_{spec}\subseteq\mathsf{SPECIFIER}$ *, einer Menge von Spezifikationsbezeichnern,*

b) $\mathsf{Contract}_{spec}\subseteq\mathsf{CONTRACT}$ *, einer Menge von Annahme-/Zusicherungsverträgen, und*

c) $\mathsf{contract_of}_{spec}:\mathsf{Specifier}_{spec}\rightarrow\mathsf{Contract}_{spec}$ *, einer Abbildung, die jedem Spezifikationsbezeichner einen Annahme-/Zusicherungsvertrag zuordnet.*

Die prädikatenbasierte formale Semantik von Spezifikationsbezeichnern aus Definition 3.7 kann auf Spezifikationen entsprechend erweitert werden. Die zugesicherten Eigenschaften einer Spezifikation ergeben sich direkt aus den zugesicherten Eigenschaften der einzelnen Spezifikationsbezeichner. Die angenommen Eigenschaften einer Spezifikation, sind die angenommen Eigenschaften der einzelnen Spezifikationsbezeichner, die noch nicht durch Annahme-/Zusicherungsverträge gewährleistet sind:

Definition 3.10: Eigenschaften von Spezifikationen komponentenbasierter Systeme

Die prädikatenbasierte formale Semantik einer Spezifikation $\mathsf{spec}\in\mathsf{SPECIFICATION}$ *ist als Erweiterung der prädikatenbasierten formalen Semantik von Spezifikationsbezeichner wie folgt definiert:*

$$\text{assured}:\text{SPECIFICATION}\rightarrow \mathrm{P}\left(\text{TERM}^{\text{v}}\right)$$
$$\text{required}:\text{SPECIFICATION}\rightarrow \mathrm{P}\left(\text{TERM}^{\text{v}}\right)$$

Wobei gilt:

$$\text{assured}(\text{spec}) =_{\text{def}} \left\{ a \,\middle|\, \left(a \in \bigcup_{\forall sp \in \text{Specifier}_{\text{spec}}} \text{assured}(sp) \right) \vee \left((a = a_1 \wedge a_2) \wedge (a_1 \in \text{assured}(\text{spec})) \wedge (a_1 \in \text{assured}(\text{spec})) \right) \right\}$$
$$\text{required}(\text{spec}) =_{\text{def}} \left\{ r \in \text{required}(sp) \,\middle|\, sp \in \text{Specifier}_{\text{spec}} \wedge \nexists (r,x,a) \in \text{contract_of}_{\text{spec}}(sp) \right\}$$

Definition 3.11: Konsistenz und Vollständigkeit von Spezifikation

Eine Spezifikation $\text{spec} \in \text{SPECIFICATION}$ *ist genau dann konsistent, wenn alle Annahme-/Zusicherungsverträge gültig sind:*

$$\text{consistent}(\text{spec}) \Leftrightarrow_{\text{def}} \forall c \in \text{Contract}_{\text{spec}} \Rightarrow \text{fulfilled}\left(c, \text{Specifier}_{\text{spec}}\right)$$

Eine Spezifikation $\text{spec} \in \text{SPECIFICATION}$ *ist genau dann vollständig, wenn keine angenommenen Eigenschaften mehr vorhanden sind: Wir schreiben kurz:*

$$\text{complete}(\text{spec}) \Leftrightarrow_{\text{def}} \text{required}(\text{spec}) = \emptyset$$

Abschließend können wir nun den Zusammenhang zwischen systemmodellbasierter und prädikatenbasierter formaler Semantik als These formulieren:

These 3.1: Zusammenhang von systemmodell- und prädikatenbasierter Semantik

Die Menge der Systeme SYSTEM *sei eine Teilmenge der Menge aller Instanzen* INSTANCE*, und die Potenzmenge der Entwicklungsdokumente* $\mathrm{P}(\text{DOCUMENT})$ *sei gleich der Menge der Spezifikationen* SPECIFICATION*:*

$$\text{SYSTEM} \subseteq_{\text{def}} \text{INSTANCE}$$
$$\mathrm{P}(\text{DOCUMENT}) =_{\text{def}} \text{SPECIFICATION}$$

Für jede Spezifikation $\text{spec} \in \text{SPECIFICATION}$ *existiert genau dann eine Implementierung, wenn die Spezifikation konsistent und vollständig ist:*

$$\text{consistent}(\text{spec}) \wedge \text{complete}(\text{spec}) \Leftrightarrow_{\text{def}} \text{sem}(\text{spec}) \neq \emptyset$$

Sei eine Spezifikation $\text{spec} \in \text{SPECIFICATION}$ *konsistent und vollständig, so ist jede Implementierung der Spezifikation in der Menge der Systeme enthalten, die in der formalen Semantik der Dokumentenmenge enthalten sind und umgekehrt:*

$$\text{consistent}(\text{spec}) \wedge \text{complete}(\text{spec}) \Leftrightarrow_{\text{def}} \text{sem}(\text{spec}) = \left\{ \text{sys} \in \text{SYSTEM} \,\middle|\, \forall a \in \text{assured}(\text{spec}) \Rightarrow a[\text{sys}] \right\}$$

Im Laufe dieser Arbeit werden wir diese These untermauern. So werden wir zum Beispiel in Kapitel 8 ein Werkzeug vorstellen, das aus konsistenten und vollständigen Spezifikationen eine Implementierung generiert. Somit zeigt dieses Werkzeug konstruktiv dass zumindest folgendes gilt:

$$\text{consistent}(\text{spec}) \wedge \text{complete}(\text{spec}) \Rightarrow \text{sem}(\text{spec}) \neq \emptyset$$

Ein vollständiger und abschließender Beweis der voranstehenden These 3.1 kann aber im Rahmen dieser Arbeit nicht erbracht werden. Dazu müssten wir nicht nur die prädikatenbasierte formale Semantik von Spezifikationen erarbeiten, sondern noch zusätzlich die system-

modellbasierte formale Semantik von Dokumentenmengen. Dies würde aber den Rahmen der vorliegenden Dissertation sprengen.

3.5 Widerspruchsfreiheit und Angemessenheit von Spezifikationen

Im letzten Kapitel haben wir ein neues semantisches Modell für komponentenbasierte Systeme vorgestellt, die prädikatenbasierte formale Semantik von Spezifikationen. Dieses Modell erhebt den Anspruch, eine adäquatere formale Grundlage für den evolutionären Architekturentwurf zu bieten als die systemmodellbasierte formale Semantik von Dokumentenmengen und das darauf aufbauende Abstraction Refinement Model. In diesem Kapitel werden wir zeigen, dass die Fragestellungen der Widerspruchsfreiheit und der Angemessenheit mit dem neuen Modell besser beantwortet werden können.

Ist eine Spezifikation widerspruchsfrei, so existiert eine Implementierung der Spezifikation (vgl. Frage 1 aus Kapitel 3.2). In Kapitel 8 stellen wir ein Werkzeug vor, das aus konsistenten und vollständigen Spezifikationen eine erste prototypische Implementierung generiert. Diese Teilfrage wird damit direkt beantwortet.

Wird eine Spezifikation $\mathrm{spec} \in \mathrm{SPECIFICATION}$, die konsistent und vollständig ist, evolutionär weiter entwickelt, so kann das dazu führen, dass die Spezifikation inkonsistent oder unvollständig wird. Dann ist entweder ein Annahme-/Zusicherungsvertrag nicht mehr erfüllt:

$$\neg \mathrm{consistent}(\mathrm{spec}) \Leftrightarrow \exists c \in \mathrm{Contract}_{\mathrm{spec}} \cdot \neg \mathrm{fulfilled}(c, \mathrm{Specifier}_{\mathrm{spec}})$$

Oder einige angenommene Eigenschaften sind noch nicht zugesichert:

$$\neg \mathrm{complete}(\mathrm{spec}) \Leftrightarrow \mathrm{required}(\mathrm{spec}) \neq \emptyset$$

Für derartige Spezifikationen ist die Existenz einer Implementierung nicht garantiert. Der Formalismus des evolutionären Entwurfs kann und soll solche Spezifikationen nicht verhindern. Jedoch unterstützt er den Architekten bei der frühzeitigen Erkennung derartiger Situationen und hilft die Teile der Spezifikation zu identifizieren, die zu den Inkonsistenzen oder zu der Unvollständigkeit geführt haben:

Definition 3.12: Inkonsistenzen und Unvollständigkeiten in Spezifikationen

Sei $\mathrm{spec} \in \mathrm{SPECIFICATION}$ *eine inkonsistente Spezifikation eines Systems mit* $\neg \mathrm{consistent}(\mathrm{spec})$*, so entsprechen die Inkonsistenzen der Menge der nicht gültigen Annahme-/Zusicherungsverträge:*

$$\mathrm{BrokenContract}_{\mathrm{spec}} =_{\mathrm{def}} \left\{ c \in \mathrm{Contract}_{\mathrm{spec}} \,\middle|\, \neg \mathrm{fulfilled}\left(c, \mathrm{Specifier}_{\mathrm{spec}}\right) \right\}$$

Darüber hinaus können die einzelnen Spezifikationsbezeichner, die für die Inkonsistenzen in der Spezifikation verantwortlich sind identifiziert werden. Durch die Menge der Eigenschaften, die in Annahme-/Zusicherungsverträgen als zugesichert vorkommen, aber von den entsprechenden Spezifikationsbezeichner nicht zugesichert werden, lassen sich diese Spezifikationsbezeichner wie folgt ermitteln:

$$\mathrm{AssumedAssured}_{\mathrm{spec}} =_{\mathrm{def}} \left\{ (sp, r) \in \left(\mathrm{Specifier}_{\mathrm{spec}} \times \mathrm{required}(sp)\right) \,\middle|\, \exists c \in \mathrm{BrokenContract}_{\mathrm{spec}} \cdot (r, x, a) \in c \right\}$$

Sei $\mathrm{spec} \in \mathrm{SPECIFICATION}$ *eine unvollständige Spezifikation eines Systems mit* $\neg \mathrm{complete}(\mathrm{spec})$*, so entsprechen die Unvollständigkeiten in der Spezifikation der Menge der angenommenen Eigenschaften der Spezifikation:*

$$\text{Required}_{spec} =_{def} \text{required}(\text{spec})$$

Darüber hinaus können die Spezifikationsbezeichner, die für die Unvollständigkeiten in der Spezifikation verantwortlich sind, identifiziert werden. Durch die Menge der Eigenschaften, die von Spezifikationsbezeichner angenommenen werden, aber nicht zugesichert sind, lassen sich diese Spezifikationsbezeichner wie folgt ermitteln:

$$\text{NotAssured}_{spec} =_{def} \left\{(\text{sp},\text{r}) \in \left(\text{Specifier}_{spec} \times \text{required}(\text{spec})\right) \middle| \text{r} \in \text{required}(\text{sp})\right\}$$

So ermöglicht das neue, verbesserte semantische Modell komponentenbasierter Systeme die Widerspruchsfreiheit von Spezifikationen stetig zu überprüfen, auch wenn diese evolutionär weiter entwickelt werden. Sollte das Modell im Zuge eines Evolutionsschrittes inkonsistent oder unvollständig werden, so können die Inkonsistenzen und die Unvollständigkeiten in der Spezifikation identifiziert und bestimmten Spezifikationsbezeichnern und deren Eigenschaften zugeordnet werden (vgl. Frage 2 aus Kapitel 3.2).

Auch in Bezug auf die Angemessenheit einer evolutionär weiter entwickelten Spezifikation hat der neue Ansatz Vorteile. Es können jetzt nicht nur die Veränderung von ganzen Systemen beschrieben werden, sondern auch die Veränderungen der Eigenschaften einzelner Spezifikationsbezeichner und damit insbesondere auch einzelner Komponentenbeschreibungen.

Die folgenden Definitionen modellieren die Mengen von Eigenschaften einer Spezifikation und von Spezifikationsbezeichnern, die im Zuge eines Evolutionsschrittes stabil geblieben, hinzugekommen oder weggefallen sind, und nehmen somit direkt Bezug auf die Fragen 3 und 4 aus Kapitel 3.2:

Definition 3.13: Stabile Eigenschaften in Spezifikationen und Spezifikationsbezeichnern

Sei $\text{spec} \in \text{SPECIFICATION}$ *eine Spezifikation eines Systems, die weiter entwickelt wurde zu* $\text{evolve}(\text{spec})$*, so bezeichnet die Menge* $\text{StableAssured}_{spec}$ *(bzw.* $\text{StableRequired}_{spec}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die sowohl in* spec *als auch in* $\text{evolve}(\text{spec})$ *vorhanden sind:*

$$\text{StableAssured}_{spec} =_{def} \text{assured}(\text{spec}) \cap \text{assured}\left(\text{evolve}(\text{spec})\right)$$

$$\text{StableRequired}_{spec} =_{def} \text{required}(\text{spec}) \cap \text{required}\left(\text{evolve}(\text{spec})\right)$$

Für jeden Spezifikationsbezeichner $\text{sp} \in \text{Specifier}_{spec}$ *und dessen weiter entwickelter Spezifikationsbezeichner* $\text{sp}' \in \text{Specifier}_{evolve(spec)}$ *bezeichnet die Menge* $\text{StableAssured}_{sp}$ *(bzw.* $\text{StableRequired}_{sp}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die sowohl in* sp *als auch in* sp' *vorhanden sind:*

$$\text{StableAssured}_{sp} =_{def} \text{assured}(\text{sp}) \cap \text{assured}(\text{sp}')$$

$$\text{StableRequired}_{sp} =_{def} \text{required}(\text{sp}) \cap \text{required}(\text{sp}')$$

Definition 3.14: Neue Eigenschaften in Spezifikationen und Spezifikationsbezeichnern

Sei $\text{spec} \in \text{SPECIFICATION}$ *eine Spezifikation eines Systems, die weiter entwickelt wurde zu* $\text{evolve}(\text{spec})$*, so bezeichnet die Menge* $\text{AddedAssured}_{spec}$ *(bzw.* $\text{AddedRequired}_{spec}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die in* spec *nicht enthalten sind, aber in* $\text{evolve}(\text{spec})$*:*

$$\text{AddedAssured}_{spec} =_{def} \text{assured}\left(\text{evolve}(\text{spec})\right) \setminus \text{assured}(\text{spec})$$

$$\text{AddedRequired}_{spec} =_{def} \text{required}\left(\text{evolve}(\text{spec})\right) \setminus \text{required}(\text{spec})$$

Für jeden Spezifikationsbezeichner $\mathrm{sp} \in \mathrm{Specifier}_{\mathrm{spec}}$ *und dessen weiter entwickelter Spezifikationsbezeichner* $\mathrm{sp}' \in \mathrm{Specifier}_{\mathrm{evolve(spec)}}$ *bezeichnet die Menge* $\mathrm{AddedAssured}_{\mathrm{sp}}$ *(bzw.* $\mathrm{AddedRequired}_{\mathrm{sp}}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die in* sp *nicht vorhanden sind, aber in* sp'*:*

$$\mathrm{AddedAssured}_{\mathrm{sp}} =_{\mathrm{def}} \mathrm{assured}(\mathrm{sp}') \setminus \mathrm{assured}(\mathrm{sp})$$
$$\mathrm{AddedRequired}_{\mathrm{sp}} =_{\mathrm{def}} \mathrm{required}(\mathrm{sp}') \setminus \mathrm{required}(\mathrm{sp})$$

Definition 3.15: Entfernte Eigenschaften in Spezifikationen und Spezifikationsbezeichnern

Sei $\mathrm{spec} \in \mathrm{SPECIFICATION}$ *eine Spezifikation eines Systems, die weiter entwickelt wurde zu* $\mathrm{evolve(spec)}$*, so bezeichnet die Menge* $\mathrm{RemovedAssured}_{\mathrm{spec}}$ *(bzw.* $\mathrm{RemovedRequired}_{\mathrm{spec}}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die in* spec *enthalten sind, aber nicht mehr in* $\mathrm{evolve(spec)}$*:*

$$\mathrm{RemovedAssured}_{\mathrm{spec}} =_{\mathrm{def}} \mathrm{assured}(\mathrm{spec}) \setminus \mathrm{assured}(\mathrm{evolve}(\mathrm{spec}))$$
$$\mathrm{RemovedRequired}_{\mathrm{spec}} =_{\mathrm{def}} \mathrm{required}(\mathrm{spec}) \setminus \mathrm{required}(\mathrm{evolve}(\mathrm{spec}))$$

Für jeden Spezifikationsbezeichner $\mathrm{sp} \in \mathrm{Specifier}_{\mathrm{spec}}$ *und dessen weiter entwickelter Spezifikationsbezeichner* $\mathrm{sp}' \in \mathrm{Specifier}_{\mathrm{evolve(spec)}}$ *bezeichnet die Menge* $\mathrm{RemovedAssured}_{\mathrm{sp}}$ *(bzw.* $\mathrm{RemovedRequired}_{\mathrm{sp}}$*) die zugesicherten (bzw. angenommenen) Eigenschaften, die in* sp *vorhanden sind, aber nicht mehr in* sp'*:*

$$\mathrm{RemovedAssured}_{\mathrm{sp}} =_{\mathrm{def}} \mathrm{assured}(\mathrm{sp}) \setminus \mathrm{assured}(\mathrm{sp}')$$
$$\mathrm{RemovedRequired}_{\mathrm{sp}} =_{\mathrm{def}} \mathrm{required}(\mathrm{sp}) \setminus \mathrm{required}(\mathrm{sp}')$$

Beim Abstraction Refinement Model müsste der Architekt diese Eigenschaften aus den zusätzlichen Modellen ableiten (vgl. Kapitel 3.3). Mit diesem Formalismus ist es jetzt möglich, nach einem Evolutionsschritt dem Architekten die stabilen, neuen und verworfenen Eigenschaften jedes Spezifikationsbezeichners, insbesondere jeder Komponentenbeschreibung, direkt zu präsentieren. Unter der Annahme, dass eine adäquate Darstellung der Eigenschaften von Komponenten existiert[1], bietet das neue semantische Modell für komponentenbasierte Systeme dem Architekten eine wesentlich verbesserte Informationsgrundlage für die Beurteilung der Angemessenheit evolutionär weiter entwickelter Spezifikationen.

3.6 Varianten der Evolution von Spezifikationen

Softwarearchitekten verändern im Rahmen ihrer Arbeit die Architekturspezifikationen des Systems, an dem sie gerade arbeiten. Die Funktion evolve charakterisiert diese Veränderungen von Spezifikationen. Die Evolutionsschritte, die von den Entwicklern vorgenommen werden, können in bestimmte Kategorien eingeteilt werden. So ist zum Beispiel in der systemmodellbasierten formalen Semantik ein Entwicklungsschritt einer Spezifikation $\mathrm{spec} \in \mathrm{SPECIFICATION}$ genau dann eine Verfeinerung, wenn die folgende Bedingung gilt:

$$\mathrm{sem}(\mathrm{spec}) \supseteq \mathrm{sem}(\mathrm{evolve}(\mathrm{spec}))$$

In der prädikatenbasierten formalen Semantik lassen sich derartige Entwicklungsschritte nach den gleichen Gesetzmäßigkeiten beschreiben. Im folgenden definieren wir drei unterschiedliche Arten der Evolution von Spezifikationen, sowohl in der systemmodellbasierten als auch

[1] In den Kapiteln 6 und 7 werden entsprechende Beschreibungstechniken vorgestellt.

in der prädikatenbasierten Semantik. Damit lassen sich die Zusammenhänge zwischen der systemmodellbasierten und der prädikatenbasierten formalen Semantik noch deutlicher fassen.

Definition 3.16: Verfeinerung

Sei $\text{evolve}(\text{spec})$ *eine evolutionäre Veränderung einer beliebigen Spezifikation* $\text{spec} \in \text{SPECIFICATION}$*, so bezeichnen wir diese Veränderung als Verfeinerung, wenn* $\text{refinement}(\text{spec}, \text{evolve}(\text{spec}))$ *gilt. Das Prädikat* refinement *ist dabei wie folgt definiert ist:*

$$\text{refinement}(\text{spec}, \text{evolve}(\text{spec})) \Leftrightarrow_{\text{def}} \text{assured}(\text{spec}) \subseteq \text{assured}(\text{evolve}(\text{spec})) \wedge \text{required}(\text{spec}) \supseteq \text{required}(\text{evolve}(\text{spec}))$$

oder:

$$\text{refinement}(\text{spec}, \text{evolve}(\text{spec})) \Leftrightarrow_{\text{def}} \text{sem}(\text{spec}) \supseteq \text{sem}(\text{evolve}(\text{spec}))$$

Definition 3.17: Vergröberung

Sei $\text{evolve}(\text{spec})$ *eine evolutionäre Veränderung einer beliebigen Spezifikation* $\text{spec} \in \text{SPECIFICATION}$*, so bezeichnen wir diese Veränderung als Vergröberung, wenn* $\text{abstraction}(\text{spec}, \text{evolve}(\text{spec}))$ *gilt. Das Prädikat* abstraction *ist dabei wie folgt definiert:*

$$\text{abstraction}(\text{spec}, \text{evolve}(\text{spec})) \Leftrightarrow_{\text{def}} \text{assured}(\text{spec}) \supseteq \text{assured}(\text{evolve}(\text{spec})) \wedge \text{required}(\text{spec}) \subseteq \text{required}(\text{evolve}(\text{spec}))$$

oder:

$$\text{abstraction}(\text{spec}, \text{evolve}(\text{spec})) \Leftrightarrow_{\text{def}} \text{sem}(\text{spec}) \subseteq \text{sem}(\text{evolve}(\text{spec}))$$

Definition 3.18: Strikte Evolution

Sei $\text{evolve}(\text{spec})$ *eine evolutionäre Veränderung einer beliebigen Spezifikation* $\text{spec} \in \text{SPECIFICATION}$*, so bezeichnen wir diese Veränderung als strikte Evolution, wenn sie weder eine Verfeinerung noch eine Vergröberung ist. Wir schreiben:*

$$\text{evolution}(\text{spec}, \text{evolve}(\text{spec})) \Leftrightarrow_{\text{def}} \neg\text{refinement}(\text{spec}, \text{evolve}(\text{spec})) \wedge \neg\text{abstraction}(\text{spec}, \text{evolve}(\text{spec}))$$

Die Definitionen von Verfeinerung und die Vergröberung ergeben sich direkt aus den Eigenschaften des Begriffs des vollständigen Verbandes (vgl. auch Kapitel 3.3). Bei der strikten Evolution wird jedoch ein wesentlicher Unterschied zwischen der systemmodellbasierten und der prädikatenbasierten formalen Semantik nochmals deutlich. Bei der systemmodellbasierten formalen Semantik lassen sich keine weiteren Aussagen über die Menge der Systeme $\text{sem}(\text{spec})$ und $\text{sem}(\text{evolve}(\text{spec}))$ festhalten, außer dass sie weder in einer Verfeinerungsbeziehung noch in einer Vergröberungsbeziehung stehen. Der Ansatz der prädikatenbasierten formalen Semantik liefert uns zusätzlich noch ein Abstandsmaß für die Größe des Evolutionsschrittes: Die Anzahl der Eigenschaften, die im Zuge eines Evolutionsschrittes stabil geblieben, hinzugekommen oder weggefallen sind (vgl. Kapitel 3.5).

Während des Architekturentwurfs entwickeln Softwarearchitekten die Spezifikation der Softwarearchitektur beständig weiter. Meist werden dabei einige Eigenschaften verworfen, neue Eigenschaften hinzugefügt und andere Eigenschaften unverändert übernommen. Folglich entsprechen die typischen Evolutionsschritte, die Designer an den Entwicklungsdokumenten durchführen, meist der sogenannten strikten Evolution.

Deshalb setzen wir uns in der restlichen Arbeit hauptsächlich mit der strikten Evolution auseinander. Dabei werden wir die Bezeichnungen Evolution und strikte Evolution als Synonyme

verwenden, außer wenn der Kontext eine Unterscheidung zwischen den unterschiedlichen Ausprägungen der Evolutionsschritte erfordert.

3.7 Zusammenfassung

Softwarearchitekturen werden schrittweise und evolutionär entwickelt. Nach jedem Evolutionsschritt stellt sich die Frage nach der Widerspruchsfreiheit und der Angemessenheit des weiter entwickelten Modells. Bestehende formale Methoden weisen Defizite bezüglich dieser Fragestellungen auf, insbesondere im Kontext der evolutionären Entwicklung komponentenbasierter Systeme. So kann beispielsweise die Frage, welche Veränderungen an Komponenten zu einem inkonsistenten Gesamtsystem geführt haben, nicht zufriedenstellend beantwortet werden.

Deshalb haben wir in diesem Kapitel ein neuartiges, weiter entwickeltes semantisches Modell für komponentenbasierte Systeme entwickelt. Dieser Formalismus ermöglicht es, die Eigenschaften von Komponenten und Systemen aus Komponenten präzise und differenziert zu beschreiben. Die Unterteilung in angenommene und zugesicherte Eigenschaften in Verbindung mit Annahme-/Zusicherungsverträgen erlaubt es, die Verantwortlichkeiten und Abhängigkeiten zwischen den Komponenten präzise und vollständig zu modellieren. Entscheidend dabei ist, dass die Abhängigkeiten zwischen den Komponenten nicht nur auf syntaktischer Ebene spezifiziert und überprüft werden können, sondern auch auf semantischer Ebene.

In Tabelle 3.1 sind nochmals die zentralen Konzepte und Definitionen des erarbeiteten Formalismus für den evolutionären Architekturentwurf komponentenbasierter Systeme zusammengefasst. Auf die Wiederholung der formalen Definitionen und der Zusammenhänge verzichten wir hier.

$\mathsf{SYSTEM} \subseteq \mathsf{INSTANCE}$ $\mathsf{SPECIFIER}$ $\mathsf{CONTRACT} = \mathsf{TERM}^{\mathsf{v}} \times \mathsf{SPECIFIER} \times \mathsf{TERM}^{\mathsf{v}}$ $\mathsf{TERM}^{\mathsf{v}}$	Menge von Systemen, Teilmenge der Instanzen Menge der Spezifikationsbezeichner Menge der Annahme-/Zusicherungsverträge Menge der logischen Ausdrücke der ersten Stufe mit einer freien Variablen v
$\mathsf{specified} : \mathsf{INSTANCE} \rightarrow \mathsf{SPECIFIER}$ $\mathsf{fulfilled} : \mathsf{CONTRACT} \times \mathrm{P}(\mathsf{SPECIFIER}) \rightarrow \mathsf{BOOLEAN}$	ordnet einer Instanz ihren Spezifikationsbezeichner zu berechnet, ob ein Annahme-/Zusicherungsvertrag erfüllt ist
$\mathsf{spec} \in \mathsf{SPECIFICATION} = \mathrm{P}(\mathsf{DOCUMENT})$ $\mathsf{Specifier}_{\mathsf{spec}} \subseteq \mathsf{SPECIFIER}$ $\mathsf{Contract}_{\mathsf{spec}} \subseteq \mathsf{CONTRACT}$ $\mathsf{contract_of}_{\mathsf{spec}} : \mathsf{Specifier}_{\mathsf{spec}} \rightarrow \mathsf{Contract}_{\mathsf{spec}}$	eine Spezifikation spec Menge der Spezifikationsbezeichner in spec Menge der Annahme-/Zusicherungsverträge in spec ordnet jedem Spezifikationsbezeichner in spec den zugehörigen Annahme-/Zusicherungsvertrag zu
$\mathsf{assured} : \mathsf{SPECIFICATION} \rightarrow \mathrm{P}\left(\mathsf{TERM}^{\mathsf{v}}\right)$ $\mathsf{required} : \mathsf{SPECIFICATION} \rightarrow \mathrm{P}\left(\mathsf{TERM}^{\mathsf{v}}\right)$	ordnet jeder Spezifikation die zugesicherten Eigenschaften zu ordnet jeder Spezifikation die angenommenen Eigenschaften zu
$\mathsf{evolve} : \mathrm{P}(\mathsf{SPECIFICATION}) \rightarrow \mathrm{P}(\mathsf{SPECIFICATION})$	beschreibt einen Evolutionsschritt einer Spezifikation
$\mathsf{consistent}(\mathsf{spec})$ $\mathsf{complete}(\mathsf{spec})$	Konsistenz einer Spezifikation spec Vollständigkeit einer Spezifikation spec
$\mathsf{sem}(\mathsf{spec}) = \left\{ \mathsf{sys} \in \mathsf{SYSTEM} \middle\vert \forall \mathsf{a} \in \mathsf{assured}(\mathsf{spec}) \Rightarrow \mathsf{a}\left[\mathsf{sys}\right] \right\}$	Menge der Systeme, die eine konsistente und vollständige Spezifikation spec implementieren

Tabelle 3.1: Formale Konzeption des evolutionären Entwurfs – Zusammenfassung

4 Der Pausenplaner – Ein einfaches Anwendungsbeispiel

In den vorhergehenden Kapiteln wurde das Vorgehensmodell und die Methodik des evolutionären Architekturentwurfs vorgestellt. Für die konsequente Umsetzung sind ein komponentenbasiertes Systemmodell und spezifische Beschreibungstechniken erforderlich. Dabei charakterisiert das Systemmodell die Laufzeitumgebung für komponentenbasierte Systeme und deren Architekturen. Mit Hilfe von Beschreibungstechniken können Spezifikationen komponentenbasierter Systeme erstellt und in dieser Laufzeitumgebung ausgeführt werden. Die prädikatenbasierte formale Fundierung der Beschreibungstechniken liefert eine präzise Vorschrift, wie die Spezifikation eines Systems in der Laufzeitumgebung zu interpretieren ist.

Das Systemmodell, die Beschreibungstechniken und die prädikatenbasierte formale Fundierung sind Gegenstand der kommenden Kapitel. Diese Themen sind teilweise von theoretischer und abstrakter Natur. Damit wir den Bezug zur Praxis nicht verlieren, werden wir die theoretischen Inhalte anhand eines konkreten Anwendungsbeispieles illustrieren.

Dieses Anwendungsbeispiel – den Pausenplaner – führen wir in diesem Kapitel ein. Der Pausenplaner ist ein kleines verteiltes System für die Erstellung und Bearbeitung von Pausenaufsichtsplänen in Schulen. Der Funktionalitätsumfang des Pausenplaners ist relativ klein und klar abgegrenzt. Aufwendige und komplexe Algorithmen sind nicht enthalten. Trotzdem werden viele Problemstellungen sichtbar, die auch bei größeren betrieblichen Informationssystemen auftreten, wie zum Beispiel Verteilung, Persistenz, Konsistenz oder Transaktionsverhalten.

Am Anfang dieses Kapitels stellen wir kurz die Historie des Pausenplaners vor. Es folgt eine Beschreibung der initialen Anforderungen und die Systemvision des Kunden. Diese beinhalten die Anwendungsszenarien und weitergehende nichtfunktionale Anforderungen. Daran schließt sich eine gründlichere Analyse der Anwendungsfälle an. So können wir zum Schluss eine erste Aufteilung des Pausenplaners in fachliche Komponenten motivieren und präsentieren.

4.1 Motivation und Hintergrund des Pausenplaners

Die Aufgabenstellung, den Pausenplaner zu entwickeln, stammt ursprünglich von der oo D.A.CH. [DACH01a]. Die oo D.A.CH. ist das Kürzel für ein Treffen von Forscherinnen und Forschern aus den Ländern Deutschland, Österreich und der Schweiz, die sich mit objektorientierten Fragestellungen auseinandersetzen.

Unter anderem wollte sich die Gruppe intensiver mit der Evaluierung von Programmierparadigmen, Werkzeugen, Frameworks und Komponententechnologien beschäftigen. Zu diesem Zweck wurde unter den Teilnehmern eines oo D.A.CH. Treffens die Entwicklung des Pausenplaners als Aufgabenstellung verteilt. In einem nächsten Treffen sollten die Teilnehmer ihre erarbeiteten Lösungen vorstellen. Ziel war es, auf dieser Basis dann die angesprochenen Themen eingehender zu diskutieren und zu untersuchen (siehe auch [DACH01b]). Die verschiedenen Versionen der Aufgabenstellung sind unter [DACH01c] zu finden.

Die Aufgabenstellung ist klar abgegrenzt, gut beschrieben und enthält eine Reihe interessanter Aspekte, wie zum Beispiel Verteilung und Persistenz. Deshalb eignet sich der Pausenplaner neben den von der oo D.A.CH. anvisierten Themen auch ausgezeichnet für die Evaluierung anderer Bereiche der Softwaretechnik, wie zum Beispiel Beschreibungstechniken und Systemmodelle.

So diente der Pausenplaner bereits als Grundlage für eine umfassende Fallstudie im Teilprojekt A1 des Forschungsverbundes FORSOFT [FORS01]. Gegenstand dieser Studie war die durchgehende Modellierung, von der Analyse bis zur Implementierung, einer verteilten, in Java realisierten, Anwendung mit der UML [BRS97].

Im Rahmen der vorliegenden Arbeit können wir auf diese und andere Ergebnisse rund um den Pausenplaner nur begrenzt zurückgreifen. Die existierenden Arbeiten im Kontext des Pausenplaners sind entweder stark technologischer Natur, wie zum Beispiel die Arbeiten der oo D.A.CH. [DACH01b], oder sie betrachten Aspekte der objektorientierten Modellierung, wie zum Beispiel in [BRS97]. Eine durchgängige komponentenbasierte Modellierung und Entwicklung des Pausenplaners existiert noch nicht.

Nicht zuletzt auf Grund der gut dokumentierten Aufgabenstellung, ist der Pausenplaner trotzdem ein geeignetes Fallbeispiel für eine praxisnahe Diskussion der von uns erarbeiteten Konzepte und Techniken. Darüber hinaus kann durch die gemeinsame Aufgabenstellung sehr leicht der Bezug zu anderen Arbeiten und Konzepten im Kontext des Pausenplaners hergestellt werden. Beispielsweise lässt sich anhand [BRS97] der Bezug zwischen dem evolutionären Architekturentwurf und der objektorientierten Modellierung darstellen.

4.2 Erste Kundenanforderungen und Systemvision

Im Rahmen der vorliegenden Arbeit greifen wir auf die ursprüngliche Aufgabenstellung des Pausenplaners zurück (vgl. [DACH01c]). In Anlehnung an diese Beschreibung der Kundenanforderungen ist das Anwendungsszenario wie folgt:

Während der Pausen müssen die Lehrer die Schüler an verschiedenen Plätzen in der Schule beaufsichtigen. Die Zuweisung von Lehrer zu Pausenaufsichten wird in sogenannten Pausenplänen für die entsprechenden Pausenorte festgehalten. Jede Pause muss von einem Lehrer überwacht werden. Die Anzahl der Pausen, die ein Lehrer zu beaufsichtigen hat, ist abhängig von seiner Arbeitsstelle. Ein Vollzeit Lehrer muss mehr Pausen beaufsichtigen wie ein Leh-

rer, der nur ein paar Stunden pro Woche Unterricht gibt. Lehrer können bestimmte Zeiten festlegen, zu denen sie keine Pause beaufsichtigen wollen.

Pausenpläne werden von dem Pausenaufsichtsplaner erstellt, beispielsweise der Rektor der Schule. Der Pausenplaner soll den Pausenaufsichtsplaner bei seiner Arbeit unterstützten. Mit Hilfe des Pausenplaners kann der Pausenaufsichtsplaner Pausenpläne erzeugen, löschen und verändern, den Lehrkörper und die Lehrer einer Schule verwalten sowie Lehrern Pausenaufsichten zuweisen. Darüber hinaus soll der Pausenplaner stets eine aktuelle Pausenstatistik anzeigen. Diese beinhaltet zum Beispiel die Anzahl der Pausen, die ein Lehrer zu beaufsichtigen hat, oder die Anzahl der Pausen, die noch unbeaufsichtigt sind.

Der Screenshot in Abbildung 4.1 ist direkt aus der ursprünglichen Aufgabenstellung der oo D.A.CH. übernommen (vgl. [DACH01d]). Er illustriert die Kundenvision der Oberfläche des Pausenplaners.

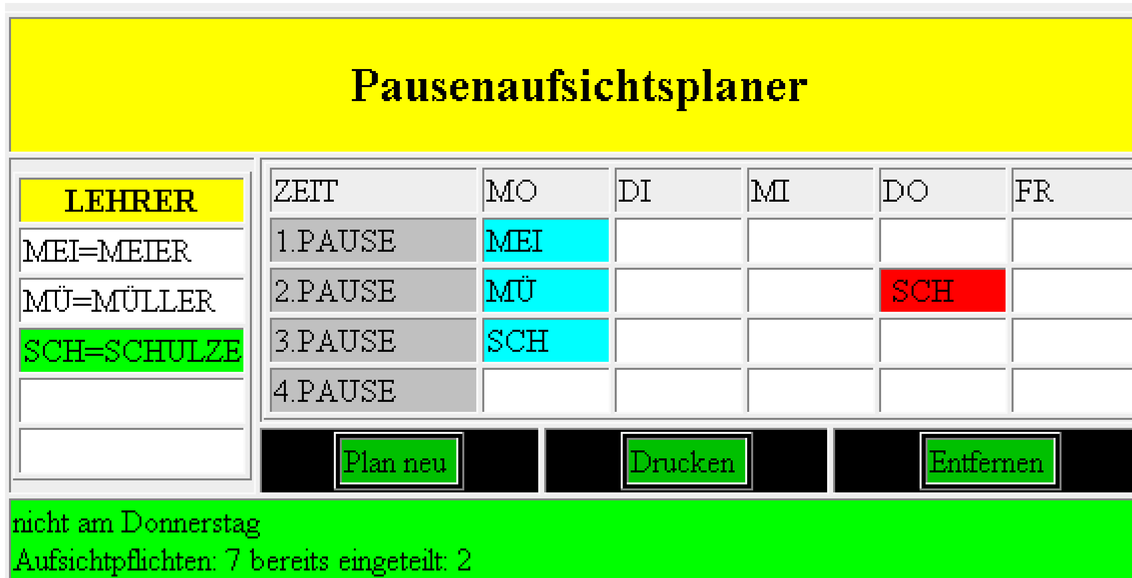

Abbildung 4.1: Systemvision des Pausenplaners (aus [DACH01d])

Links im Werkzeug werden die Lehrer angezeigt. Diese können mittels Drag & Drop auf die einzelnen Felder des Pausenplans rechts gezogen werden. Wird eine Lehrer markiert, wird in der Statuszeile angezeigt, welche Pausen nicht von dem Lehrer beaufsichtigt werden können. Das Ablegen des Lehrers auf eine solche Pause ist dennoch möglich. Entstandene Konflikte werden farblich hervorgehoben. Bereits vorhandene Belegungen können durch Verschieben des Lehrers auf eine andere Pause geändert werden. Soll eine Belegung auf dem Aufsichtsplan aufgehoben werden, wird das entsprechende Feld selektiert und der Knopf „Entfernen" betätigt. Durch Aktivieren des Knopfes „Drucken" wird der Pausenplan ausgedruckt. Der Knopf „Plan neu" dient zum kompletten Löschen des Aufsichtsplanes. In der zweiten Statuszeile werden für den gerade markierten Lehrer die Aufsichtspflichten und die Anzahl der bereits eingeteilten Pausen angezeigt.

Neben diesen funktionalen sind in der Aufgabenstellung einige nichtfunktionale Anforderungen enthalten, die für die weitere Bearbeitung bedeutsam sind (vgl. [DACH01b]):

- Verteilung und verteilte Benutzung:
  Der Pausenplaner sollte über das Internet die verteilte und parallele Benutzung ermöglichen. Der Pausenaufsichtsplaner kann Pausenpläne bearbeiten, während gleichzeitig Leh-

rer ihre Pausenpläne betrachten. Aktuelle Änderungen am Lehrkörper oder an Pausenplänen werden allen Benutzer sofort zugänglich gemacht.

- Persistente Datenhaltung:
  Der Lehrkörper und die Pausenpläne sind persistent abzulegen. Der Pausenplaner ermöglicht das Verwalten des Lehrkörpers und der Pausenpläne. Entsprechende Auswahlmasken und Bearbeitungsmasken werden zur Verfügung gestellt.

Eine derartige, vom Kunden erstellte, erste Beschreibung der Anforderungen an eine neues System ist auch für industrielle Projekte realistisch. Häufig ist die Anforderungsbeschreibung nicht eindeutig, vollständig oder fehlerfrei. Die vorliegende Beschreibung lässt beispielsweise die Art und Weise wie der Lehrkörper der Schule verwaltet wird offen. Außerdem ist die Aufgabenstellung an einigen stellen nicht ganz korrekt. Es gibt keine Schule, die nur an einem Ort Pausen zu beaufsichtigen hat. Der Pausenplaner muss mehrere Pausenpläne verwalten und bearbeiten können. Diese und andere Korrekturen an der Aufgabenstellung wurden in einer neueren Version vorgenommen und stehen unter [DACH01e] zur Verfügung.

4.3 Analyse der Anwendungsfälle des Pausenplaners

Ausgangspunkt des evolutionären Architekturentwurfs ist eine mehr oder weniger fein ausgearbeitete Anforderungsanalyse (vgl. Kapitel 2). Das zentrale Ergebnis der Anforderungsanalyse ist im allgemeinen eine Liste von Anforderungen, die beliebig detailliert beschrieben sein können.

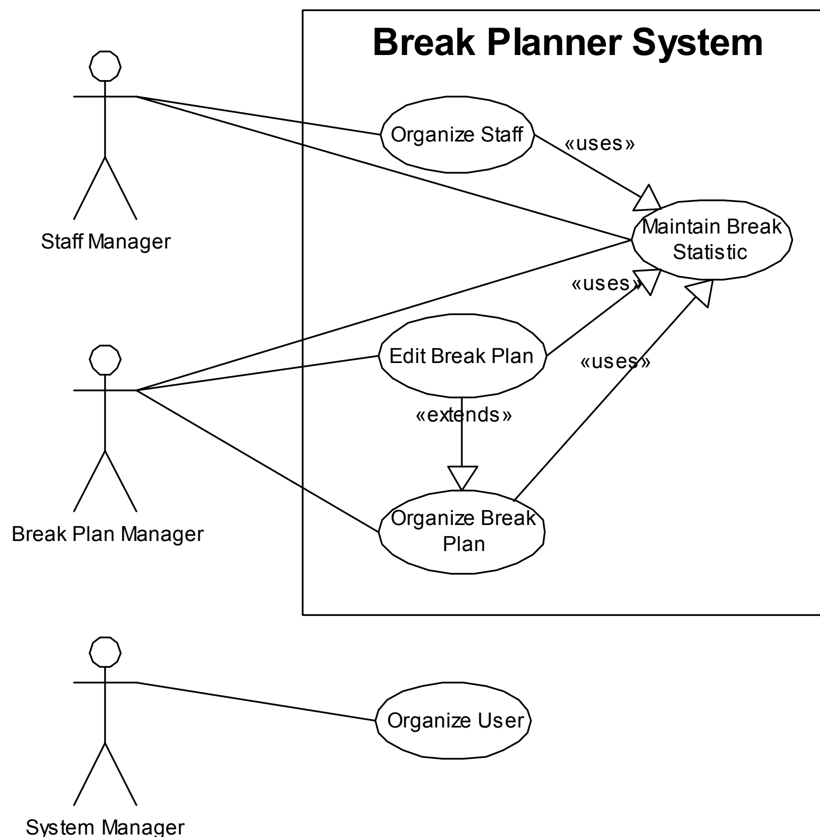

Abbildung 4.2: UML Anwendungsfalldiagramm des Pausenplaners aus [BRS97]

Der Pausenplaner wurde bereits in mehreren Fallstudien verwendet. Die Anwendungsfallanalyse aus [BRS97] ist ein optimaler Ausgangspunkt für eine weiterführende komponentenba-

sierte Architekturmodellierung des Pausenplaners. Abbildung 4.2 zeigt das UML Anwendungsfalldiagramm aus [BRS97].

Der zentrale Anwendungsfall ist `Edit Break Plan`. Er beinhaltet die Funktionalität Lehrern Pausenaufsichten zuzuweisen. Dieser Anwendungsfall ist eine Erweiterung von `Organize Break Plan`, der die Verwaltung von Pausenplänen umfasst, wie zum Beispiel das Erzeugen von neuen, leeren Pausenplänen, das Definieren der Pausenzeiten innerhalb von Pausenplänen und das Löschen oder Drucken von Pausenplänen.

`Organize Staff` entspricht dem Anwendungsfall `Organize Break Plan.` Anstelle der Verwaltung von Pausenplänen beinhaltet `Organize Staff` die Funktionalität für das Verwalten des Lehrkörpers mit den zugehörigen Lehren.

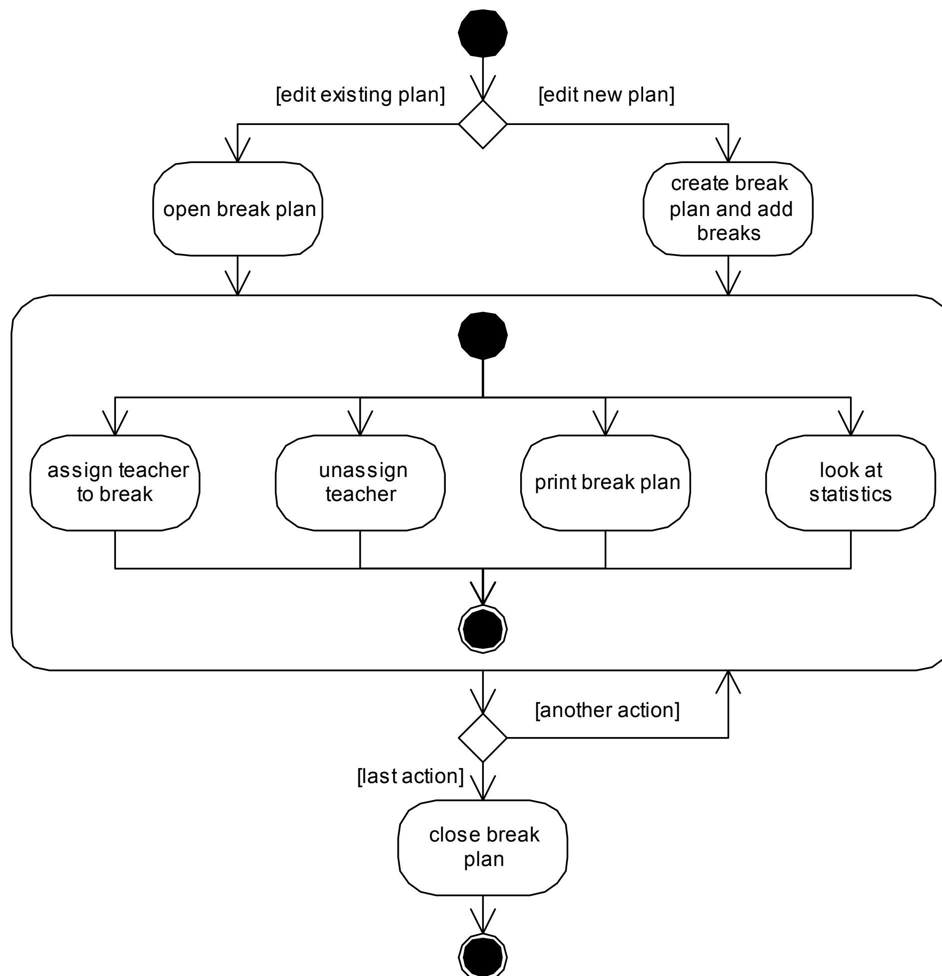

Abbildung 4.3: UML Aktivitätsdiagramm des Pausenplaners aus [BRS97]

Der Anwendungsfall `Maintain Break Statistic` berechnet die Werte der Pausenplanstatistik. Er wird von allen anderen Anwendungsfällen aus verwendet. Die Pausenplanstatistik umfasst beispielsweise die Soll- und Ist-Anzahl der Pausenaufsichten eines Lehrers und die Anzahl der belegten, unbelegten sowie konfliktbehafteten Pausen.

Entsprechend den Kundenanforderungen soll der Pausenplaner über das Internet verwendet werden können. Anwendungen, die über das Internet zugreifbar sind, unterliegen besonderen Sicherheitsanforderungen, wie zum Beispiel das anmelden am System über Benutzername und Passwort. Der Anwendungsfall `Organize User` dient der Benutzerverwaltung des Systems. Dieser Anwendungsfall wird in unserem Beispiel nicht weiter betrachtet. Er kann au-

ßerhalb des Systems umgesetzt werden, zum Beispiel durch die Mechanismen des WWW Servers über den der Pausenplaner angesprochen wird.

`Edit Break Plan` ist einer der zentralen Anwendungsfälle des Pausenplaners. Die initialen Kundenanforderungen in Kapitel 4.2 enthielten bereits eine erste Beschreibung des Ablaufs dieses Anwendungsfalles. Abbildung 4.3 illustriert diesen Ablauf nochmals grafisch als UML Aktivitätsdiagramm: In dem beschriebenen Anwendungsfall kann der Benutzer zu Beginn entweder einen existierenden Pausenplan öffnen oder einen neuen leeren Pausenplan anlegen. Mit dem ausgewählten Pausenplan führt der Benutzer eine beliebige Folge von Aktionen aus. Er kann Lehrer für Pausenaufsichten einteilen oder bestehende Pausenaufsichten entfernen. Dabei verändert sich die Pausenplanstatistik. Außerdem kann der Benutzer den Pausenplan ausdrucken, wann immer er es für notwendig erachtet. Mit dem Schließen des bearbeiteten Pausenplans ist der Anwendungsfall beendet.

In Verbindung mit der verteilten und parallelen Bearbeitung beinhaltet der Anwendungsfall `Maintain Break Statistic` einige besonders interessante Aspekte: Laut Kundenanforderung steht jedem Anwender stets eine aktuelle Pausenplanstatistik zur Verfügung. Die Pausenplanstatistik ändert sich aber, wenn Daten des Pausenplaners verändert werden.

Wird ein Lehrer einer Pausenaufsicht zugewiesen, so ändert sich die Pausenplanstatistik. Die Anzahl der unbelegten Pausen verringert sich und die Anzahl der konfliktbehafteten wird unter Umständen größer. Wird ein neuer Lehrer in den Lehrkörper aufgenommen, so hat dies ebenfalls Auswirkungen auf die Pausenplanstatistik. Die Anzahl der zu beaufsichtigenden Pausen pro Lehrer wird geringer.

Zusammenfassend kann man festhalten, wenn sich die persistenten Daten des Pausenplaners verändern, wie zum Beispiel Lehrer, Lehrkörper, Pausenpläne oder die Pausenaufsichtszuordnung, dann ist die Pausenplanstatistik davon betroffen. Der Pausenplaner unterstützt aber mehrere Bearbeiter gleichzeitig. Führt einer der Bearbeiter eine Operation durch, welche die Pausenplanstatistik verändert, dann muss nicht nur seine Pausenplanstatistik sondern auch die Pausenplanstatistik der anderen Benutzer aktualisiert werden. Prinzipiell gibt es drei Möglichkeiten dieses Verhaltens zu realisieren:

- Aktualisierung auf Anfrage: Dem Benutzer steht ein spezieller Knopf in der Oberfläche zur Verfügung. Wird dieser Knopf betätigt, so wird eine neue Pausenplanstatistik berechnet und angezeigt.
- Aktualisierung in Intervallen: Die Pausenplanstatistik wird automatisch in bestimmten Zeitintervallen aktualisiert und angezeigt.
- Aktualisierung bei Änderung: Wenn relevante Daten verändert werden, dann wird die Pausenplanstatistik sofort aktualisiert.

Offensichtlich ist die dritte Variante die benutzerfreundlichste und angenehmste. Es wird automatisch die aktuelle Pausenplanstatistik angezeigt, ohne dass der Benutzer zusätzlich mit dem System interagieren muss. Deshalb wird in unserem Anwendungsbeispiel die Strategie „Aktualisierung bei Änderung“ umgesetzt.

4.4 Identifikation der Komponenten des Pausenplaners

Zu Begin des Entwurfs einer komponentenbasierten Softwarearchitektur stellt sich die Frage der Identifikation der Komponenten, die Bestandteil des Systems sein werden (vgl. Kapitel 2). Die Identifizierung der Komponenten ist ein wichtiges Thema beim Architekturentwurf, steht

aber nicht im Zentrum dieser Arbeit. In [Ambl98,Ambl99,BRSV98a,BRSV98b] sind einige Prozessmuster für die Durchführung dieser Aktivität beschrieben: Zum Beispiel das Muster „Komponentengrenzen anhand der Funktionalität" oder „Einheit der Wiederverwendung bestimmt Komponentengrenzen".

Beim ersten Muster wird die Grenze einer Komponente durch einen wohl definierten Funktionalitätsbereich bestimmt. Das zweite Muster besagt, dass man Komponenten bestimmen kann, indem man die Teile eines Systems identifiziert, die wiederverwendbar sind. Wiederverwendbar in diesem Sinne sind Komponenten die bereits existieren oder die zumindest das Potential für eine spätere Wiederverwendung besitzen.

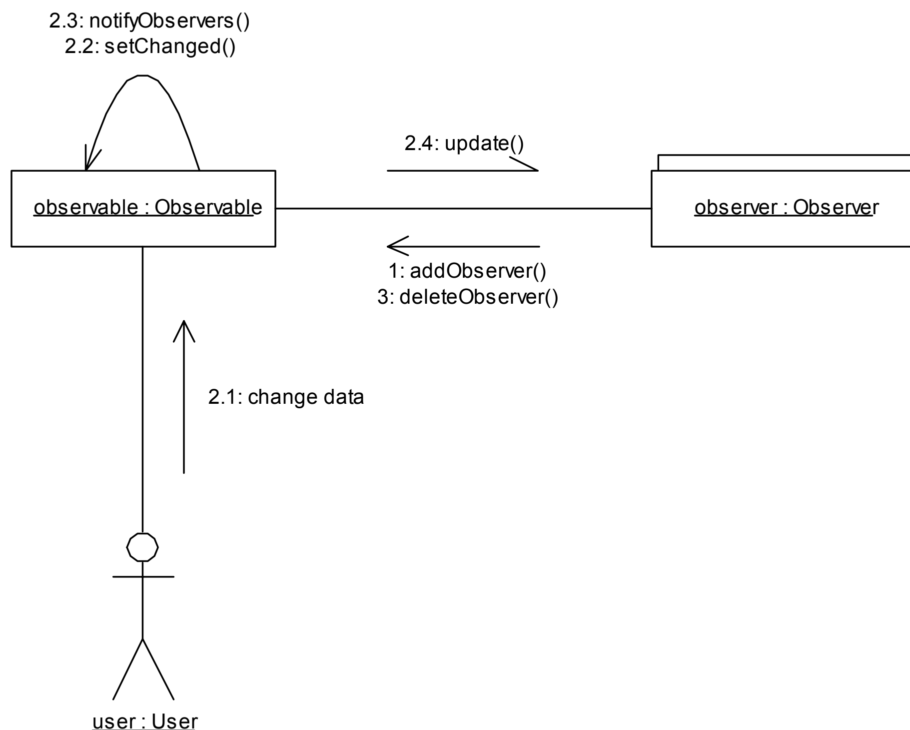

Abbildung 4.4: UML Kollaborationsdiagramm des Observer Patterns

Für die Identifikation der Komponenten des Pausenplaners wenden wir aber der Einfachheit nur das erste Prozessmuster an. Die Komponenten werden im wesentlichen durch die zentralen Funktionalitätsbereiche des Systems bestimmt. Für den Pausenplaner lassen sich anhand der Anwendungsfälle drei grobe Funktionalitätsbereiche bestimmen:

- `StaffEditor`
  Die Anwendungsfälle `Edit Break Plan` und `Organize Break Plan` werden von dieser Komponente realisiert. Der `Break Plan Manager` kann mit dieser Komponente die Pausenpläne verwalten und Lehrern Aufsichten zuweisen.
- `BreakPlanEditor`
  Diese Komponente unterstützt den Anwendungsfall `Organize Staff`. Mit dieser Komponente kann der `Staff Manager` den Lehrkörper und die zugehörigen Lehrer verwalten.
- `BreakPlanStatisticView`
  Der verbleibende Anwendungsfall `Maintain Break Statistic` wird von dieser Komponente umgesetzt. Diese Komponente stellt den Benutzern die Pausenplanstatistik zur Verfügung.

Alle Anwendungsfälle sind Komponenten zugewiesen. Nur die Anforderung, dass allen Benutzer immer die aktuelle Pausenstatistik über die „Aktualisierung bei Änderung“ Strategie bereitgestellt wird, wurde noch nicht berücksichtigt. Das Observer Pattern liefert eine Lösungen, die sich für diese Problemstellung bewährt hat [GHJV95].

Dabei werden gemeinsam benutzte Daten bzw. Objekte als `Observable` bezeichnet. Sogenannte `Observer` können sich bei den `Observable` registrieren und so ihr Interesse an Änderungen des Zustandes bekunden. Ändert sich der Zustand des `Observable`, benachrichtigt er automatisch alle registrierten `Observer` über den Aufruf der Methode `update()`. Abbildung 4.4 illustriert das Verhalten des Observer Patterns grafisch als UML Kollaborationsdiagramm.

Wenden wir dieses Pattern im Kontext des Pausenplaners an, dann sind alle Daten, die persistent gespeichert werden müssen `Observable`, wie zum Beispiel Lehrer, Lehrkörper, Pausenpläne oder die Pausenaufsichtszuordnung. Die Oberfläche des Anwenders und insbesondere die Darstellung der Pausenstatistik übernehmen den Part des `Observers`.

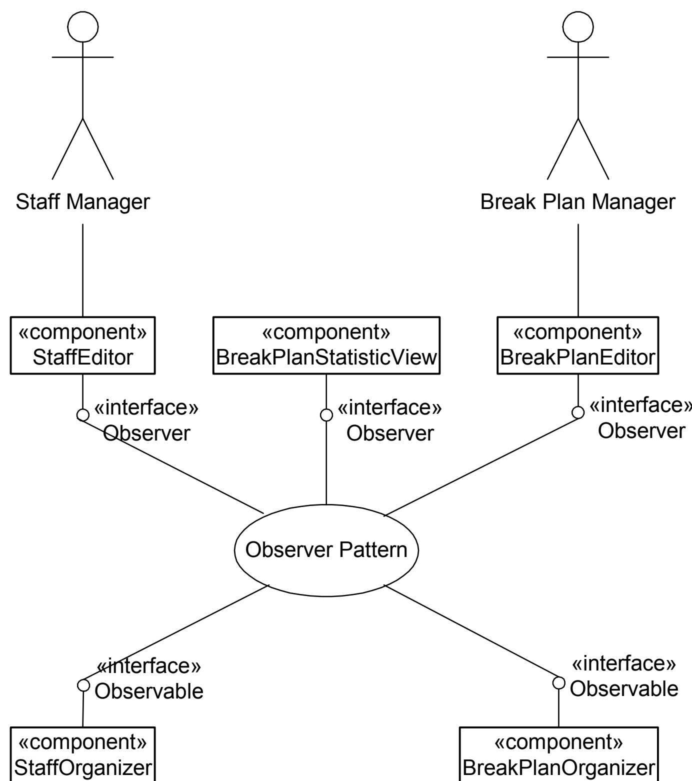

Abbildung 4.5: Komponenten des Pausenplaners

Mit Hilfe des Observer Patterns wird die verbleibende Anforderung realisiert, die „Aktualisierung bei Änderung“ Strategie. Die bereits identifizierten Komponenten müssen anhand des Observer Patterns in `Observable` und `Observer` aufgeteilt werden. Bei den Komponenten `StaffEditor` und `BreakPlanEditor` isolieren wir jeweils den `Observable` Teil in den Komponenten `StaffOrganizer` und `BreakPlanOrganizer`. `StaffEditor`, `BreakPlanEditor` und `BreakPlanStatisticView` übernehmen die Rolle des `Observer`. Abbildung 4.5 stellt diese Strukturierung der Komponenten mit den beschriebenen Verantwortlichkeiten grafisch als UML Klassendiagramm dar.

4.5 Zusammenfassung

Für die optimale Umsetzung des evolutionären Architekturentwurfs sind spezifische Beschreibungstechniken notwendig, die durch ein Systemmodell und eine prädikatenbasierte formale Semantik fundiert werden. Für die anschauliche Erarbeitung und Einführung dieser komplexen Konzepte und Zusammenhänge in den kommenden Kapiteln haben wir als Anwendungsbeispiel den Pausenplaner eingeführt.

Der Pausenplaner ist ein verteiltes, Internet-basiertes System, das in nebenläufiger und paralleler Bearbeitung die Planung der Pausenaufsichten von Schulen unterstützt. Die Anforderungen an den Pausenplaner aus Kundensicht wurden in Form einer ersten Beschreibung veranschaulicht. Auf Basis einer gründlicheren Analyse der Anwendungsfälle erfolgte eine erste Identifikation der Komponenten des Pausenplaners.

Die Komponenten `StaffEditor`, `BreakPlanEditor` und `BreakPlanStatisticView` realisieren die Benutzerschnittstelle. Der `StaffEditor` dient zur Verwaltung des Lehrkörpers, der `BreakPlanEditor` zur Verwaltung der Pausenpläne. Die Aufgabe der Komponente `BreakPlanStatisticView` ist es, den Benutzern die aktuelle Pausenstatistik zu präsentieren. Die Komponenten `StaffOrganizer` und `BreakPlanOrganizer` übernehmen die dauerhafte Speicherung und Bereitstellung der entsprechenden Daten.

Durch das Observer Pattern werden Änderungen eines Benutzers anderen Benutzern sofort angezeigt. `StaffOrganizer` und `BreakPlanOrganizer` sind `Observable`, die von den `Observer` `StaffEditor`, `BreakPlanEditor` und `BreakPlanStatisticView` beobachtet werden.

Obwohl der Pausenplaner ein relativ kleines und klar umrissenes Beispiel ist, beinhaltet er trotzdem die wesentlichen Fragestellen, die bei der Entwicklung von großen Informationssystemen anstehen, wie zum Beispiel persistente Datenhaltung, Verteilung, parallele Benutzung über das Internet oder Transaktionsmanagement. Die Entwicklung einer präzisen Spezifikation der Softwarearchitektur des Pausenplaners wird uns als ideales, praxisnahes Anschauungsobjekt in den kommenden Kapiteln dienen. Dabei werden wir die vollständige Spezifikation des Pausenplaners evolutionär entwickeln. Somit können wir die spezifischen Bedürfnisse des evolutionären Architekturentwurfs anschaulich darstellen.

5 Grundlagen komponentenbasierter Systeme

Das zentrales Ziel dieser Arbeit ist es, die evolutionäre Modellierung von Softwarearchitekturen komponentenbasierter Systeme weitreichend und durchgängig zu unterstützen. Für die Generierung eines ersten Prototypen oder eines lauffähigen Systems aus dem Architekturmodell ist eine Zielumgebung notwendig, in der das generierte System ausgeführt wird.

In der Praxis sind gegenwärtig drei Modelle für komponentenbasierte Systeme vorherrschend: COM+ [Isem00], CORBA Components [OMG99] und EJB [SUN01]. Sie bestehen jeweils aus einem Komponentenmodell und einer Laufzeitumgebung. Diese stellen einen ersten Schritt in Richtung eines methodischen Entwurfs von Architekturen dar, sind jedoch stark geprägt von den derzeit am Markt vorherrschenden Programmiersprachen, Systemplattformen und Technologien. Sie tragen den eigentlichen, eher konzeptionellen Erfordernissen im Entwurf und in der Analyse nur bedingt Rechnung.

Deshalb entwickeln wir in dieser Arbeit ein neuartiges eigenständiges Systemmodell und greifen nicht auf eines der existierenden Komponentenmodelle zurück. Dieses Systemmodell umfasst eine formale, mathematische Spezifikation der Laufzeitumgebung und des Komponentenmodells. Das entwickelte Systemmodell ist eine Abstraktion der existierenden Zielumgebungen COM+, CORBA Components und EJB. Eine Implementierung des Systemmodells auf Basis dieser existierenden Technologien ist deshalb relativ einfach möglich (vgl. Kapitel 8).

Anhand des Pausenplaners veranschaulichen wir zuerst im Kapitel 5.1 die wesentlichen Eigenschaften eines komponentenbasierten Systems. Im Kapitel 5.2 werden dann die grundlegenden Konzepte eines komponentenbasierten Systems zur Laufzeit definiert. Dies erlaubt es uns, den Zustand eines komponentenbasierten Systems zu einem bestimmten Zeitpunkt präzise zu beschreiben.

Während der Laufzeit verändern sich wesentliche Bestandteile eines Systems. Die Struktur ändert sich, Attribute werden mit neuen Werten belegt und Nachrichten werden verschickt. Damit wir diese Veränderungen beschreiben können, führen wir im Kapitel 5.3 den Begriff der Zeit und Veränderungen über die Zeit ein.

Mit der in Kapitel 5.4 eingeführten Verhaltensbeschreibung von Komponenten ist es dann möglich ein konstruktives Verfahren anzugeben, um aus dem aktuellen Systemzustand und den Verhaltensbeschreibungen der Komponenten den nächsten Systemzustand zu berechnen.

Komponentenbasierte Systeme sind selbst wieder Komponenten, die in anderen Systemen verwendet werden. Deshalb erweitern wir in Kapitel 5.5 unser Systemmodell um die notwendigen Konzepte. So können wir Softwarearchitekturen komponentenbasierter hierarchischer Systeme vollständig charakterisieren. Außerdem steht uns damit eine entsprechende Spezifikation einer Laufzeitumgebung zur Verfügung, um diese Systeme ausführen zu können.

5.1 Beispielablauf in einem komponentenbasierten System

In der Literatur existiert noch keine allgemein anerkannte Definition des Begriffs einer Komponente oder eines komponentenbasierten Systems (vgl. Kapitel 1). Durch die Analyse eines exemplarischen Benutzerszenarios des Pausenplaners aus Kapitel 4 kann ein erstes, informelles Verständnis für die grundlegenden Konzepte komponentenbasierter System erreicht werden.

Angenommen, der Pausenplaner wurde an einer Schule neu installiert. Bevor die Pausenaufsichtspläne bearbeitet werden können, müssen der Lehrkörper und die leeren Pausenpläne im System angelegt werden. Herr Maier ist verantwortlich für die Verwaltung des Lehrkörpers im Pausenplaner. Die Verwaltung der Pausenpläne ist Herrn Huber zugeteilt worden. Beide starten den Pausenplaner an ihrem lokalen Arbeitsplatz. Herr Maier verwendet den `StaffEditor` um den Lehrkörper anzulegen. Herr Huber dagegen erkundet zuerst die Funktionalität der Komponente `BreakPlanStatisticView`.

Abbildung 5.1 zeigt eine Reihe von Schnappschüssen, die eine denkbare Folge von Systemzuständen beschreiben, die der Pausenplaners einnimmt, während Herr Maier und Herr Huber mit dem System arbeiten. Die Syntax der Darstellung eines Schnappschusses ist ein UML Instanzendiagramm. Alle Elemente in dieser Diagrammart sind Instanzen zur Laufzeit des Systems. In der UML sind Instanzen generell dadurch gekennzeichnet, dass Bezeichner unterstrichen dargestellt werden. Der Anteil des Bezeichners vor dem Doppelpunkt ist der Name der Instanz, der Anteil des Bezeichners nach dem Doppelpunkt ist der Name des Typs (siehe [BJR98, RJB98, OMG00a]).

Die Benutzer des Systems, sogenannte Akteure, werden als Strichmännchen abgebildet. Komponenten werden durch Rechtecke und Schnittstellen durch Kreise repräsentiert. Eine Linie verdeutlicht die Zugehörigkeit einer Schnittstelle zu einer Komponente. Eine Verbindung zwischen zwei Schnittstellen wird ebenfalls durch eine Linie verkörpert. Attribute einer Komponente und deren Werte werden in einem speziellen Rechteck innerhalb der Komponente dargestellt. Die an einer Schnittstelle aktuell anliegenden Nachrichten werden durch einen gerichteten Pfeil symbolisiert.

Der erste Schnappschuss des Systems, zum Zeitpunkt t_1, in Abbildung 5.1 zeigt zwei Benutzer, `Herr Maier : Staff Manager` und `Herr Huber : Break Plan Manager`, die jeweils mit einer Schnittstelle der Komponenteninstanz `: StaffEditor` bzw. `: BreakPlanStatisticView` verbunden sind. An der Schnittstelle der Komponente `: StaffEditor` liegt die Nachricht `addTeacherAsNew` mit den Parameterwerten `newName : String = "Herr Sepp"` und `newJobFactor : Real = 0.5` an. Herr Maier hat diese Nachricht über die Benutzerschnittstelle an die Komponente geschickt, um den Lehrer Herr Sepp, der eine halbe Stelle hat, im System anzulegen.

Im nächsten Schnappschuss, zum Zeitpunkt t_2, hat die Komponente `: StaffEditor` die anliegende Nachricht verarbeitet. Über ihre Verbindung zu der Schnittstelle der Komponente `: StaffOrganizer` hat sie die entsprechende Nachricht an `: StaffOrganizer` weitergeleitet.

Der letzte Schnappschuss zeigt das System zum Zeitpunkt t_3. Die Komponente `: StaffOrganizer` hat die anliegende Nachricht verarbeitet. Sie hat eine neue Subkomponente mit einer entsprechenden Schnittstelle erzeugt, die Komponente `Herr Sepp : Teacher`. Über eine neue interne Schnittstelle ist der `: StaffOrganizer` mit der neu erzeugten Komponente verbunden.

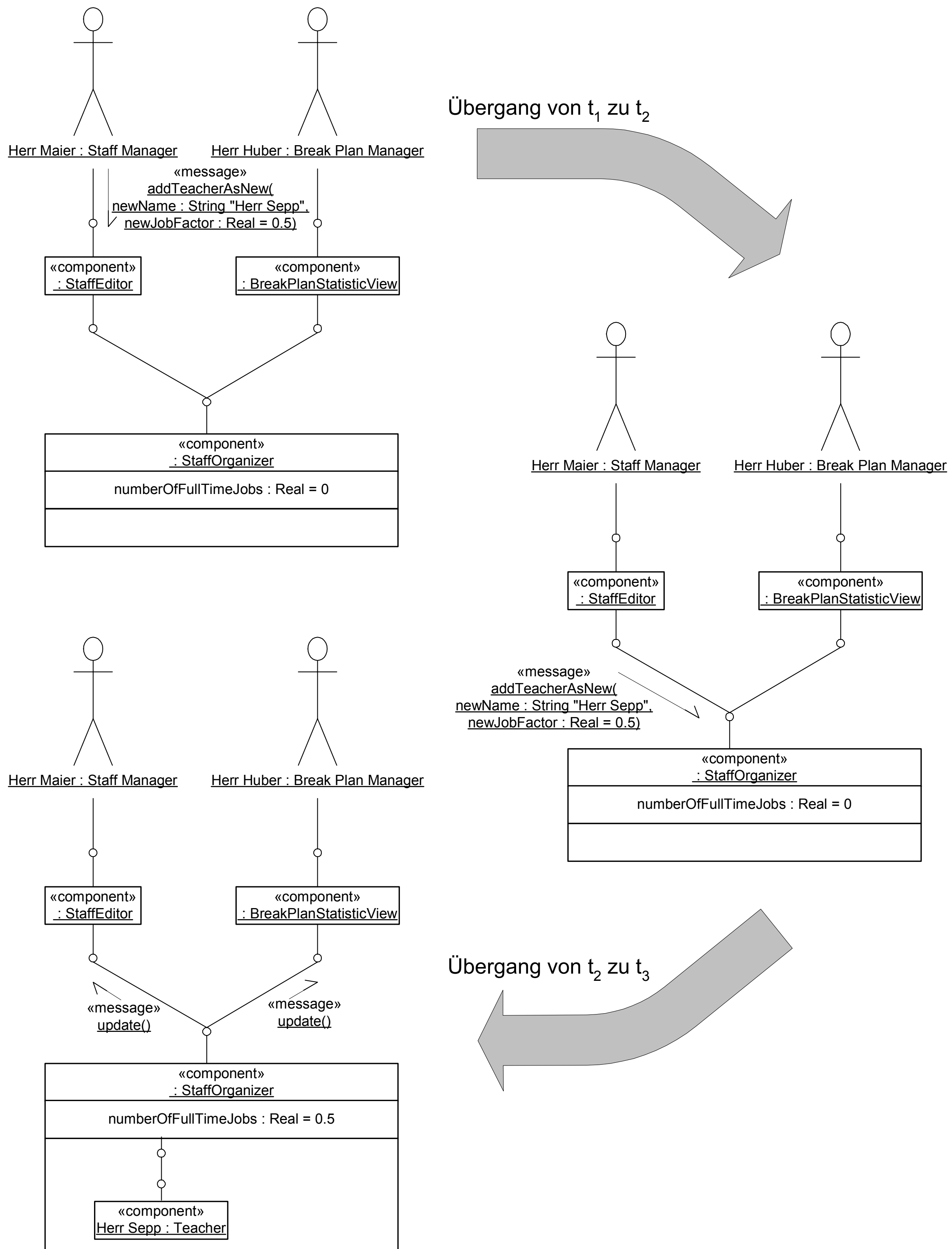

Abbildung 5.1: Eine Folge von Schnappschüssen des Pausenplaners

Der Wert des Attributes `numberOfFullJobs` der Komponente `: StaffOrganizer` hat sich von `0` auf `0.5` geändert, da der Lehrkörper jetzt ein Lehrer mit einer halben Stelle enthält. Und schließlich, dem Observer Pattern entsprechend, hat der `: StaffOrganizer` allen anderen Komponenten die Nachricht `update` geschickt, damit diese die neuen Werte anzeigen können.

Dieses Beispiel zeigt deutlich unser Verständnis eines komponentenbasierten Systems: Zur Ausführungszeit besteht das System aus einer Menge von Komponenten. Den Komponenten sind Attribute und Schnittstellen zugeordnet. Attribute haben einen Wert, Schnittstellen sind über Verbindungen mit anderen Schnittstellen zusammengeschlossen. An Schnittstellen können Nachrichten anliegen. Die Komponenten verarbeiten diese Nachrichten und können dabei neue Nachrichten versenden, die Attributwerte verändern und neue Komponenten, Schnittstellen und Verbindungen erzeugen.

5.2 Grundlegende Konzepte komponentenbasierter Systeme

Das Systemmodell, das wir im Zuge dieses Kapitels erarbeiten, charakterisiert die Menge der Systeme, mit denen wir uns in dieser Arbeit beschäftigen. Dabei beschreibt das Systemmodell insbesondere die Instanzen eines komponentenbasierten Systems zur Laufzeit. Im voranstehenden Kapitel wurden bereits Ausprägungen von Instanzen in komponentenbasierten Systemen identifiziert: Systeme, Komponenten, Attribute, Schnittstellen und Verbindungen.

Definition 5.1: Instanzen in komponentenbasierten Systemen

Die Mengen SYSTEM*,* $\mathsf{COMPONENT}$*,* $\mathsf{INTERFACE}$*,* $\mathsf{ATTRIBUTE}$ *und* $\mathsf{CONNECTION}$ *seien paarweise disjunkte Teilmengen der Menge aller Instanzen* $\mathsf{INSTANCE}$*. Diese Mengen repräsentieren die Bezeichner der Instanzen von Systemen, Komponenten, Schnittstellen, Attribute und Verbindungen in komponentenbasierten Systemen:*

$$\mathsf{SYSTEM} \cup \mathsf{COMPONENT} \cup \mathsf{INTERFACE} \cup \mathsf{ATTRIBUTE} \cup \mathsf{CONNECTION} \subseteq_{def} \mathsf{INSTANCE}$$

Ein komponentenbasiertes System $s \in \mathsf{SYSTEM}$ *besteht dabei aus den folgenden Mengen von Instanzen:*

a) $\mathsf{System}_s \subseteq \mathsf{SYSTEM}$ *sei die Menge der Subsysteme im System* s *mit* $s \notin \mathsf{System}_s$*,*

b) $\mathsf{Component}_s \subseteq \mathsf{COMPONENT}$ *sei die Menge der Komponenten im System* s*,*

c) $\mathsf{Interface}_s \subseteq \mathsf{INTERFACE}$ *sei die Menge der Schnittstellen im System* s*,*

d) $\mathsf{Attribute}_s \subseteq \mathsf{ATTRIBUTE}$ *sei die Menge der Attribute im System* s*,*

e) $\mathsf{Connection}_s \subseteq \mathsf{CONNECTION}$ *sei die Menge der Verbindungen im System* s*, und*

f) $\mathsf{Instance}_s =_{def} \{\mathsf{System}_s \cup \mathsf{Component}_s \cup \mathsf{Interface}_s \cup \mathsf{Attribute}_s \cup \mathsf{Connection}_s\}$ *sei die Menge aller Instanzen im System* s*.*

Das Systemmodell charakterisiert nicht nur die Instanzen, sondern auch deren Verhalten. Aus den Schnappschüssen in Abbildung 5.1 wurde bereits ersichtlich, dass in komponentenbasierten Systemen drei Ausprägungen von Verhaltensformen existieren:

- Struktur komponentenbasierter Systeme:
  Mit dieser Verhaltensform können wir die dynamische Veränderung der Struktur eines Systems während der Laufzeit darstellen. Dies umfasst beispielsweise das Erzeugen und Löschen von Komponenten, Schnittstellen oder Verbindungen, aber auch die Migration einer Subkomponente von einer Superkomponente zur anderen.
- Datenzustand komponentenbasierter Systeme:
  Diese Verhaltensform erlaubt es uns, Veränderungen im lokalen und globalen Datenzustandsraum von Komponenten und Systemen zu beschreiben. Das Ändern eines lokalen Attributwertes einer Komponente oder aber auch das Ändern des Wertes einer globalen Variable ist ein Beispiel für diese Verhaltensform.

- Kommunikation in komponentenbasierten Systemen:
  Mit dieser Verhaltensform können Komponenten sich gegenseitig koordinieren und miteinander interagieren. Dabei kommunizieren Komponenten in unserem Systemmodell durch den asynchronen Austausch von Nachrichten.

In den folgenden Unterkapiteln werden wir diese Verhaltensformen detaillierter analysieren und präzise definieren. Schließlich, im Unterkapitel 5.2.4 charakterisieren wir dann vollständige Schnappschüsse, wie sie in Abbildung 5.1 dargestellt werden, formal.

5.2.1 Struktur komponentenbasierter Systeme

Komponenten sind Bausteine aus denen andere Komponenten konstruiert werden können. Jeder Komponente sind eine Menge von Schnittstellen zugeordnet (ASSIGNMENT). Diese Schnittstellen haben eine Menge von Attribute bzw. Variablen (ALLOCATION). Schnittstellen sind mit anderen Schnittstellen über Verbindungen zusammengeschlossen (CONNECTS). Instanzen komponentenbasierter Systeme können während der Laufzeit eines Systems erzeugt und gelöscht werden (ALIVE).

Definition 5.2: Struktur komponentenbasierter Systeme

Die folgenden Funktionsdefinitionen charakterisieren die Struktur eines komponentenbasierten Systems zu einem bestimmten Zeitpunkt während der Laufzeit des Systems:

$$\begin{aligned}&\text{ALIVE} =_{def} \text{INSTANCE} \rightarrow \text{BOOLEAN}\\&\text{ASSIGNMENT} =_{def} \text{INTERFACE} \rightarrow \text{COMPONENT}\\&\text{ALLOCATION} =_{def} \text{ATTRIBUTE} \rightarrow \text{INTERFACE}\\&\text{CONNECTS} =_{def} \text{CONNECTION} \rightarrow \{\{i, j\} \mid i, j \in \text{INTERFACE}\}\end{aligned}$$

Die Struktur von Schnappschüssen komponentenbasierter Systeme, wie beispielsweise in Abbildung 5.1, können, bis auf Hierarchie und Mobilität, mit diesen Funktionen vollständig beschrieben werden. Hierarchische Systeme führen wir im Kapitel 5.5 ein. Mobile Systeme, bei denen eine Komponente von einer Superkomponente zur nächsten migriert, werden in dieser Arbeit nicht untersucht. Für detailliertere Betrachtungen mobiler Systeme und deren Systemmodelle sei an dieser Stelle auf [BGR+99] verwiesen.

5.2.2 Datenzustand komponentenbasierter Systeme

Der Zustand eines komponentenbasierten Systems wird nicht nur durch die Struktur des Systems bestimmt, sondern auch durch die Werte aller Attribute der einzelnen Komponenten – den Datenzustand der Komponenten. ALLOCATION weist den einzelnen Komponenten ihre Attribute zu. Während der Ausführung des Systems hat jedes Attribut einen aktuell gültigen Wert.

Definition 5.3: Datenzustand komponentenbasierter Systeme

VALUE *sei die Menge der gültigen Attributwerte und eine Teilmenge der Menge aller Instanzen:*

$$\text{VALUE} \subseteq_{def} \text{INSTANCE}$$

Die folgende Funktionsdefinition VALUATION *erlaubt es uns die Wertebelegungen der Attribute zu einem bestimmten Zeitpunkt während der Laufzeit eines Systems zu beschreiben.*

$$\text{VALUATION} =_{def} \text{ATTRIBUTE} \rightarrow \text{VALUE}$$

Dabei gilt dass die Spezifikation eines Wertes der einem Attribut zugewiesen ist der Spezifikation des Attributes entsprechen muss:

$$\forall a \in \mathrm{ATTRIBUTE}, valuation \in \mathrm{VALUATION} \Rightarrow specified\big(valuation(a)\big) = specified(a)$$

Mit dem hier definierten Mechanismus sind wir auch in der Lage, lokale Attribute, Variablen in Methoden und Methodenparameter zu modellieren. Dazu müsste die Menge ATTRIBUTE noch geeignet unterteilt werden. Auf diese Weise könnte man diese Konzepte, die auch in objektorientierten Programmiersprachen oder in komponentenbasierten Infrastrukturen vorhanden sind, in das Modell integrieren. Im Rahmen dieser Arbeit ist dies aber nicht von tieferem Interesse. Eine entsprechende Erweiterung des Modells ist aber durchaus denkbar und problemlos möglich.

5.2.3 Kommunikation in komponentenbasierten Systemen

Die Kommunikation zwischen den Komponenten ist die dritte und letzte Verhaltensformen in komponentenbasierten Systemen. Der hier eingeführte, asynchrone Kommunikationsmechanismus basiert im wesentlichen auf dem formalen Systemmodell von FOCUS [BDD+92, BS01], das am Lehrstuhl von Professor Dr. Manfred Broy entwickelt wurde.

Die Menge M bezeichnet dabei das Universum der Nachrichten. M^* sind endliche Sequenzen von Nachrichten und bilden die grundlegende Einheit der Kommunikation. Dabei empfangen Komponenten über ihre Schnittstellen Nachrichtensequenzen und verschicken Sequenzen von Nachrichten an die Schnittstellen anderer Komponenten.

Definition 5.4: Kommunikation in komponentenbasierten Systemen

Sei M *eine Menge von Zeichen, die wir im folgenden auch als Nachrichten bezeichnen und eine Teilmenge der Menge aller Instanzen:*

$$\mathrm{M} \subseteq_{def} \mathrm{INSTANCE}$$

So, bezeichnet M^* *die Menge aller endlichen Sequenzen über* M*, die als die Menge aller endlichen Tupel über* M *definiert ist. Mit Hilfe der folgenden Funktionsdefinition lässt sich die aktuelle Belegung der Schnittstellen mit Nachrichtensequenzen zu einem bestimmten Zeitpunkt während der Laufzeit eines Systems charakterisieren.*

$$\mathrm{EVALUATION} =_{def} \mathrm{INTERFACE} \rightarrow \mathrm{M}^*$$

An dieser Stelle sei angemerkt, dass dieser Kommunikationsmechanismus sich grundlegend von den Techniken unterscheidet, die in den gängigen prozeduralen Programmiersprachen anzutreffen sind: In diesen Programmiersprachen wird der synchrone bzw. blockierende Prozedur- oder Methodenaufruf verwendet. Der Aufrufer einer Prozedur bzw. Methode wird solange in einen Wartezustand versetzt, bis der Aufruf abgearbeitet worden ist, oder ein Fehler eintritt. Nach erfolgreicher Bearbeitung wird der Aufrufer wieder aktiviert. Eine eventuelles Ergebnis des Aufrufs steht ihm dann sofort zur Verfügung.

Soll das Verhalten einer Prozedur bzw. Methode vollständig spezifiziert werden, muss man dementsprechend das Verhalten der aufgerufenen Prozeduren und Methoden mit beschreiben. In objektorientierten Systemen kann aber unter Umständen, beispielsweise auf Grund von Vererbung, erst zur Laufzeit ermittelt werden, welche konkreten Methoden aufgerufen werden. Deshalb ist es in pragmatischen objektorientierten Ansätzen, die zu einer stärkeren formaleren Fundierung tendieren, wie zum Beispiel Eiffel [Meye92], [Meye97] oder JML

[LBR99], nicht möglich das Verhalten von Methoden vollständig zu beschreiben [BW97]. Wesentliche Eigenschaften des Systems lassen sich nicht spezifizieren.

In unserem Systemmodell soll es aber möglich sein, das Verhalten komponentenbasierter Systemen vollständig zu beschreiben. Basiert die Kommunikation zwischen den Komponenten in einem System auf dem asynchronen bzw. nicht blockierenden Prozedur- oder Methodenaufruf, so kann das Verhalten einer Methode vollständig beschrieben werden. Denn bei der Ausführung einer Methode werden nur asynchrone Nachrichten verschickt. Die Ausführung der Methoden, die den Nachrichten entsprechen, werden unabhängig davon im nächsten Ausführungsschritt durchgeführt. Der Aufrufer einer Methode wird nicht in einen Wartezustand versetzt.

Dementsprechend kann der Aufrufer nicht direkt auf ein Ergebnis zurückgreifen. Ist ein blockierender Funktions- oder Methodenaufruf mit einem entsprechenden Rückgabewert notwendig, so kann dieser zusätzlich modelliert werden. Hierfür ist ein spezielles Protokoll auf Basis der asynchronen Kommunikationsprimitive zu spezifizieren. Einige Ansätze für eine derartige Modellierung wurden in [BMS96], [Broy96] und [Støl96] bereits vorgestellt.

Auf Basis der asynchronen Kommunikationsprimitive ist es möglich im Rahmen eines überschaubaren und handhabbaren Systemmodells das Verhalten von Prozeduren und Methoden vollständig zu spezifizieren. Dabei nehmen wir natürlich den Nachteil in Kauf, dass der normale, blockierende Prozedur- und Methodenaufruf „teuer“ integriert werden muss. Häufig ist aber eine synchrone und blockierende Kommunikation nicht notwendig.

Beispielsweise, kann die Benachrichtigung aller `Observer` durch den `Observable` im Observer Pattern über einen asynchronen Aufruf der Methode `update()` erfolgen (vgl. [GHJV95]). Ein anderes Beispiel ist das Layers Pattern (vgl. [BMR+96]). Die Aufrufe in einer Schichtenarchitektur von einer niedrigeren Schicht zu der nächst höheren können mit Hilfe von asynchronen Nachrichten realisiert werden, wie in [BMR+96] und [BRSV98d] beschrieben wird.

Allerdings gibt es Situationen, bei denen eine asynchrone Modellierung zu einer nicht vertretbaren Komplexität führen würde, wie zum Beispiel bei der Änderung von Attributwerten oder der Änderung der Struktur eines komponentenbasierten Systems. In unserem Systemmodell werden diese Verhaltensformen nicht durch Kommunikation zwischen den Komponenten modelliert, sondern durch die Struktur und den Datenzustand komponentenbasierter Systeme (vgl. Definition 5.2 und Definition 5.3). Aufwendige und komplizierte Kommunikationsprotokolle für den Zugriff und die Veränderung von Zustand und Struktur eines Systems sind somit nicht notwendig.

5.2.4 Schnappschüsse komponentenbasierter Systeme

Auf Basis der drei Verhaltensformen, die wir in den vorhergehenden Kapiteln vorgestellt haben, können wir jetzt Schnappschüsse komponentenbasierter Systeme, wie sie in Abbildung 5.1 gezeigt werden, formal charakterisieren. Ein Schnappschuss besteht aus den aktuellen Informationen über Struktur, Datenzustand und Kommunikationssituation eines komponentenbasierten Systems.

Definition 5.5: Schnappschüsse eines komponentenbasierten Systems

Die Menge der Schnappschüsse aller komponentenbasierter Systeme ist durch die Relationen $\mathsf{SNAPSHOT}$ *beschrieben:*

$$\mathsf{SNAPSHOT} =_{def} \mathsf{ALIVE} \times \mathsf{ASSIGNMENT} \times \mathsf{ALLOCATION} \times \mathsf{CONNECTS} \times \mathsf{VALUATION} \times \mathsf{EVALUATION}$$

$\mathsf{Snapshot_s} \subseteq \mathsf{SNAPSHOT}$ *ist die Menge der möglichen Schnappschüsse eines komponentenbasierten Systems* $s \in \mathsf{SYSTEM}$. *Dabei ist bei jedem Schnappschuss* $\mathsf{snapshot_s} \in \mathsf{Snapshot_s}$ *aus dieser Menge der Definitions- und Wertebereich der einzelnen Funktionen in dem Schnappschuss entsprechend eingeschränkt. Außerdem sind die Funktionen total:*

$$\begin{aligned}
&\forall s \in \mathsf{System}, \mathsf{snapshot_s} = (\mathsf{alive_s}, \mathsf{assigenment_s}, \mathsf{allocation_s}, \mathsf{connects_s}, \mathsf{valuation_s}, \mathsf{evaluation_s}) \in \mathsf{Snapshot_s} \Rightarrow \\
&\quad \mathsf{alive_s} \subseteq \mathsf{Instance_s} \to \mathsf{BOOLEAN} \subseteq \mathsf{ALIVE} \wedge \\
&\quad \mathsf{assignment_s} \subseteq \mathsf{Interface_s} \to \mathsf{Component_s} \subseteq \mathsf{ASSIGNMENT} \wedge \\
&\quad \mathsf{allocation_s} \subseteq \mathsf{Attribute_s} \to \mathsf{Component_s} \subseteq \mathsf{ALLOCATION} \wedge \\
&\quad \mathsf{connects_s} \subseteq \mathsf{Connection_s} \to \{\{i,j\} \mid i,j \in \mathsf{Interface_s}\} \subseteq \mathsf{CONNECTS} \wedge \\
&\quad \mathsf{valuation_s} \subseteq \mathsf{Attribute_s} \to \mathsf{VALUE} \subseteq \mathsf{VALUATION} \wedge \\
&\quad \mathsf{evaluation_s} \subseteq \mathsf{Interface_s} \to M^* \subseteq \mathsf{EVALUATION}
\end{aligned}$$

Ein Schnappschuss eines Systems $\mathsf{snapshot_s}$ ist somit ein Element aus der Menge $\mathsf{Snapshot_s}$ und besteht aus einem Tupel von Funktionen, das

- die aktiven Komponenten, Schnittstellen, Verbindungen und Attribute,
- die Zuordnung von Schnittstellen zu Komponenten,
- die Zuordnung von Attributen zu Komponenten,
- die aktuellen Verbindungen zwischen Schnittstellen,
- die aktuelle Wertebelegung der Attribute, und
- die an den Schnittstellen aktuell anliegenden Nachrichtensequenzen

in dem System $s \in \mathsf{SYSTEM}$ beschreibt. Mit $\mathsf{SNAPSHOT}$ ist die Menge aller möglichen Schnappschüsse definiert – das Universum der Systemzustände.

5.3 Zeit und Verhalten komponentenbasierter Systeme

Das Verhalten eines Systems kann als über die Zeit beobachtbare Veränderungen betrachtet werden. Notwendige Voraussetzung für eine entsprechende Verhaltensbeschreibung ist ein Zeitbegriff. Analog zu FOCUS [BDD+92, BS01] oder Temporal Logic [Lamp89] betrachten wir die Zeit als einen unendlichen Strom vom Zeitintervallen gleicher Länge. Wir verwenden dabei die natürlichen Zahlen N als abstrakte Zeitachse und bezeichnen diese Menge als T.

Aus Gründen der Einfachheit und Generalität verwenden wir ein synchrones, globales Zeitmodell. Dementsprechend gibt es eine globale Uhr deren Zeitachse für alle Komponenten des Systems gültig ist. In Prototyping Umgebungen und realen Systemen lässt sich eine solche globale Uhr durchaus realisieren [HMR+98].

Für die Beschreibung beobachtbarer Veränderungen verwenden wir gezeitete Ströme. Dies sind endliche oder unendliche Sequenzen von Elementen aus einer gegebenen Domäne bzw. Menge.

Definition 5.6: Gezeitete Ströme

Ein gezeiteter Strom, oder präziser ein Strom mit einer diskreten Zeit, von Elementen der Menge X, *ist ein Element von folgendem Typ:*

$$X^T =_{def} \mathbb{N}^+ \to X, \text{ mit } \mathbb{N}^+ =_{def} \mathbb{N} \setminus \{0\}$$

Dabei bezeichnet x^t *ein Element des gezeiteten Stroms* $x \in X^T$ *zum Zeitpunkt* $t \in T$ *wobei gilt* $x^t = x(t)$.

Ein gezeiteter Strom bildet jedes Zeitintervall auf ein Element der Menge X ab. x^t ist das Element aus X zum Zeitpunkt $t \in T$. Auf dieser Basis können wir das Verhalten komponentenbasierter Systeme modellieren: $SNAPSHOT^T$ ist der Typ aller Schnappschusshistorien aller Systeme und $Snapshot_s^T$ ist die Verhaltensrelationen des komponentenbasierten Systems $s \in SYSTEM$.

Definition 5.7: Verhalten eines komponentenbasierter Systeme

Die Menge aller möglichen Schnappschusshistorien – das Universum der Verhaltensrelationen – ist wie folgt charakterisiert:

$$SNAPSHOT^T =_{def} ALIVE^T \times ASSIGNMENT^T \times ALLOCATION^T \times CONNECTS^T \times VALUATION^T \times EVALUTATION^T$$

$Snapshot_s^T \subseteq SNAPSHOT^T$ *ist eine Relation, die das Verhalten des komponentenbasierten Systems* $s \in SYSTEM$ *vollständig beschreibt. Ein Historie von Schnappschüssen – ein konkreter Systemablauf – ist ein gezeiteter Strom* $snapshot_s \in Snapshot_s^T$ *sich ändernder Schnappschüsse* $snapshot_s^t$ *zum Zeitpunkt* $t \in T$.

Damit lassen sich beliebige komponentenbasierte Systeme beschreiben, unter anderem aber auch Systeme, die wir nicht als wohlgeformt bezeichnen würden. Beispielsweise wäre es möglich, dass ein Attribut in verschiedenen Komponenten verwendet wird oder dass in einem System zu einem Zeitpunkt eine Komponente nicht aktiv ist, deren Schnittstellen aktiviert sind und Nachrichten empfangen können. Mit Hilfe von entsprechenden Konsistenzbedingungen können wir sukzessive derartige pathologische Systeme aus unserem Systemmodell ausschließen.

Wir werden im folgenden alle Konsistenzbedingungen angeben, die im Rahmen dieser Arbeit notwendig und sinnvoll sind. Wir erheben aber nicht den Anspruch auf Vollständigkeit. Weiterführende Konsistenzbedingungen können in das Systemmodell integriert werden, ohne dass die Arbeiten in den kommenden Kapiteln davon betroffen sind.

Definition 5.8: Erzeugen und Löschen von Instanzen

Sei $s \in SYSTEM$*, dann gelte zu jeden beliebigen Zeitpunkt, dass alle Attribute, die einer Schnittstelle zugeordnet sind, den gleichen Aktivierungszustand wie die Schnittstelle selbst besitzen:*

$$\forall a \in Attribute_s, i \in Interface_s, t \in T \,.\, allocation_s^t(a) = i \Rightarrow alive_s^t(a) = alive_s^t(i)$$

Außerdem können Verbindungen zwischen Schnittstellen nur dann aktiviert sein, wenn beide Schnittstellen aktiv sind:

$$\forall i, j \in Interface_s, c \in Connection_s, t \in T \,.\, connects_s^t(c) = \{i, j\} \Rightarrow alive_s^t(c) = alive_s^t(i) \wedge alive_s^t(j)$$

Schließlich gilt ganz allgemein, dass Instanzen, die vom aktiven in den inaktiven Zustand übergegangen sind, nicht mehr aktiviert werden können:

$$\forall i \in Instance_s, t \in T \,.\, alive_s^t(i) \wedge \exists n \in T \,.\, n > t \wedge \neg alive_s^n(i) \Rightarrow \neg \exists m \in T \,.\, m > n \wedge alive_s^m(i)$$

Die letzte Bedingung entspricht direkt dem Verhalten gängiger Programmiermodelle. Auch dort ist es eine wesentliche Eigenschaft, dass Instanzen, die gelöscht wurden, nicht mehr benutzt werden können bzw. Speicher der freigegeben wurde nicht mehr verwendet werden kann. Die vorherigen Bedingungen geben den Zusammenhang zwischen dem Erzeugen von Instanzen und der Struktur komponentenbasierter Systeme wider.

Ebenfalls direkt aus den Programmiersprachen übernommen ist die folgende Konsistenzbedingung. Die Signatur von Komponenten darf sich während der Laufzeit des Systems nicht verändern.

Definition 5.9: Zuordnung von Komponenten, Schnittstellen und Attributen

In einem System $s \in \text{SYSTEM}$ *ändert sich über die Zeit die Zuordnung von Schnittstellen zu Komponenten und Attributen zu Schnittstellen nicht:*

$$\forall i \in \text{Interface}_s, t, n \in T \Rightarrow \text{assignment}_s^t(i) = \text{assignment}_s^n(i)$$
$$\forall a \in \text{Attribute}_s, t, n \in T \Rightarrow \text{allocation}_s^t(a) = \text{allocation}_s^n(a)$$

Schließlich sollen die einzelnen Systeme unabhängig und isoliert voneinander betrachtet werden. So können zum Beispiel Schnittstellen oder Komponenten nur in einem System sichtbar sein und existieren.

Definition 5.10: Unabhängigkeit der Systeme

Komponentenbasierte Systeme sind unabhängig voneinander. Die Menge der Instanzen von Systemen sind disjunkt:

$$\forall s_1, s_2 \in \text{System} \,.\, s_1 \neq s_2 \Rightarrow \text{Instance}_{s_1} \cap \text{Instance}_{s_2} = \emptyset$$

5.4 Verhalten und Komposition von Komponenten

Im vorhergehenden Kapitel haben wir das beobachtbare Verhalten komponentenbasierter Systeme definiert. In diesem Kapitel zeigen wir, wie das Systemverhalten aus den Verhaltensbeschreibungen der einzelnen Komponenten berechnet werden kann. Eine konstruktive Vorschrift ermöglicht es aus dem Schnappschuss $\text{snapshot}_s^t \in \text{Snapshot}_s^T$ eines Systems $s \in \text{SYSTEM}$ und den Verhaltensbeschreibungen der Komponenten den nächsten Schnappschuss snapshot_s^{t+1} zu berechnen.

5.4.1 Verhalten von Komponenten

In formalen Modellen werden häufig Zustandsübergangsrelationen verwendet, um das Verhalten von Objekten bzw. Komponenten zu beschreiben (siehe [Rump96] oder [Berg97]). In der Regel beschreiben diese Zustandsübergangsrelationen das Kommunikationsverhalten nach außen und die Änderung des internen Komponentenzustands. In unserem Systemmodell kann eine Komponente aber nicht nur ihre eigenen Attribute, sondern auch die Attribute anderer Komponenten sowie die Struktur des gesamten Systems verändern. Deshalb unterscheidet sich die Interpretation der Zustandsübergangsrelationen in unserem Modell an einigen Stellen entscheidend von den bekannten Ansätzen.

Im Gegensatz zu herkömmlichen Übergangsrelationen – einer Beziehung zwischen dem Vorgänger- und Nachfolgezustand einer Komponente – ist die hier verwendete Zustandsübergangsrelation eine Beziehung zwischen einem Ausschnitt aus dem systemweiten Vorgängerzustand und einem Ausschnitt aus dem „erwünschten“, systemweiten Nachfolgezustand. Der tatsächliche Nachfolgezustand des gesamten Systems wird unter Einbeziehung der Übergangsrelationen aller Komponenten von einer Ausführungsumgebung ermittelt.

Definition 5.11: Zustandsübergangsrelationen von Komponenten eines Systems

Die Menge aller Zustandsübergangsrelationen von Komponenten ist wie folgt definiert:

$$\text{BEHAVIOR} =_{def} \text{SNAPSHOT} \rightarrow \text{SNAPSHOT}$$

Das Verhalten einer Komponente $c \in \text{Component}_s$ *in einem System* $s \in \text{SYSTEM}$ *ist durch die Funktion* behavior_c *charakterisiert, deren Definitions- und Wertebereich entsprechend eingeschränkt ist:*

$$\text{behavior}_c : \text{Snapshot}_s \rightarrow \text{Snapshot}_s$$

Ein Tupel $cbt \in \text{behavior}_c$ *besteht aus zwei Schnappschüssen. Stimmt der im ersten Schnappschuss beschriebene Ausschnitt des systemweiten Zustandes mit dem tatsächlichen Systemzustand überein, so kann die Transition schalten. Das Schalten der Transition bedeutet, dass der nächste Systemzustand sich mit dem, im Tupel beschriebenen, zweiten Ausschnitt des systemweiten Zustands deckt.*

Einer der zentralen Vorteile dieses Modells ist, dass die Realisierung einer entsprechenden Ausführungsumgebung relativ einfach ist (vgl. Kapitel 8). Denn an wesentlichen Stellen gleicht dieses Modell den in der Praxis gängigen komponentenbasierten Programmiermodellen: Alle Komponenten werden nebenläufig ausgeführt. Ein Tupel $cbt \in \text{behavior}_c$ beschreibt das Ergebnis einer möglichen Ausführung einer Methode einer Komponente. Dabei folgt jede Ausführung einem einheitlichen Schema:

Zuerst werden die in der Methode notwendigen, Informationen über den Systemzustand berechnet. Stimmen diese Informationen mit dem Ausgangsschnappschuss in cbt überein, so werden Änderungen an dem Systemzustand vorgenommen werden, in Form von Strukturänderungen, Datenzustandsänderungen und Nachrichtenaustausch. Diese Änderungen sind im Ergebnisschnappschuss in cbt beschrieben.

Im Gegensatz zu den existierenden Programmiermodellen führen in unserem Systemmodell die Komponenten die Änderungen aber nicht selbst durch. Sie beauftragen ein Laufzeitsystem mit der Durchführung. Das Laufzeitsystem kann so Überlappungen in den, von den Komponenten „gewünschten" Nachfolgezuständen identifizieren und auflösen. Mit dieser Technik werden viele der Probleme, die in der Praxis bei der verteilten und nebenläufigen Ausführung komponentenbasierter Systeme auftreten, bereits durch das Ausführungsmodell ausgeschlossen.

Angenommen die Zustandsübergangsrelation einer Komponente fordert, dass ein Attribut den Wert 5 annimmt. Entsprechend der Zustandsübergangsrelation einer anderen Komponente, soll dieses Attribut aber mit dem Wert 0 belegt werden. In diesem Fall ist es unmöglich einen „vernünftigen" Nachfolgezustand zu berechnen. In der Praxis führt dies zu einem nichtdeterministischen Systemverhalten mit nicht reproduzierbaren Fehlerfällen und Abstürzen. In unserem Modell erkennt das Laufzeitsystem derartige Probleme und führt das System in einen wohl definierten Fehlerzustand über. Aus diesem Fehlerzustand kann das System keine weiteren Zustandsübergänge oder anderen Aktivitäten durchführen. Das System wird durch das Ausführungsmodell kontrolliert angehalten.

5.4.2 Vom Komponentenverhalten zum Systemverhalten

Die Aufgabe des Laufzeitsystems ist es, aus dem Schnappschuss $\text{snapshot}_s^t \in \text{Snapshot}_s^T$ des Systems $s \in \text{SYSTEM}$ und aus der Menge der Übergangsrelationen $\{\text{behavior}_{c_1},\ldots,\text{behavior}_{c_n}\}$ aller Komponenten $c_1,\ldots,c_n \in \text{Component}_s$, mit $n \in \mathbb{N}$, den Schnappschuss snapshot_s^{t+1} zu berechnen. Dazu holt sich die Laufzeitumgebung am Ende eines Ausführungsschrittes die gewünschten Ausschnitte des systemweiten Zustandes bei allen Komponenten ab und berechnet daraus einen wohldefi-

nierten Nachfolgezustand des Gesamtsystems. Bevor wir aber eine entsprechende Berechnungsvorschrift definieren können, müssen wir zuerst noch einige Hilfsfunktionen einführen:

Definition 5.12: Projektion auf Mengen

Sei R *eine Relation mit der Stelligkeit* $r \in \mathbb{N}$. *Dann sei* $\pi_{i_1,\ldots,i_n}(R)$ *die Menge der* n*-Tupel mit* $n \in \mathbb{N} \wedge n \leq r$, *die das Ergebnis der Anwendung der Projektion auf* R *sind. Dabei sind in jedem Tupel in* $\pi_{i_1,\ldots,i_n}(R)$ *die Stellen* $i_1,\ldots,i_n$ *des entsprechenden Tupels aus* R *enthalten, wobei gilt* $1 \leq i_k \leq r, \text{ mit } k \in \{1,\ldots,n\} \subseteq \mathbb{N}$.

Definition 5.13: Zustandsübergangsrelation der aktiven Komponenten in einem System

Sei $s \in SYSTEM$ *ein komponentenbasiertes System und* $snapshot_s^t \in Snapshot_s^T$ *der Systemzustand zum Zeitpunkt* $t \in T$, *so ist die Vereinigungsmenge der Tupel in den Übergangsrelationen aller aktiven Komponenten wie folgt definiert:*

$$all_active_behavior_s^t =_{def} \bigcup_{\forall c \in Component_s \cdot (\pi_1(snapshot_s^t))(c)} behavior_c$$

Dabei ist anzumerken, dass im Gegensatz zu der Verhaltensfunktion einer einzelnen Komponente (vgl. Definition 5.11) $all_active_behavior_s^t$ eine Teilmenge der Relation $Snapshot_s \times Snapshot_s$ ist. Denn mehrere aktive Komponenten können Transitionen mit identischen Vorgängerzuständen enthalten und unterschiedlichen Nachfolgezuständen oder umgekehrt.

Definition 5.14: Menge der schaltbaren Zustandsübergänge in einem System

Sei $all_active_behavior_s^t$ *die Zustandsübergangsrelation der aktiven Komponenten eines Systems* $s \in SYSTEM$ *und* $snapshot_s^t \in Snapshot_s^T$ *der zugehörige Systemzustand zum Zeitpunkt* $t \in T$, *so ist die Menge der schaltbaren Zustandsübergänge in einem System wie folgt definiert:*

$$all_active_transition_s^t =_{def} \left\{(x,y) \in all_active_behavior_s^t \middle| \pi_i(x) \subseteq \pi_i\left(snapshot_s^t\right), \forall i = 1\ldots6\right\}$$

Die Menge $all_active_transition_s^t$ enthält nun alle Zustandsübergänge aller aktiven Komponenten, deren erster Schnappschuss mit dem tatsächlichen systemweiten Zustand übereinstimmt. $all_active_transition_s^t$ ist somit die Menge der Zustandsübergänge aller Komponenten, die in einem Zeitschritt ausgeführt werden.

Bevor wir nun die abschließende Definition der Berechnungsvorschrift für den Schnappschuss $snapshot_s^{t+1} \in Snapshot_s^T$ definieren, benötigen wir noch einen zusätzlichen Operator auf Relationen. Dieser Operator erzeugt aus zwei Relationen X und Y eine neue Relation Z. In Z sind alle Tupel aus Y enthalten und alle Tupel aus X, deren erstes Element nicht dem ersten Element irgend eines Tupels aus Y entspricht.

Definition 5.15: Operator für das Ersetzen von Tupeln in Mengen

Sei A *eine Relation und* $X \subseteq A$ *und* $Y \subseteq A$ *zwei Teilmengen dieser Relation, dann sei* $X_{\triangleleft Y} \subseteq A$ *ebenfalls eine Teilmenge von* A *und wie folgt definiert:*

$$X_{\triangleleft Y} =_{def} \left\{a \middle| a \in Y \vee \left(a \in X \wedge \pi_1(\{a\}) \cap \pi_1(Y) = \varnothing\right)\right\}$$

Für die Berechnungsvorschrift für den Schnappschuss $snapshot_s^{t+1}$ ist dieser Operator zentral. Intuitiv gesprochen werden dabei alle Tupel in den einzelnen Funktionen, die im Schnappschuss $snapshot_s^t \in Snapshot_s^T$ enthalten sind, durch die Tupel in den einzelnen Funktionen in der

Menge der schaltbaren Zustandsübergänge $all_active_transition_s^t$ ersetzt, wenn die ersten Elemente in den Tupeln der Funktionen übereinstimmen. Die Veränderungen des Systemzustandes, beschrieben in $all_active_transition_s^t$, werden so in die einzelnen Funktionen eingefügt, wobei die Totalität der Funktionen erhalten bleibt. Somit beschreiben sie den nächsten vollständigen Systemschnappschuss.

Definition 5.16: Einfaches Ausführungsmodell komponentenbasierter Systeme

Die Funktion $next_snapshot$ *charakterisiert die Ausführungsumgebung komponentenbasierter Systeme:*

$$next_snapshot : SNAPSHOT \rightarrow SNAPSHOT$$

Sei $all_active_transition_s^t$ *die Menge der schaltbaren Zustandsübergänge in einem System* $s \in SYSTEM$ *und sei* $snapshot_s^t = (alive_s^t, assignment_s^t, allocation_s^t, connects_s^t, valuation_s^t, evaluation_s^t) \in Snapshot_s^T$ *der zugehörige Systemzustand zum Zeitpunkt* $t \in T$*, so ist der nächste Systemzustand* $next_snapshot(snapshot_s^t)$ *des Systems zum Zeitpunkt* $t+1$ *wie folgt definiert:*

$$next_snapshot(snapshot_s^t) =_{def} snapshot_s^{t+1} = (alive_s^{t+1}, assignment_s^{t+1}, allocation_s^{t+1}, connects_s^{t+1}, valuation_s^{t+1}, evaluation_s^{t+1}).$$

$$alive_s^{t+1} = alive_s^t \triangleleft \pi_7(all_active_transition_s^t) \wedge$$

$$assignment_s^{t+1} = assignment_s^t \triangleleft \pi_8(all_active_transition_s^t) \wedge$$

$$allocation_s^{t+1} = allocation_s^t \triangleleft \pi_9(all_active_transition_s^t) \wedge$$

$$connects_s^{t+1} = connects_s^t \triangleleft \pi_{10}(all_active_transition_s^t) \wedge$$

$$valuation_s^{t+1} = valuation_s^t \triangleleft \pi_{11}(all_active_transition_s^t) \wedge$$

$$evaluation_s^{t+1} = evaluation_s^t \triangleleft \pi_{12}(all_active_transition_s^t)$$

Basierend auf dieser Definition lässt sich der Zustand eines komponentenbasierten Systems stets aus dem Vorgängerzustand und den Zustandsübergangsrelationen der einzelnen Komponenten des Systems berechnen.

Ein nicht lösbarer Konflikt zwischen den, von den Komponenten gewünschten Nachfolgezuständen kann durch das Laufzeitsystem sehr einfach erkannt werden. In obiger Berechnungsvorschrift ist dies genau dann der Fall, wenn bei den einzelnen Zuweisungen auf der rechten Seite eine Relation und keine Funktion steht. Beispielsweise, ist die Auswertung des Terms $valuation_s^t \triangleleft \pi_{11}(all_active_transition_s^t)$ ein Element aus $ATTRIBUTE \times VALUE$ statt aus $ATTRIBUTE \rightarrow VALUE$, so haben mehrere Komponenten unterschiedliche „Vorstellungen" über den Wert, den ein Attribut einnehmen soll. In diesem Fall führt das Laufzeitsystem das System in einen Fehlerzustand über und hält es kontrolliert an.

So kann die Ausführungsumgebung relativ einfach Konflikte erkennen und entsprechend reagieren. In Kapitel 8 zeigen wir, wie diese Berechnungsvorschrift und das zugehörige Systemmodell in einer Prototyping-Umgebung realisiert werden kann.

5.5 Von flachen zu hierarchischen Komponenten und Systemen

Im vorhergehenden Kapitel haben wir die Komposition von Komponenten zu einem komponentenbasierten System beschrieben. Ein zentraler Grundsatz von Componentware ist, dass komponentenbasierte Systeme selbst wieder Komponenten sind und in anderen Systemen verwendet werden. So kann ein höherer Grad an Wiederverwendung und Qualität erreicht werden, eines der meistgenannten Potentiale und Vorteile von Componentware.

In letzter Konsequenz führt dies zu komponentenbasierten hierarchischen Systemen, bei denen ein komponentenbasiertes System aus Komponenten besteht, die ihrerseits wiederum durch weitere komponentenbasierte Systeme realisiert werden. Unser gegenwärtiges Systemmodell kann solche Systeme noch nicht charakterisieren. Ziel dieses Kapitels ist es, das Systemmodell um die notwendigen Konzepte zu erweitern.

Die vorgestellte Lösung orientiert sich dabei prinzipiell an dem Ansatz in [Broy97]. Zuerst erweitern wir die Struktur unserer Systeme um das Konzept der Hierarchie. Im nächsten Schritt führen wir dann eine Abbildung zwischen den Elementen der Systeme auf den verschiedenen Hierarchiestufen ein. Diese, sogenannte Sichtbarkeitsrelation, ermöglicht es, dass die autarken Subsysteme eines komponentenbasierten hierarchischen Systems interagieren können. Damit ist es dann abschließend möglich, in Verbindung mit einem erweiterten Zeit- und Ausführungsmodell, eine Berechnungsvorschrift für das Verhalten komponentenbasierter hierarchischer Systeme anzugeben und so die Ausführungsumgebung derartiger Systeme formal zu spezifizieren.

5.5.1 Struktur komponentenbasierter hierarchischer Systeme

In komponentenbasierten Systemen unterscheiden wir zwei Sichten auf eine Komponente:

- Die Zugriffssicht bzw. Black Box Sicht einer Komponente beinhaltet die Schnittstellen der Komponente. Sie ist ausreichend für die Verwendung der Komponente in einem System.
- Die Implementierungssicht bzw. Glass Box Sicht beschreibt eine mögliche Realisierung der Black Box Sicht einer Komponente.

Die Realisierung einer Komponente kann durch ein komponentenbasiertes System erfolgen. Wir bezeichnen solche Komponenten als hierarchische Komponenten. Die Glass Box Sicht hierarchischer Komponenten besteht aus einer Menge von Black Box Sichten anderer Komponenten, sogenannter Subkomponenten, und einer geeigneten Komposition dieser Black Box Sichten in einem Subsystem (bzw. inneren System). Das System in dem die hierarchische Komponente verwendet wird, wird auch als Supersystem (bzw. äußeres System) bezeichnet. Hat eine Komponente keine Glass Box Sicht oder ist die Glass Box Sicht nicht in Form eines komponentenbasierten System gegeben, so nennen wir diese Komponente auch atomare Komponente. Abbildung 5.2 illustriert diese Begriffe nochmals anhand unseres Anwendungsbeispiels.

Definition 5.17: Struktur komponentenbasierter hierarchischer Systeme

Sei $s \in \mathrm{SYSTEM}$ *ein komponentenbasiertes hierarchisches System, so unterteilen wir die Menge der Komponenten des Systems* $\mathrm{Component}_s$ *in zwei disjunkte Teilmengen. Die Menge der atomaren und die Menge der hierarchischen Komponenten:*

$$\mathrm{AtomicComponent}_s \cup \mathrm{HierachicalComponent}_s =_{def} \mathrm{Component}_s$$

Jede hierarchische Komponente $\mathrm{component}_{hs} \in \mathrm{HierachicalComponent}_s$ *wird von einem entsprechenden Subsystem* $hs \in \mathrm{System}_s$ *des Systems* $s \in \mathrm{SYSTEM}$ *implementiert:*

$$\mathrm{implements}_s : \mathrm{HierachicalComponent}_s \rightarrow \mathrm{System}_s$$

wobei gilt:

$$\mathrm{implements}_s(\mathrm{component}_{hs}) = hs$$

Beispielsweise wird in dem komponentenbasierten hierarchischen System `: BreakPlanner` aus Abbildung 5.2 die hierarchischen Komponente `: StaffOrganizer` von dem Subsystem `: StaffOrganizerImpl` implementiert und somit gilt:

$$\text{implements}_{\underline{:\text{BreakPlanner}}}(\underline{:\text{StaffOranizer}}) = \underline{:\text{StaffOrganizerImpl}}$$

Wie bereits in Kapitel 5.2 erwähnt, sind mobile Systeme nicht Gegenstand dieser Arbeit. Die Super-/Sub-Beziehungen zwischen Komponenten und Systemen, charakterisiert durch die Funktion implements_S, verändert sich deshalb nicht über die Zeit.

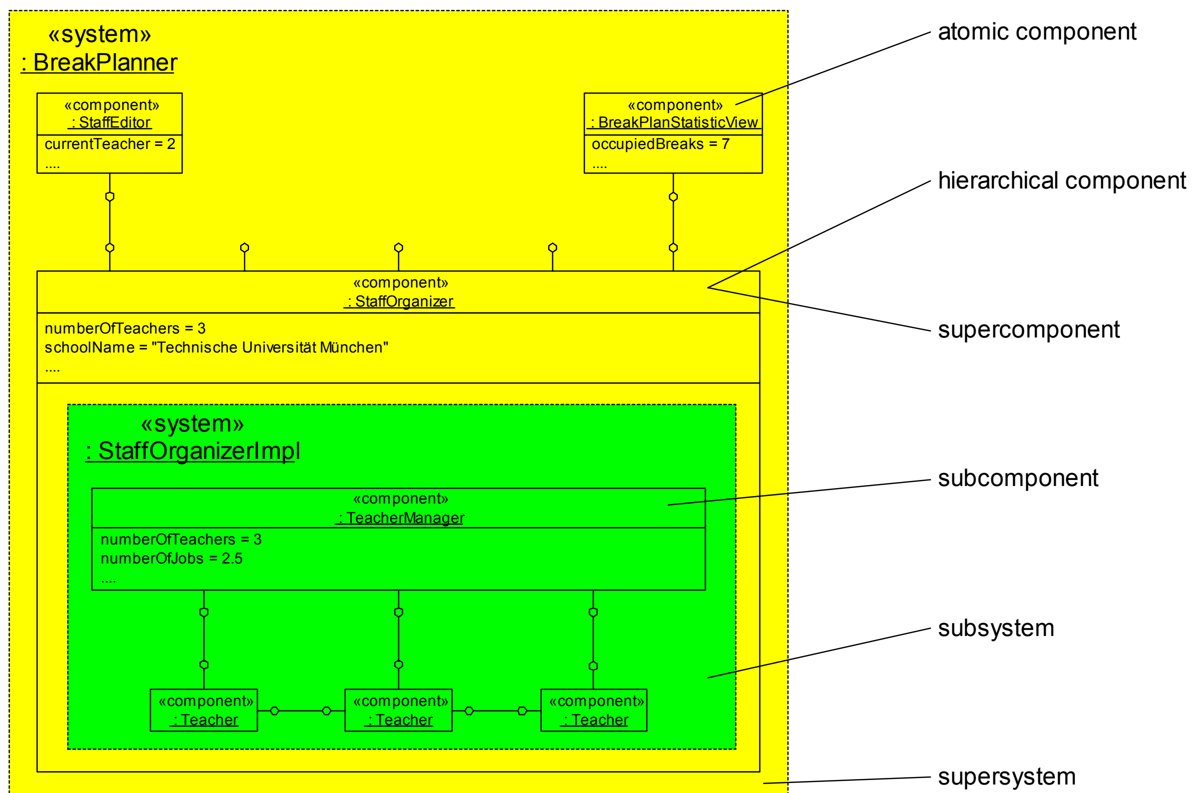

Abbildung 5.2: Struktur komponentenbasierter hierarchischer Systeme

5.5.2 Sichtbarkeitsregeln in komponentenbasierten hierarchischen Systemen

Komponentenbasierte hierarchische Systeme bestehen aus zwei Arten von Komponenten: Atomare Komponenten werden nicht weiter unterteilt und unterscheiden sich nicht von denen, die wir bis jetzt betrachtet haben. Hierarchische Komponenten dagegen besitzen „zwei Gesichter“: Sie treten im äußeren System als eine Komponenteninstanz mit einem eigenen Verhalten in Form einer Zustandsübergangsrelation in Erscheinung (Black Box Sicht). Intern bestehen sie selbst wiederum aus einem vollständigem und geschlossenem komponentenbasierten Subsystem (Glass Box Sicht).

Die einzelnen Subsysteme eines komponentenbasierten hierarchischen Systems sind in sich abgeschlossen und unabhängig von den anderen. Subsystemübergreifend sind keine Interaktionen zwischen den Komponenten möglich. Um die Funktionalität des Gesamtsystems zu gewährleisten, ist es aber in der Regel essentiell, dass die unterschiedlichen Subsysteme miteinander interagieren können. Andererseits muss diese Interaktion bestimmten strukturellen Gesetzmäßigkeiten gehorchen, damit ein geeigneter Rahmen für den Entwurf einer tragfähigen Softwarearchitektur gewährleistet bleibt.

Sowohl in den gängigen Programmiermodellen als auch in den formalen Ansätzen stehen derartige Strukturierungsmöglichkeiten über Sichtbarkeitskonzepte zur Verfügung. Komponenten können nur miteinander interagieren, wenn sie sich kennen. Die Sichtbarkeitsregelen legen fest, welche Teile eines Systems von welchen anderen Teilen „gesehen“ werden und somit interagieren können. Mit Hilfe festgelegter Sichtbarkeitseigenschaften wird so eine wohl definierte Grundstruktur für das komponentenbasierte hierarchische Gesamtsystem erzwungen.

So bieten beispielsweise objektorientierte Programmiersprachen dem Entwickler die Möglichkeit durch das Geheimnisprinzip bzw. Kapselung bestimmte Implementierungsinformationen zu verstecken. In formaleren Ansätzen sind derartige Strukturierungskonzepte noch wesentlich stärker verankert. ROOM erlaubt nur Systeme, deren Komponentenstruktur einem azyklischen Graphen entspricht [SGW94], und in FOCUS sind sogar nur hierarchische Baumstrukturen möglich [BDD+92, BS01].

Bei den formalen Ansätzen sind die Strukturierungskonzepte dabei meist eng an die funktionale Dekomposition der Komponenten geknüpft. Die Sichtbarkeitseigenschaften in unserem Systemmodell sollen wesentlich offener und stärker an die in der Praxis vorherrschenden Programmiermodelle angelehnt sein. In unserem Systemmodell legen die Sichtbarkeitsregeln nur fest, welche Ausschnitte eines Systems einem Subsystem zur Verfügung gestellt werden und umgekehrt. So können die unterschiedlichen Subsysteme miteinander interagieren und die Funktionalität des Gesamtsystems bereit stellen.

Definition 5.18: Sichtbarkeit in komponentenbasierten hierarchischen Systemen

Die Sichtbarkeitseigenschaften eines komponentenbasierten hierarchischen Systems werden durch die Sichtbarkeitsregeln der einzelnen hierarchischen Komponenten festgelegt. Eine Sichtbarkeitsregel ist eine Zuordnung zwischen zwei Instanzen eines hierarchischen Systems:

$$\text{INSTANCE_MAPPING} =_{def} \text{INSTANCE} \times \text{INSTANCE}$$

Sei $\text{component}_{hs} \in \text{HierarchicalComponent}_s$ *eine hierarchische Komponente in dem komponentenbasierten hierarchischen System* $s \in \text{SYSTEM}$*, die von dem Subsystem* $hs \in \text{System}_s$ *implementiert wird. Die Relation* $\text{InstanceMapping}_{(s,\text{component}_{hs},hs)}$ *legt die Sichtbarkeitsregeln der hierarchischen Komponente* component_{hs} *fest und ist wie folgt definiert:*

$$\text{InstanceMapping}_{(s,\text{component}_{hs},hs)} \subseteq_{def} \{\text{Instance}_s \cup \text{Instance}_{hs}\} \times \{\text{Instance}_s \cup \text{Instance}_{hs}\} \subseteq \text{INSTANCE_MAPPING}$$

Die Relation $\text{InstanceMapping}_{(s,\text{component}_{hs},hs)}$ *legt dabei fest, welche Instanzen des Systems* s*, in dem* component_{hs} *verwendet wird, auf welche Instanzen des implementierenden Subsystems* hs *abgebildet werden und umgekehrt. Dabei gilt für jedes Paar aus dieser Relation die folgende Bedingung:*

$$\begin{aligned}
&\forall (x,y) \in \text{InstanceMapping}_{(s,\text{component}_{hs},hs)} \Rightarrow \\
&\quad (x \in \text{COMPONENT} \wedge y \in \text{COMPONENT}) \vee (x \in \text{INTERFACE} \wedge y \in \text{INTERFACE}) \vee \\
&\quad (x \in \text{ATTRIBUTE} \wedge y \in \text{ATTRIBUTE}) \vee (x \in \text{CONNECTION} \wedge y \in \text{CONNECTION})
\end{aligned}$$

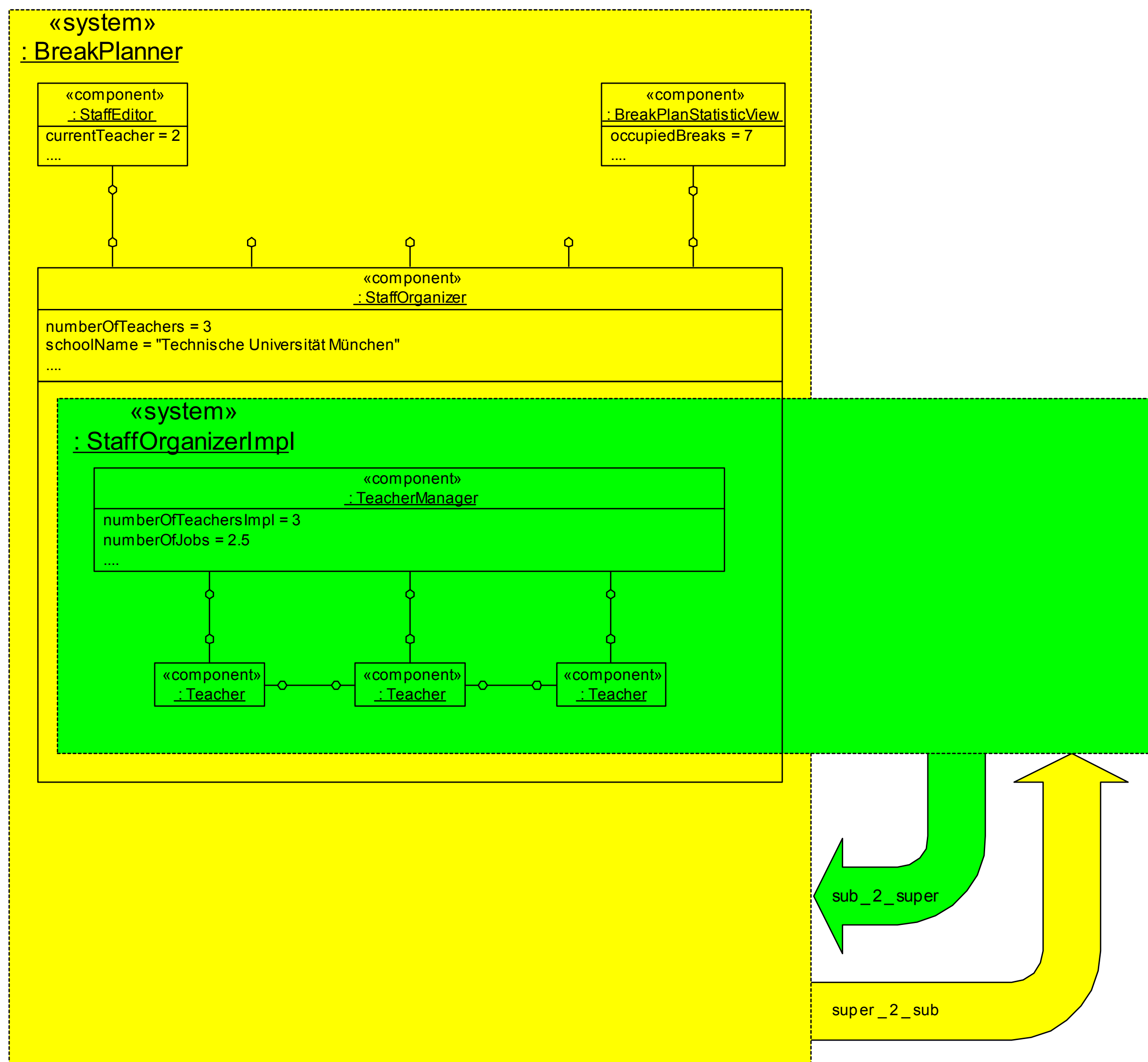

Abbildung 5.3: Sichtbarkeitsregeln und Abbildungen der Systemzustände

So enthält beispielsweise das komponentenbasierte hierarchische System `: BreakPlanner` aus Abbildung 5.3 die hierarchische Komponente `: StaffOrganizer`. Dieser Komponente ist das Attribut `numberOfTeachers` mit dem Wert `3` zugeordnet. Die hierarchische Komponente wird von dem Subsystem `: StaffOrganizerImpl` implementiert, das die Subkomponente `: TeacherManager` enthält. Diese Subkomponente hat ein Attribut mit dem Namen `numberOfTeachersImpl` und dem Wert `3`. Diese Attribute sind zwei verschiedene Instanzen in zwei unterschiedlichen Systemen. Die Sichtbarkeitsregeln der hierarchischen Komponente `: StaffOrganizer` setzen diese zwei Instanzen miteinander in Beziehung:

$$(\text{numberOfTeachersImpl}, \text{numberOfTeachers}) \in \text{InstanceMapping}_{(\underline{:\text{BreakPlanner}}, \underline{:\text{StaffOrganizer}}, \underline{:\text{StaffOrganizerImpl}})}$$

Diese Sichtbarkeitsregel bedeutet, dass das Attribut `numberOfTeachers` der hierarchischen Komponente `: StaffOrganizer` identisch ist mit dem Attribut `numberOfTeachersImpl` der Subkomponente `: TeacherManager`. Das Attribut `numberOfTeachersImpl` ist somit in Form des Attributes `numberOfTeachers` der hierarchischen Komponente `: StaffOrganizer` in dem Supersystem `: BreakPlanner` sichtbar.

Die Relation $\mathrm{InstanceMapping}_{(\underline{:BreakPlanner},\underline{:StaffOrganizer},\underline{:StaffOrganizerImpl})}$ definiert aber nur die Verbindung zwischen dem Super- und dem Subsystem. Sie legt fest, welche Teile des Supersystems auf das Subsystem abgebildet werden sollen und umgekehrt. Sollte sich der Wert eines Attributes beispielsweise ändern, so muss der Wert seines Spiegelbildes ebenfalls entsprechend geändert werden. Die eigentliche Durchsetzung der Sichtbarkeitsregeln erfolgt durch die zwei Funktionen $\mathrm{super_2_sub}$ und $\mathrm{sub_2_super}$, wie in Abbildung 5.3 dargestellt. Entsprechend den Sichtbarkeitsregeln übertragen diese Funktionen Bestandteile des Systemzustandes eines Supersystems zum Subsystem und umgekehrt.

Für die Definition dieser Funktionen führen wir einen neuen Hilfsoperator ein. Dieser Hilfsoperator tauscht in einem Systemschnappschuss entsprechend der Sichtbarkeitsregeln alle Instanzen aus und erzeugt so einen neuen Schnappschuss. Der Definitions- und Wertebereich der einzelnen Funktionen in dem resultierenden Schnappschuss ist dabei auf die Instanzen eingeschränkt, die in den Sichtbarkeitsregeln vorkommen. Dementsprechend wirkt dieser Operator wie ein Filter und Transformator, der die spezifizierten Sichtbarkeitsregeln durchsetzt.

Definition 5.19: Operator für die Durchsetzung der Sichtbarkeitsregeln

Sei $\mathrm{InstanceMapping} \subseteq \mathrm{INSTANCE_MAPPING}$ *eine beliebige Menge von Sichtbarkeitsregeln und sei* $\mathrm{snapshot} = (\mathrm{alive}, \mathrm{assignment}, \mathrm{allocation}, \mathrm{connects}, \mathrm{valuation}, \mathrm{evaluation}) \in \mathrm{SNAPSHOT}$ *ein beliebiger Systemschnappschuss, so ist der Schnappschuss* $[\![\mathrm{snapshot} \| \mathrm{InstanceMapping}]\!]$ *wie folgt definiert:*

$$[\![.\|.]\!] : (\mathrm{SNAPSHOT} \times \mathrm{INSTANCE_MAPPING}) \to \mathrm{SNAPSHOT}$$

dabei gilt:

$$[\![\mathrm{snapshot} \| \mathrm{InstanceMapping}]\!] =_{def} \mathrm{snapshot}' = (\mathrm{alive}', \mathrm{assignment}', \mathrm{allocation}', \mathrm{connects}', \mathrm{valuation}', \mathrm{evaluation}').$$

$$\mathrm{alive}' = \{(a,b) \in \mathrm{ALIVE} \mid (x,a) \in \mathrm{InstanceMapping} \wedge (x,b) \in \mathrm{alive}\} \wedge$$

$$\mathrm{assignment}' = \{(a,b) \in \mathrm{ASSIGNMENT} \mid (x,a) \in \mathrm{InstanceMapping} \wedge (y,b) \in \mathrm{InstanceMapping} \wedge (x,y) \in \mathrm{assignment}\} \wedge$$

$$\mathrm{allocation}' = \{(a,b) \in \mathrm{ALLOCATION} \mid (x,a) \in \mathrm{InstanceMapping} \wedge (y,b) \in \mathrm{InstanceMapping} \wedge (x,y) \in \mathrm{allocation}\} \wedge$$

$$\mathrm{connects}' = \left\{ (a,\{b,c\}) \in \mathrm{CONNECTS} \left| \begin{array}{l} (x,a) \in \mathrm{InstanceMapping} \wedge (y,b) \in \mathrm{InstanceMapping} (z,c) \wedge \\ (y,b) \in \mathrm{InstanceMapping} \wedge (x,\{y,z\}) \in \mathrm{connects} \end{array} \right. \right\} \wedge$$

$$\mathrm{valuation}' = \{(a,b) \in \mathrm{VALUATION} \mid (x,a) \in \mathrm{InstanceMapping} \wedge (x,b) \in \mathrm{valuation}\} \wedge$$

$$\mathrm{evaluation}' = \{(a,b) \in \mathrm{EVALUATION} \mid (x,a) \in \mathrm{InstanceMapping} \wedge (x,b) \in \mathrm{evaluation}\}$$

Auf Basis dieses Hilfsoperators können wir jetzt die zwei Funktionen $\mathrm{super_2_sub}$ und $\mathrm{sub_2_super}$ relativ einfach angeben.

Definition 5.20: Abbildung des Systemzustandes vom Supersystem in das Subsystem

Die Funktion $\mathrm{super_2_sub}$ *sei wie folgt definiert:*

$$\mathrm{super_2_sub} : (\mathrm{SNAPSHOT} \times \mathrm{SNAPSHOT}) \to \mathrm{SNAPSHOT}$$

Sei $\mathrm{component}_{hs} \in \mathrm{HierarchicalComponent}_s$ *eine hierarchische Komponente in dem komponentenbasierten hierarchischen System* $s \in \mathrm{SYSTEM}$. *Das Subsystem* $hs \in \mathrm{System}_s$ *implementiere die hierarchische Komponente* $\mathrm{component}_{hs}$. $\mathrm{InstanceMapping}_{(s,\mathrm{component}_{hs},hs)}$ *beinhalte die Sichtbarkeitsregeln der hierarchische Komponente* $\mathrm{component}_{hs}$.

So ist die den Sichtbarkeitsregeln entsprechende Übertragung des Supersystemzustands $snapshot_s = (alive_s, assignment_s, allocation_s, connects_s, valuation_s, evaluation_s) \in Snapshot_s^T$ *in den Subsystemzustand* $snapshot_{hs} = (alive_{hs}, assignment_{hs}, allocation_{hs}, connects_{hs}, valuation_{hs}, evaluation_{hs}) \in Snapshot_{hs}^T$ *wie folgt definiert:*

$$\begin{aligned}
&super_2_sub(snapshot_{hs}, snapshot_s) =_{def} snapshot'_{hs} = \\
&= (alive'_{hs}, assignment'_{hs}, allocation'_{hs}, connects'_{hs}, valuation'_{hs}, evaluation'_{hs}). \\
&\quad alive'_{hs} = alive_{hs} \lhd \left\| alive_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad assignment'_{hs} = assignment_{hs} \lhd \left\| assignment_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad allocation'_{hs} = allocation_{hs} \lhd \left\| allocation_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad connects'_{hs} = connects_{hs} \lhd \left\| connects_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad valuation'_{hs} = valuation_{hs} \lhd \left\| valuation_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad evaluation'_{hs} = evaluation_{hs} \lhd \left\| evaluation_s \right\| InstanceMapping_{(s,component_{hs},hs)} \right\|
\end{aligned}$$

Die Funktion `sub_2_super` ist analog zur Funktionen `super_2_sub` definiert. Sie sei hier aus Gründen der Vollständigkeit angegeben.

Definition 5.21: Abbildung des Systemzustandes vom Subsystem in das Supersystem

Die Funktion `sub_2_super` *sei wie folgt definiert:*

$$sub_2_super : (SNAPSHOT \times SNAPSHOT) \rightarrow SNAPSHOT$$

Sei $component_{hs} \in HierarchicalComponent_s$ *eine hierarchische Komponente in dem komponentenbasierten hierarchischen System* $s \in SYSTEM$. *Das Subsystem* $hs \in System_s$ *implementiere die hierarchische Komponente* $component_{hs}$. $InstanceMapping_{(s,component_{hs},hs)}$ *beinhalte die Sichtbarkeitsregeln der hierarchische Komponente* $component_{hs}$.

So ist die den Sichtbarkeitsregeln entsprechende Übertragung des Subsystemzustands $snapshot_{hs} = (alive_{hs}, assignment_{hs}, allocation_{hs}, connects_{hs}, valuation_{hs}, evaluation_{hs}) \in Snapshot_{hs}^T$ *in den Supersystemzustand* $snapshot_s = (alive_s, assignment_s, allocation_s, connects_s, valuation_s, evaluation_s) \in Snapshot_s^T$ *wie folgt definiert:*

$$\begin{aligned}
&super_2_sub(snapshot_s, snapshot_{hs}) =_{def} snapshot'_s = \\
&= (alive'_s, assignment'_s, allocation'_s, connects'_s, valuation'_s, evaluation'_s). \\
&\quad alive'_s = alive_s \lhd \left\| alive_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad assignment'_s = assignment_s \lhd \left\| assignment_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad allocation'_s = allocation_s \lhd \left\| allocation_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad connects'_s = connects_s \lhd \left\| connects_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad valuation'_s = valuation_s \lhd \left\| valuation_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\| \wedge \\
&\quad evaluation'_s = evaluation_s \lhd \left\| evaluation_{hs} \right\| InstanceMapping_{(s,component_{hs},hs)} \right\|
\end{aligned}$$

5.5.3 Verhalten von hierarchischen Komponenten und Systemen

In den voranstehenden Kapiteln haben wir die Strukturierungskonzepte unseres Systemmodells so erweitert, dass aus komponentenbasierten Systemen komponentenbasierte hierarchische Systeme geformt werden können. In einem nächsten Schritt wurden Sichtbarkeitsregeln eingeführt, damit die autarken Subsysteme eines komponentenbasierten hierarchischen Systems miteinander interagieren können. Dies erlaubt es abschließend eine konstruktive Vorschrift anzugeben, um das Verhalten eines komponentenbasierten hierarchischen Systems aus

dem Verhalten der Subkomponenten zu berechnen. Dabei können diese Subkomponenten selbst wieder aus weiteren komponentenbasierten hierarchischen Systemen bestehen.

Mit dieser Berechnungsvorschrift ist die funktionale Komposition komponentenbasierter hierarchischer Systeme vollständig in unser Systemmodell integriert. Gleichzeitig liefert sie eine mathematische Spezifikation der Ablaufumgebung für die komponentenbasierten Systeme, die wir in unserem Systemmodell charakterisieren.

Grundlage dieses Verfahrens ist ein erweitertes Ausführungs- und Zeitmodell. In einem hierarchischen System führt jedes System zwei Schritte pro Zeiteinheit aus. Zuerst werden die Subsysteme der hierarchischen Komponenten ausgeführt. Stehen die Ergebnisse dieser Ausführung zur Verfügung, so kann dann das Supersystem ausgeführt werden. Im einzelnen folgt die Ausführung eines komponentenbasierten hierarchischen Systems dem folgenden Schema, das in Abbildung 5.4 grafisch illustriert ist:

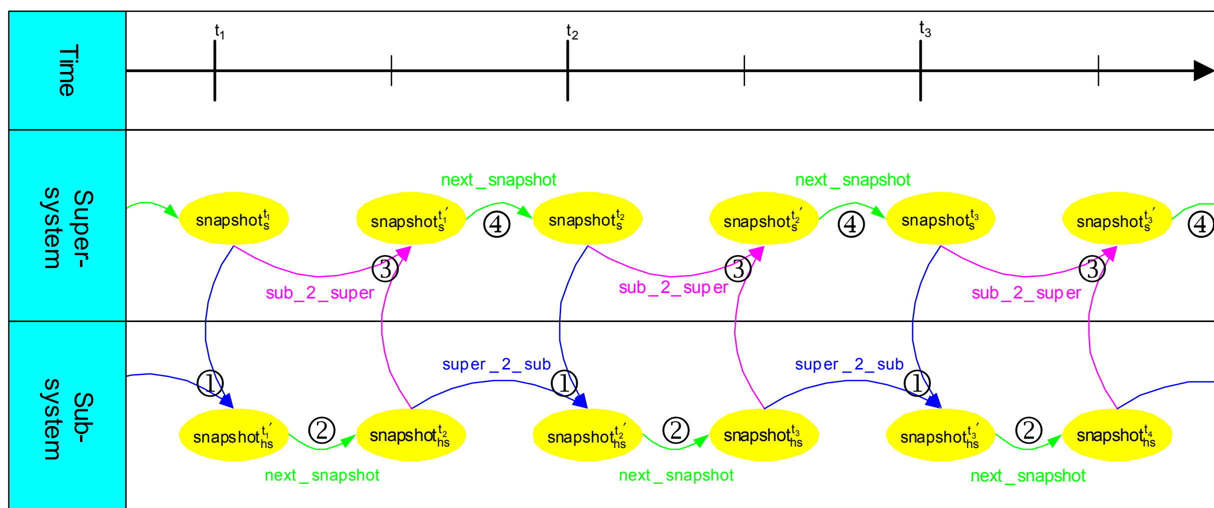

Abbildung 5.4: Erweitertes Zeitmodell und Verhalten hierarchischer Systeme

1. Übertragung des Systemzustandes in die Subsystemzustände:
   Alle Subsysteme werden für den nächsten Ausführungsschritt vorbereitet. Mit der Funktion super_2_sub wird der Systemzustand des Supersystems anhand der Sichtbarkeitsregeln in die Subsystemzustände übernommen.
   Im Beispiel wird aus dem Systemzustand des Supersystems $\text{snapshot}_s^{t_2}$ und dem Systemzustand des Subsystems $\text{snapshot}_{hs}^{t_2}$ mit Hilfe der Funktion super_2_sub und den zugehörigen Sichtbarkeitsregeln $\text{InstanceMapping}_{(s,component_{hs},hs)}$ der neue Systemzustand $\text{snapshot}_{hs}^{t_2'}$ für das Subsystem berechnet.
2. Parallele Ausführung der Subsysteme:
   Der nächste Ausführungsschritt der Subsysteme kann durchgeführt werden. Jedes Subsystem wird unabhängig von den anderen anhand dieser Vorschrift, angefangen beim ersten Punkt, rekursiv ausgeführt.
   Im Beispiel aus Abbildung 5.4 hat das Subsystem hs keine weiteren Subsystem und hierarchischen Komponenten. Die Rekursion terminiert somit an dieser Stelle und der nächste Systemzustand $\text{snapshot}_{hs}^{t_3}$ des Subsystems wird mittels der Funktion next_snapshot berechnet.
3. Übertragung der neuen Subsystemzustände in den Systemzustand:
   Wurden alle Subsysteme erfolgreich ausgeführt, so kann das Supersystem für den nächsten Ausführungsschritt vorbereitet werden. Hierfür wird mit Hilfe der Funktion

sub_2_super die neuen Subsystemzustände in den Systemzustand des Supersystems anhand der zugehörigen Sichtbarkeitsregeln übernommen.
Im Beispiel wird aus dem neuen Systemzustand des Subsystems $snapshot_{hs}^{t_3}$ und dem bestehenden Systemzustand des Supersystems $snapshot_{s}^{t_2}$ ein neuer Systemzustand $snapshot_{s}^{t_2'}$ für das Supersystem mit Hilfe der Funktion sub_2_super berechnet.

4. Ausführung des Systems:
   Nun ist das System bereit für die Ausführung des nächsten Schrittes anhand der Funktion $next_snapshot$.
   Im Beispiel wird aus dem Systemzustand $snapshot_{s}^{t_2'}$ mit der Funktion $next_snapshot$ der nächste Systemzustand des Supersystems $snapshot_{s}^{t_3}$ berechnet.

Durch wiederholte Anwendung dieses Schemas werden die einzelnen Ausführungsschritte eines komponentenbasierten hierarchischen Systems durchgeführt. Die folgende Definition gibt dieses Verfahren mathematisch wieder und stellt so die abschließende Formalisierung der Ausführungsumgebung komponentenbasierter Systeme dar, die wir im Rahmen dieser Arbeit betrachten.

Definition 5.22: Ausführungsmodell komponentenbasierter hierarchischer Systeme

Die Funktion $next_hierarchical_snapshot$ *sei wie folgt definiert:*

$$next_hierarchical_snapshot : SNAPSHOT \rightarrow SNAPSHOT$$

Sei $s \in SYSTEM$ *ein komponentenbasiertes hierarchisches System und sei* $snapshot_s^t \in Snapshot_s^T$ *der Systemzustand dieses Systems zum Zeitpunkt* $t \in T$*, so ist der nächste Systemzustand* $snapshot_s^{t+1}$ *wie folgt definiert:*

$$next_hierarchical_snapshot\left(snapshot_s^t\right) =_{def} snapshot_s^{t+1} = \begin{cases} next_snapshot\left(snapshot_s^t\right), \text{if } HierarchicalComponent_s = \emptyset \\ next_snapshot\left(snapshot_s^{t'}\right), \text{otherwise} \end{cases}$$

$snapshot_s^{t'}$ *ist dabei wie folgt definiert:*

$$snapshot_s^{t'} =_{def} \bigcup_{\forall c \in HierarchicalComponent_s, hs \in System_s \,.\, implements_s(c)=hs} sub_2_super\left(snapshot_s^t, snapshot_{hs}^{t+1}\right)$$

$snapshot_{hs}^{t+1}$ *ist dabei wie folgt definiert:*

$$snapshot_{hs}^{t+1} =_{def} next_hierarchical_snapshot\left(snapshot_{hs}^{t'}\right)$$

und $snapshot_{hs}^{t'}$ *ist dabei wie folgt definiert:*

$$snapshot_{hs}^{t'} =_{def} super_2_sub\left(snapshot_{hs}^t, snapshot_s^t\right)$$

Die Terminierung dieser rekursiven Funktion ist stets gewährleistet. Mit jedem rekursiven Aufruf steigt man in der hierarchischen Struktur des Systems eine Stufe tiefer. Da die Struktur eines Systems in unserem Systemmodell stets eine hierarchische Baumstruktur mit endlicher Breite und Tiefe ist, terminiert die Funktion nach endlich vielen Schritten. Auf einen formalen Terminierungsbeweis verzichten wir im Rahmen dieser Arbeit.

Eine weitere wesentliche Eigenschaft dieser Funktion sei an dieser Stelle angemerkt. Die Ergebnisse der Ausführung aller Subsysteme eines Supersystems (2. Schritt des Verfahrens) werden zusammen mit dem Systemzustand des Supersystems zu einem neuen Systemzustand zusammengefasst (3. Schritt des Verfahrens). Hierbei könnten wiederum Konflikte auftreten:

Angenommen, ein Subsystem setzt den Wert eines Attribut auf 3 und ein anderes Subsystem setzt diesen Wert auf 5. Somit ist ein Konflikt entstanden, der nicht „sinnvoll“ lösbar ist.

In der Definition der Funktion $next_hierarchical_snapshot$ wird diese Problematik bei der Vereinigung der Ergebnisse der Ausführung der Subsysteme sichtbar – bei der Berechung des Systemzustandes $snapshot_s^{t\,\prime}$ des Supersystems. In Analogie zur Definition 5.16 ist der Systemzustand $snapshot_s^{t\,\prime}$ nur dann ein gültiger Systemzustand, wenn die einzelnen Bestandteile des Systemzustands Funktionen und keine Relationen sind. Dementsprechend kann einem Attribut nur ein Wert zugewiesen werden. Würde bei der Berechnung des neuen Systemzustands ein Konflikt auftreten, so kann die Ausführungsumgebung diesen sofort erkennen, das gesamte System anhalten und in einen wohldefinierten Fehlerzustand überführen.

Eine weitere Eigenschaft dieses Verfahrens ist, dass es sowohl die Kompositionseigenschaften formaler Ansätze, als auch die praxisorientierter Programmiermodelle unterstützt. So besteht beispielsweise in den gängigen Programmiermodellen eine Komponente aus Subkomponenten, hat aber zusätzlich noch ein eigenes Verhalten. Die hierarchische Komponente ist somit mehr als die Summe ihrer Bestandteile. In FOCUS dagegen ist eine hierarchische Komponente stets die funktionale Komposition ihrer Subkomponenten und somit nicht mehr als die Summe ihrer Bestandteile. Zwar lässt sich in FOCUS das Modell der gängigen Programmiermodell nachbilden, indem man innerhalb der hierarchischen Komponente das zusätzlich benötigte Verhalten in einer „künstlichen“ Subkomponente beschreibt. Die daraus resultierende Struktur ist aber durch die Spezifikationstechnik bedingt und entspricht nicht unbedingt der Struktur, die der Architekt modellieren wollte. Deshalb ist es aus methodischen Gesichtspunkten sinnvoll beide Kompositionsvarianten zu ermöglichen. In dem vorgestellten Ansatz ist dies möglich: Je nachdem ob die Verhaltensfunktion der hierarchischen Komponente leer ist oder nicht, ist die hierarchische Komponente nur eine funktionale Komposition ihrer Subkomponenten oder besitzt noch zusätzlich ein eigenständiges Verhalten.

5.6 Zusammenfassung

Ein wesentlicher Bestandteil des evolutionären Architekturentwurfs ist die frühzeitige Generierung lauffähiger Prototypen aus Architekturspezifikationen. Hierfür sind ein fundiertes Modell komponentenbasierter Systeme und eine wohldefinierte Ausführungsumgebung notwendig. Getrieben durch technische Fragestellungen sind die in der Praxis eingesetzten Komponentenmodelle den methodischen und konzeptionellen Anforderungen des evolutionären Architekturentwurfs nicht gewachsen.

Deshalb haben wir in diesem Kapitel ein neues, weiter entwickeltes Systemmodell erarbeitet, das eine präzise mathematische Beschreibung des erforderlichen Komponenten- und Architekturmodells sowie der Ausführungsumgebung liefert. Die Architektur eines komponentenbasierten Systems besteht dabei aus Komponenten, die über Schnittstellen Verbindungen zu Komponenten aufbauen, asynchrone Nachrichten austauschen und Attributwerte von Komponenten ändern. Diese Komponenten können selbst wiederum durch weitere komponentenbasierte Systeme realisiert werden.

Jede Komponente hat eine Zustandsübergangsfunktion die ihr Verhalten charakterisiert. Mit Hilfe des formal definierten Ausführungsmodells wird aus dem aktuellen Zustand eines komponentenbasierten Systems und den Verhaltensbeschreibungen der Komponenten der nächste Systemzustand konstruktiv berechnet.

Somit liefert das erarbeitete Systemmodell eine grundlegende Definition und ein exaktes Verständnis für den Begriff einer Softwarearchitektur komponentenbasierter Systeme. Darüber hinaus stellt es eine Spezifikation für eine Ausführungsumgebung zur Verfügung. Sowohl das Architektur- und Komponentenmodell als auch die Ausführungsumgebung sind dabei an die in der Praxis vorherrschenden Ansätze angelehnt. Eine Implementierung ist demzufolge einfach möglich und der pragmatische Nutzen entsprechend hoch.

Mit dem Systemmodell steht ein umfassendes formales Modell zur Verfügung, dass es uns ermöglicht Softwarearchitekturen komponentenbasierter Systeme präzise und vollständig zu spezifizieren. Damit ist dieses Systemmodell die optimale Ausgangsbasis für die semantische Fundierung von Architekturspezifikationen und deren Beschreibungstechniken, die in Kapitel 6 und 7 eingeführt werden. Gleichzeitig ist dieses Systemmodell aber auch die Spezifikation für die Entwicklung einer Werkzeugunterstützung für den evolutionären Architekturentwurf, die in Kapitel 8 vorgestellt wird.

Tabelle 5.1 fasst nochmals die zentralen Konzepte und Definition des Systemmodells für Softwarearchitekturen komponentenbasierter Systeme und des zugehörigen Ausführungsmodells zusammen. Auf die Wiederholung der formalen Definitionen und der Zusammenhänge verzichten wir hier.

$\text{SYSTEM} \cup \text{COMPONENT} \cup \text{INTERFACE} \cup \text{ATTRIBUTE} \cup$ $\cup \text{CONNECTION} \cup \text{VALUE} \cup \text{M} \subseteq \text{INSTANCE}$	Disjunkte Mengen von Systemen, Komponenten, Schnittstellen, Attributen, Verbindungen, Attributwerten und Nachrichten; Teilmengen der Menge aller Instanzen.
$s \in \text{SYSTEM}$ $\text{System}_s \subseteq \text{SYSTEM}$ $\text{AtomicComponent}_s \cup \text{HierarchicalComponent}_s =$ $= \text{Component}_s \subseteq \text{COMPONENT}$ $\text{Interface}_s \subseteq \text{INTERFACE}$ $\text{Attribute}_s \subseteq \text{ATTRIBUTE}$ $\text{Connection}_s \subseteq \text{CONNECTION}$ $\text{Instance}_s = \{\text{System}_s \cup \text{Component}_s \cup \text{Interface}_s \cup$ $\cup \text{Attribute}_s \cup \text{Connection}_s\} \subseteq \text{INSTANCE}$	ein komponentenbasiertes System s Menge der Subsysteme des Systems s Menge der atomaren und hierarchischen Komponenten des Systems s Menge der Schnittstellen des Systems s Menge der Attribute des Systems s Menge der Verbindungen des Systems s Menge aller Instanzen des Systems s
$\text{implements}_s : \text{HirachicalComponent}_s \to \text{System}_s$	ordnet einer hierarchische Komponente das Subsystem zu, das die Komponente implementiert
$\text{ALIVE} : \text{INSTANCE} \to \text{BOOLEAN}$ $\text{ASSIGNMENT} : \text{INTERFACE} \to \text{COMPONENT}$ $\text{ALLOCATION} : \text{ATTRIBUTE} \to \text{INTERFACE}$ $\text{CONNECTS} : \text{CONNECTION} \to \{\{i,j\} \mid i,j \in \text{INTERFACE}\}$ $\text{VALUATION} : \text{ATTRIBUTE} \to \text{VALUE}$ $\text{EVALUATION} : \text{INTERFACE} \to \text{M}^*$	beschreibt welche Instanzen aktiviert sind ordnet den Schnittstellen Komponenten zu ordnet den Attributen Schnittstellen zu ordnet den Verbindungen Schnittstellen zu ordnet den Attributen Werte zu ordnet den Nachrichten Schnittstellen zu
$\text{Snapshot}_s^T \subseteq \text{SNAPSHOT}^T = \text{ALIVE}^T \times \text{ASSIGNMENT}^T \times$ $\times \text{ALLOCATION}^T \times \text{CONNECTS}^T \times$ $\times \text{VALUATION}^T \times \text{EVALUTATION}^T$	Menge aller Schnappschusshistorien des Systems s; die Verhaltensbeschreibung von s
$\text{behavior}_c : \text{SNAPSHOT} \to \text{SNAPSHOT}$	Zustandsübergangsfunktion der Komponente c; die Verhaltensbeschreibung von c
$\text{next_hierarchical_snapshot} : \text{SNAPSHOT} \to \text{SNAPSHOT}$	berechnet den nächsten Systemzustand aus dem aktuellen Systemzustand und den Zustandsübergangsfunktionen aller Komponenten in dem System; die Spezifikation der Ausführungsumgebung

Tabelle 5.1: Das Systemmodell für komponentenbasierte Systeme – Zusammenfassung

6 Architekturspezifikation komponentenbasierter Systeme

Typischerweise werden während der Entwicklung eines Softwaresystems eine Vielzahl verschiedener Entwicklungsdokumente erstellt. Diese Dokumente beinhalten unterschiedliche Beschreibungsmittel, wie zum Beispiel die Unified Modeling Language (UML) [BJR98, RJB98, OMG00a], Entity/Relationship Diagramme [Chen76], Statecharts [HP98], Prosatext oder Pseudo- und Programmcode.

Diese Beschreibungstechniken eigenen sich aber nur eingeschränkt für die Modellierung von Softwarearchitekturen komponentenbasierter Systeme und die maschinelle Weiterverarbeitung der so entstandenen Spezifikationen. Entweder kann der Entwickler damit nur spezifische Aspekte von Komponenten beschreiben, oder er wird von einer Fülle von Beschreibungstechniken überfordert, deren komplexen Querbeziehungen ihm verborgen bleiben.

Statecharts beispielsweise würden sich unter Umständen für die Beschreibung des Ein-/Ausgabeverhaltens von Komponenten eignen. Es existiert aber kein entsprechendes Konzept zur Beschreibung der Struktur komponentenbasierter Systeme und deren Veränderung.

Mit der UML dagegen könnte man verschiedenste Aspekte eines komponentenbasierten Systems spezifizieren. Die Struktur eines Systems ließe sich durch Klassendiagramme ausdrücken. Die Kommunikation könnte man über Sequenzdiagramm beschreiben. Die Zusammenhänge zwischen diesen Beschreibungstechniken sind jedoch noch weitgehend ungeklärt. Darüber hinaus beinhaltet die UML auch Beschreibungstechniken, die für die Modellierung von Softwarearchitekturen nicht relevant sind, wie zum Beispiel Implementierungsdiagramme.

Deshalb erarbeiten wir in diesem Kapitel eine neuartige textbasierte Beschreibungstechnik für Softwarearchitekturen komponentenbasierter Systeme. In Kapitel 6.1 wird am Beispiel des Pausenplaners diese Beschreibungstechnik zuerst exemplarisch demonstriert. Im darauffolgenden Kapitel 6.2 präsentieren wir die wesentlichen Bestandteil der Grammatik der Beschreibungstechnik. Für eine klare Semantik der Beschreibungstechnik sorgt die formale Fundierung in Kapitel 6.3. Diese basiert auf der prädikatenbasierten formalen Semantik aus Kapitel 3 und auf dem Systemmodell für komponentenbasierte Systeme aus Kapitel 5. Dadurch ist die maschinelle Weiterverarbeitung prinzipiell möglich, beispielsweise in Form der Programmgenerierung. Aus Gründen der praktischen Relevanz demonstrieren wir im Kapitel 6.4 eine UML-basierte, grafische Darstellung der zuvor entwickelten und formal fundierten textbasierten Beschreibungstechnik.

6.1 Beispiel einer Architekturspezifikation

Adäquate Beschreibungstechniken für die Spezifikation der Architektur komponentenbasierter Systeme, die wir in den vorhergehenden Kapiteln formal charakterisiert haben, existieren noch nicht. Anhand eines Beispiels gibt dieses Kapitel einen ersten Einblick in eine neuartige, verbesserte Spezifikationstechnik. In den anschließenden Kapiteln werden wir diese Beschreibungstechnik noch eingehender betrachten und präzise definieren.

Unser Anwendungsbeispiel aus Kapitel 4, der Pausenplaner, besteht aus den Komponenten `StaffEditor`, `BreakPlanEditor`, `BreakPlanStatisticView`, `StaffOrganizer` und `BreakPlanOrganizer`. Die Komponente `StaffOrganizer` verwaltet den Lehrkörper und die zugehörigen Lehrer. Mit Hilfe des `BreakPlanOrganizer` werden Pausenpläne und Pausen organisiert.

Abbildung 6.1 zeigt eine erste, vereinfachte Spezifikation der Komponente `StaffOrganizer`. Die Bestandteile der Komponentenspezifikation `BreakPlanOrganizer`, die von `StaffOrganizer` referenziert werden, sind in Abbildung 6.2 dargestellt. So ist das Beispiel eine in sich abgeschlossene und konsistente Spezifikationen, bleibt dabei aber trotzdem überschaubar und verständlich. Schlüsselwörter der Beschreibungstechnik sind in den Abbildungen unterstrichen abgebildet.

Eine Komponentenspezifikation besteht aus einem Namen, wie zum Beispiel `COMPONENT StaffOrganizer`. Das Schlüsselwort `ASSURED` leitet den Anteil der Komponentenbeschreibung ein, der die Zusicherungen der Komponente spezifiziert[1]. Diese bestehen aus einer Menge von Schnittstellen, wie beispielsweise `INTERFACE StaffManager`, mit einer minimalen und maximalen Instanzierungskardinalität, wie zum Beispiel `[1,1]`. Eine Schnittstelle enthält eine Menge von

- Attributen, wie zum Beispiel `ATTRIBUTE school : String`,
- Verbindungen, wie zum Beispiel `CONNECTION observerPattern END observers : Observer [0,*]`,
- Nachrichten, wie zum Beispiel `MESSAGE addNewTeacher(newName : String, newJobFactor : Real)`, und
- Invarianten, wie zum Beispiel `INVARIANT observableBehavior()`.

Ein Attribut besteht aus einem Namen und einem Typ. Die Schnittstelle `StaffManager` beinhaltet beispielsweise den Namen der Schule als Attribut: `ATTRIBUTE school : String`. Der Name des Attributes ist `school` und der Typ ist `String`.

Zusätzlich kann ein Attribut eine Berechnungsvorschrift für den Attributwert enthalten, eingeleitet durch das Schlüsselwort `CALCULATED BY`. Verhaltensspezifikationen sind in unserer Beschreibungstechnik eng an die Object Constraint Language (OCL) [WK98] angelehnt. Allerdings darf bei der Berechung von Attributwerten der Systemzustand nicht verändert werden. Deshalb kann hier nur eine eingeschränkte Variante von OCL verwendet werden.

Ein Beispiel für eine Berechnungsvorschrift eines Attributwertes ist das Attribut `numberOfFullTimeJobs`, die Summe der Lehrstellen des Lehrkörpers. Da Lehrer auch Teilzeitstellen haben können, entspricht diese Summe nicht direkt der Anzahl der Lehrer. Für die Berechnung ist die Summe über die Lehrstellen aller Lehrer eines Lehrkörpers zu bilden. Dies wird durch folgenden Ausdruck in unserer Beschreibungstechnik dargestellt, der gleichzeitig auch

[1] Die Beschreibung der Annahmen einer Komponente führen wir im Kapitel 7 ein.

ein korrekter OCL Ausdruck ist: `result := self.teachers.iterate(t : Teacher ; sum : Real := 0 | sum := sum + t.jobFactor ).`

```
COMPONENT StaffOrganizer

  ASSURED

    INTERFACE StaffManager [1,1]

      ATTRIBUTE school : String
      ATTRIBUTE numberFullTimeJobs : Real CALCULATED BY
        result := self.teachers.iterate(t : Teacher; sum : Real := 0 |
          sum := sum + t.jobFactor);

      CONNECTION breakPlanManagerOfStaffManager
        END theBreakPlanManager : BreakPlanManager [1,1]
      CONNECTION teachersOfStaffManager END teachers : Teacher [0,*]
      CONNECTION observerPattern END observers : Observer [0,*]

      MESSAGE addNewTeacher(newName : String, newJobFactor : Real)
        newTeacher : Teacher := NEW Teacher ASSIGNED TO
          self.assignedComponent;
        newTeachersOfStaffManager : teachersOfStaffManager := NEW
          teachersOfStaffManager BETWEEN newTeacher AND self;
        newTeacher.name := newName;
        newTeacher.jobFactor := newJobFactor;

      INVARIANT observableBehavior()
          self.numberFullTimeJobs <> self@past.numberFullTimeJobs
        implies
          self.observers.forAll(observer : Observer | observer.update());

    INTERFACE Teacher [0,*]

      ATTRIBUTE name : String
      ATTRIBUTE jobFactor : Real
      ATTRIBUTE numberNeededDuties : Real CALCULATED BY
        result := (self.theStaffManager.theBreakPlanManager.numberBreaks /
          self.theStaffManager.numberFullTimeJobs) * self.jobFactor;
      ATTRIBUTE numberCurrentDuties : Integer CALCULATED BY
        result := self.duties.size();

      CONNECTION teachersOfStaffManager END theStaffManager :
        StaffManager [1,1]
      CONNECTION supervisedBreaks END duties : Break [0,*]
      CONNECTION observerPattern END observers : Observer [0,*]

      INVARIANT observableBehavior()
          self.numberNeededDuties <> self@past.numberNeededDuties or
            self.numberCurrentDuties <> self@past.numberCurrentDuties
        implies
          self.observers.forAll(observer : Observer | observer.update());
```

Abbildung 6.1: Textbasierte Spezifikation der Komponente `StaffOrganizer`

Das Schlüsselwort `CONNECTION` spezifiziert jeweils eine Menge von Verbindungen zwischen zwei Schnittstellen. Direkt nach dem Schlüsselwort folgt der Name der Verbindungsmenge. Die Mengen von Endpunkten – die „linke“ und „rechte“ Seite der Verbindungsmenge – werden ebenfalls benannt und haben einen Bezeichner sowie eine Kardinalität.

Der Ausdruck: `CONNECTION teachersOfStaffManager END teachers : Teacher [0,*]` beschreibt dementsprechend die Menge von Verbindungen zwischen Lehrern und dem Lehrkörper mit dem Namen `teachersOfStaffManager`. Diese Verbindung repräsentiert die Zuordnung der Lehrer zu „ihrem" Lehrkörper.

Die „linke" Seite dieser Verbindungsmenge hat den Namen `teachers` und enthält eine Menge von Schnittstellen mit der Spezifikationsbezeichnung `Teacher`. In dieser Menge können null bis beliebig viele Schnittstellen enthalten sein.

Die „rechte" Seite der Verbindungsmenge ist bei der Schnittstelle `Teacher` beschrieben: `CONNECTION teachersOfStaffManager END theStaffManager : StaffManager [1,1]`. Diese Menge enthält genau eine Schnittstelle mit der Spezifikationsbezeichnung `StaffManager`. Damit ist jeder Lehrer stets genau einem Lehrkörper zugeordnet. Die Zuordnung zwischen der „linken" und der „rechten" Seite einer Verbindungsmenge ist durch den Namen der Verbindungsmenge gewährleistet, in unserem Beispiel durch die Bezeichnung `teachersOfStaffManager`.

```
COMPONENT BreakPlanOrganizer

  ASSURED

    INTERFACE BreakPlanManager [1,1]

      ATTRIBUTE numberBreaks : Integer

      CONNECTION breakPlanManagerOfStaffManager END theStaffManager :
        StaffManager [1,1]

    INTERFACE Break [0,*]

    /* the rest of the specification is omitted */
```

Abbildung 6.2: Textbasierte Spezifikation der Komponente `BreakPlanOrganizer`

Das Schlüsselwort `MESSAGE` kennzeichnet die Spezifikation einer Nachrichten, die von einer Schnittstelle verarbeitet werden kann. Dem Schlüsselwort folgt der Name der Nachricht und eine Liste der Parameter, die mit der Nachricht verschickt werden müssen. In unserem Beispiel kann die Schnittstelle `StaffManager` die Nachricht `addNewTeacher` mit der Parameterliste `(newName : String, newJobFactor : Real)` verarbeiten.

Die Beschreibung des Verhaltens einer Komponente bei der Verarbeitung einer Nachricht erfolgt wiederum durch eine Spezifikation die an OCL angelehnt ist. Im Gegensatz zur Spezifikation von Attributwertberechungen verwenden wir hierbei eine erweiterte Variante von OCL. Die folgenden zwei Erweiterungen wurden an OCL vorgenommen:

- Spezifikation des Erzeugens und Löschens von Komponenten, Schnittstellen und Verbindungen, sowie
- Beschreibung der Versendung von Nachrichten an andere Schnittstellen.

Bei der Abarbeitung der Nachricht `addNewTeacher` beispielsweise wird eine neue Schnittstelleninstanz von der spezifizierten Schnittstelle `Teacher` erzeugt und der gleichen Komponente zugeordnet, der die aktuelle Schnittstelle angehört: `newTeacher : Teacher := NEW Teacher ASSIGNED TO self.assignedComponent`.

Die letzte Art der Verhaltensbeschreibung sind Invarianten. Invarianten haben einen Namen, der dem einleitenden Schlüsselwort `INVARIANT` folgt. Der Name ist nur aus methodischen

Gründen sinnvoll. Er ermöglicht es den Entwicklern leichter über spezifische Invarianten in einer Spezifikation zu diskutieren.

Eine Invariante besteht aus einer Bedingung und einer Verhaltensbeschreibung, die durch das OCL Schlüsselwort `implies` getrennt werden. Wenn die Bedingung wahr ist, dann wird die Verhaltensbeschreibung ausgeführt. Die Verhaltensbeschreibung von Invarianten ist identisch zu der Verhaltensspezifikation von Nachrichten. Die Beschreibung der Bedingung einer Invariante entspricht, bis auf eine Erweiterung, einer Attributwertberechungen, deren Ergebnis ein logischer Wert ist. Zusätzlich zu einer Attributwertberechungen kann man bei der Bedingungsspezifikation über das Post-Fix `@past` auf den vorhergehenden Zustand der Schnittstelle und somit des gesamten erreichbaren Systems zugreifen. So können als Bedingungen in Invarianten auch Veränderungen des Systemzustands spezifiziert werden.

Entsprechend den Anforderungen des Pausenplaners unterstützen alle Schnittstellen der Komponente `StaffOrganizer` das Observer Pattern in der Rolle des Observable [GHJV95]. Alle Observer müssen benachrichtigt werden, wenn sich die Anzahl der Lehrstellen eines Lehrkörpers verändert. Die Invariante `observableBehavior()` der Schnittstelle `StaffManager` beschreibt diese Bedingung durch folgenden Ausdruck: `self.numberOfJobs <> self@past.numberOfJobs`. Ist dieser Ausdruck wahr, so werden über die folgende Verhaltensspezifikation alle Observer benachrichtigt: `self.observers.forAll( observer : Observer | observer.update())`.

Mit diesen Sprachmitteln kann eine Komponente vollständig spezifiziert werden. Für die Beschreibung und Modellierung von Softwarearchitekturen komponentenbasierter Systeme sind aber weitere Spezifikationstechniken notwendig. So muss es möglich sein, aus bereits spezifizierten Komponenten Systeme zu formen und deren Initialkonfiguration festzulegen.

Abbildung 6.3 zeigt ein Beispiel für eine Systemspezifikation des komponentenbasierten Systems `BreakPlanner`. Der erste Abschnitt, markiert durch das Schlüsselwort `USED COMPONENTS`, enthält eine Liste von Komponenten, aus denen das System besteht. In unserem Beispiel sind das die zwei Komponenten `StaffOrganizer` und `BreakPlanOrganizer`, die wir zuvor beschrieben haben.

Mit dem Schlüsselwort `INITIALIZATION` beginnt die Spezifikation der Initialkonfiguration des Systems. Diese enthält eine Folge von Anweisungen, die Komponenten, Schnittstellen und Verbindungen erzeugen, Attribute mit Werten besetzen und erste Nachrichten versenden. Die Syntax der Anweisungen in diesem Abschnitt sind identisch mit der Verhaltensbeschreibung für die Abarbeitung von Nachrichten.

Im Beispiel werden zuerst durch die ersten zwei Anweisungen zwei Komponenten erzeugt, beispielsweise mit der Anweisung: `theStaffOrganizer : StaffOrganizer := NEW StaffOrganizer`. Dann werden die entsprechenden Schnittstellen erzeugt und den Komponenten zugewiesen. So wird für die Komponente mit dem lokalen Namen `theStaffOrganizer` eine Schnittstelleninstanz mit dem Namen `theStaffManager` vom der spezifizierten Schnittstelle `StaffManager` erzeugt. Schließlich werden die ersten Verbindungen zwischen den Schnittstellen aufgebaut. Beispielsweise wird die Verbindung zwischen den Schnittstellen `theStaffManager` und `theBreakPlanManager` erzeugt.

Damit ist die anfängliche Struktur des Systems beschrieben. Jetzt werden die Attribute mit Werten belegt und erste Nachrichten an den Schnittstellen angelegt. Beispielsweise mit der Anweisung: `theStaffManager.school := "Technische Universität München"` wird das

Attribut `school` der Schnittsstelle `theStaffManager` mit dem Wert `"Technische Universität München"` vorbelegt.

```
SYSTEM BreakPlanner

  USED COMPONENTS StaffOrganizer, BreakPlanOrganizer

  INITIALIZATION

    theStaffOrganizer : StaffOrganizer := NEW StaffOrganizer;
    theBreakPlanOrganizer : BreakPlanOrganizer := NEW BreakPlanOrganizer;

    theStaffManager : StaffManager := NEW StaffManager
      ASSIGNED TO theStaffOrganizer;
    theBreakPlanManager : BreakPlanManager := NEW BreakPlanManager
      ASSIGNED TO theBreakPlanOrganizer;

    theBreakPlanManagerOfStaffManager : breakPlanManagerOfStaffManager :=
      NEW breakPlanManagerOfStaffManager BETWEEN theStaffManager
      AND theBreakPlanManager;

    theStaffManager.school := "Technische Universität München";
```

Abbildung 6.3: Textbasierte Spezifikation des Systems `BreakPlanner`

6.2 Syntax von Architekturspezifikationen

Das Beispiel aus dem vorhergehenden Kapitel gibt einen ersten Einblick in die textbasierte Beschreibungstechnik für Softwarearchitekturen komponentenbasierter Systeme, die in dieser Arbeit entwickelt und semantisch fundiert werden. Voraussetzung für die semantische Fundierung ist die eindeutige Festlegung der Syntax der Spezifikationstechnik. Für die Beschreibung der Syntax verwenden wir eine erweiterte Form der Backus-Naur Form (BNF) [Wirt86]. Die Produktionsregeln der Grammatik haben die folgende Form:

$$\begin{aligned}\langle \text{NTS} \rangle &::= \\ &\quad pr_1^{\langle \text{NTS} \rangle} \\ &\quad \| \; pr_2^{\langle \text{NTS} \rangle} \\ &\quad \| \; \ldots \\ &\quad \| \; pr_n^{\langle \text{NTS} \rangle}\end{aligned}$$

Nichtterminale Symbole der Grammatik werden durch die umschließenden spitzen Klammern dargestellt. Terminale Symbole werden unterstrichen abgebildet. Auf der rechten Seite des Zuweisungssymbols „$::=$“ werden die verschiedenen Produktionsregeln $pr_1^{\langle \text{NTS} \rangle}$ bis $pr_n^{\langle \text{NTS} \rangle}$ des nichtterminalen Symbols $\langle \text{NTS} \rangle$ angegeben, wobei die Alternativen durch das Symbol „$\|$“ getrennt werden. In Produktionsregeln werden endliche Wiederholungen eines nichtterminalen Symbols $\langle \text{NTS} \rangle$ durch $\{\langle \text{NTS} \rangle\}^*$ dargestellt. Soll das Symbol mindestens einmal wiederholt werden, so wird das durch $\{\langle \text{NTS} \rangle\}^+$ ausgedrückt. Das optionales Auftreten eines Symbols wird durch $\{\langle \text{NTS} \rangle\}^?$ repräsentiert. Wenn ein terminales Symbol $\underline{\texttt{ts}}$ jede Wiederholung des nichtterminalen Symbols $\langle \text{NTS} \rangle$ trennen soll, so schreiben wir $\{\langle \text{NTS} \rangle\}^*_{\underline{\texttt{ts}}}$ bzw. $\{\langle \text{NTS} \rangle\}^+_{\underline{\texttt{ts}}}$.

In den folgenden fünf Unterkapiteln stellen wir die zentralen Bestandteile der Grammatik der neuen Spezifikationstechnik vor, die für das weitere Verständnis und für die semantische Fundierung notwendig sind. Das erste Unterkapitel beschreibt die Struktur von Komponentenspezifikationen. Die nächsten drei Unterkapitel befassen sich jeweils mit den Anteilen der

Grammatik, die bestimmte Ausprägungen von Verhaltensspezifikation beinhalten: die Berechnung von Attributwerten, Bedingungen in Invarianten und das Verhalten von Nachrichten und Invarianten. Das letzte Unterkapitel zeigt die Bestandteile der Grammatik, mit denen Beschreibungen für komponentenbasierte Systeme aus Komponentenspezifikationen geformt werden.

6.2.1 Aufbau und Struktur von Komponentenspezifikationen

Eine Komponentenspezifikation wird über die Produktionsregeln des nichtterminalen Symbols ⟨COMPONENT_SPECIFICATION⟩ erzeugt. Die Spezifikation einer Komponente besteht aus dem Namen der Komponente und einer Menge von zugesicherten Schnittstellenspezifikationen[1].

⟨COMPONENT_SPECIFICATION⟩ ::=
COMPONENT ⟨COMPONENT_NAME⟩ ASSURED {⟨INTERFACE_SPECIFICATION⟩}*

Ein Wort abgeleitet vom dem Symbol ⟨INTERFACE_SPECIFICATION⟩ repräsentiert eine Spezifikation einer Schnittstelle. Diese beinhaltet einen Namen, eine minimale und maximale Instanzierungskardinalität, eine Menge von Attributen, Verbindungen, Nachrichten und Invarianten.

⟨INTERFACE_SPECIFICATION⟩ ::=
INTERFACE ⟨INTERFACE_NAME⟩ [⟨MINIMUM_CARDINALITY⟩ , ⟨MAXIMUM_CARDINALITY⟩]
{⟨ATTRIBUTE_SPECIFICATION⟩}* {⟨CONNECTION_SPECIFICATION⟩}* {⟨MESSAGE_SPECIFICATION⟩}*
{⟨INVARIANT_SPECIFICATION⟩}*

Eine Attributspezifikation ist eine Ableitung aus dem Symbol ⟨ATTRIBUTE_SPECIFICATION⟩. Sie besteht aus einem Namen, einem Typ und optional aus einer Berechnungsvorschrift für das Attribut. Der Typ des Attributes ist primitiver Datentyp, wie zum Beispiel `Integer` oder `String`. Mit der Berechnungsvorschrift für Attributwerte befassen wir uns im Kapitel 6.2.2.

⟨ATTRIBUTE_SPECIFICATION⟩ ::=
ATTRIBUTE ⟨ATTRIBUTE_NAME⟩ : ⟨OCLX_BASIC_TYPE⟩
{CALCULATED BY ⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩}?

Spezifikationen für Verbindungen werden über ⟨CONNECTION_SPECIFICATION⟩ hergeleitet. Die Verbindungsspezifikation besteht aus einem Namen für die Verbindung, einem Namen für das Ende der Verbindung, dem Spezifikationsbezeichner des Verbindungsendes sowie einer minimalen und maximalen Kardinalität.

⟨CONNECTION_SPECIFICATION⟩ ::=
CONNECTION ⟨CONNECTION_NAME⟩ END ⟨CONNECTION_END_NAME⟩ : ⟨INTERFACE_NAME⟩
[⟨MINIMUM_CARDINALITY⟩ , ⟨MAXIMUM_CARDINALITY⟩]

Nachrichtenspezifikationen werden durch Ableitungen aus dem Symbol ⟨MESSAGE_SPECIFICATION⟩ repräsentiert. Eine Nachrichtenspezifikation besteht aus dem Namen der Nachricht, einer Menge von Parameternamen und -Typen und einer optionalen Verhaltensbeschreibung in Form einer erweiterten OCL Spezifikation, die im Kapitel 6.2.4 eingehender besprochen wird.

[1] Die Beschreibung der angenommen Eigenschaften und Schnittstellen werden in Kapitel 7 eingeführt.

⟨MESSAGE_SPECIFICATION⟩ ::=

MESSAGE ⟨MESSAGE_NAME⟩ ({⟨MESSAGE_PARAMETER_NAME⟩ : ⟨OCLX_BASIC_TYPE⟩}*,)

{⟨BEHAVIOR_SPECIFICATION⟩}?

Das nichtterminale Symbol ⟨INVARIANT_SPECIFICATION⟩ entspricht der Menge aller möglichen Invariantenspezifikationen. Eine Invariante besteht aus einem Namen, einer optionalen Bedingung in Form einer erweiterten OCL Spezifikation (siehe Kapitel 6.2.3) und einer optionalen Verhaltensbeschreibung in Form einer erweiterten OCL Spezifikation, die ausgeführt wird wenn die Bedingung gültig ist (siehe. Kapitel 6.2.4).

⟨INVARIANT_SPECIFICATION⟩ ::=

INVARIANT ⟨INVARIANT_NAME⟩ () {⟨INVARIANT_CONDITION_SPECIFICATION⟩ implies

⟨BEHAVIOR_SPECIFICATION⟩}?

6.2.2 Spezifikation von Attributwertberechnungen

Bei der Abarbeitung von Nachrichten und Invarianten greifen Komponenten lesend auf ihre und die Attributwerte anderer Komponenten zu. Attributwerte können dabei durch Berechnungsvorschriften bestimmt sein. Die Ausführung der Berechnungsvorschrift darf den Systemzustand nicht verändern. Trotzdem soll sich die Syntax von Verhaltensbeschreibungen möglichst an OCL anlehnen. Deshalb haben wir für die Spezifikation der Berechnungsvorschrift von Attributwerten eine stark eingeschränkte Version von OCL ausgearbeitet. Damit OCL Anteile in unserer Grammatik auch sichtbar sind, beginnen alle Produktionen, die sich an OCL anlehnen oder sogar identisch zu OCL sind mit dem Präfix „OCLX_“.

Das nichtterminale Symbol ⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩ charakterisiert alle Berechnungsvorschriften für Attributwerte. Eine Berechnungsvorschriften besteht entweder aus einem OCL let- bzw. OCL if-Block, der rekursiv weitere Berechnungsvorschriften enthält, oder aus der terminalen Produktion einer Zuweisung. Am Ende der Auswertung der Berechungsvorschrift enthält das OCL Schlüsselwort result den Attributwert.

⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩ ::=

let {⟨OCLX_LOCAL_BASIC_VARIABLE_DECLARATION⟩}+, in ⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩

‖ if ⟨OCLX_BOOLEAN_VALUE_CALCULATION_EXPRESSION⟩
then ⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩
else ⟨ATTRIBUTE_CALCULATION_SPECIFICATION⟩ endif

‖ result := ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ ;

Explizite Zuweisungen gibt es in OCL nicht, da OCL keine operationale Sprache ist. Statt dessen kann man in OCL über den Gleichheitsoperator = das Ergebnis einer Zuweisung beschreiben. Aus Gründen der Klarheit und Einfachheit haben wir die OCL Syntax um das Konzept der Zuweisung erweitert. Der Zuweisungsoperator ist in unserer Spezifikationssprache := und der Gleichheitsoperator bleibt =.

In allen anderen Bereichen haben wir für die Syntax der Berechnungsvorschrift von Attributwerten OCL massiv beschnitten. Beispielsweise sind keine Sequenzen von Anweisungen möglich, nur eine einzige Zuweisung kann ausgeführt werden. In let-Blöcken können nur lokale Variablem mit primitiven Typen deklariert werden, wie zum Beispiel Integer oder

`String`. Variablen vom Typ einer Schnittstelle oder einer Komponente sind bereits syntaktisch nicht möglich und semantisch auch nicht sinnvoll.

Das nichtterminale Symbol ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ liefert die eigentliche Berechung des Attributwertes. Die ersten drei Produktionen erzeugen wohlgeformte Terme über primitive Werte. Die restlichen fünf Produktionen beschreiben die Berechung der primitiven Werte.

```
⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ ::=
    ( ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ )
  ‖ {⟨OCLX_BASIC_UNARY_OPERATION⟩}+ ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩
  ‖ ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩⟨OCLX_BASIC_INFIX_OPERATION⟩
      ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩
  ‖ ⟨OCLX_BASIC_VALUE⟩
  ‖ ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ . ⟨OCLX_BASIC_VALUE_FEATURE_CALL⟩
  ‖ self. ⟨ATTRIBUTE_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ .
      ⟨OCLX_ELEMENT_OPERATION_ON_COLLECTION⟩ . ⟨ATTRIBUTE_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ .
      ⟨OCLX_BASIC_VALUE_OPERATION_ON_COLLECTION⟩
```

Die einfachste Variante ist, dass der berechnete Werte einem konstanten Wert entspricht, der durch die Spezifikation vorgegebenen ist (⟨OCLX_BASIC_VALUE⟩), wie zum Beispiel dem Integer Wert `7` oder dem String Wert `"Technische Universität München"`.

Darüber hinaus gibt es in OCL die Möglichkeit vordefinierte Operationen auf primitiven Werten auszuführen (⟨OCLX_BASIC_VALUE_FEATURE_CALL⟩). Deren Ergebnis ist wiederum ein primitiver Werte. So kann man auf einem String die Operation `size()` aufrufen, die als Ergebnis die Länge der Zeichenkette zurückgibt (vgl. [WK98]).

Über das Schlüsselwort `self` wird in der OCL Syntax der aktuelle Kontext während der Ausführung selektiert. Somit referenziert die Produktion `self.` ⟨ATTRIBUTE_NAME⟩ einen Attributwert der aktuellen Schnittstelleninstanz.

Die letzten zwei Produktionen sind die interessantesten, aber auch komplexesten. Sie zeigen eine der zentralen Eigenschaften von OCL, die Navigation in Objektgeflechten. In unserem Systemmodell entspricht diese Navigation dem Folgen von Verbindungen zwischen Schnittstellen. Folgt man einer Verbindung so erhält man eine Menge von Instanzen von Schnittstellen. Soll aus dieser Menge von Schnittstelleninstanzen ein primitiver Wert berechnet werden, gibt es zwei Möglichkeiten:

- Entweder selektiert man aus der Menge der Schnittstellen eine und verweist dann auf ein Attribut dieser Schnittstelle, ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ `.` ⟨OCLX_ELEMENT_OPERATION_ON_COLLECTION⟩ `.` ⟨ATTRIBUTE_NAME⟩,
- oder man verwendet eine vordefinierte OCL Operation auf Mengen, die einen primitiven Wert zurückliefert, ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ `.` ⟨OCLX_BASIC_VALUE_OPERATION_ON_COLLECTION⟩.

Letzteres wird beispielsweise in der Komponentenspezifikation in Abbildung 6.1 verwendet, um die Anzahl der Pausen, die ein Lehrer beaufsichtigt, zu berechnen, das Attribut `numberCurrentDuties` der Schnittstelle `Teacher`. Der Ausdruck `result :=`

`self.duties.size()` navigiert zuerst zu der Menge der Pausenaufsichtenschnittstellen und berechnet die Anzahl der Elemente in dieser Menge über die vordefinierte OCL Operation `size()`. Das Ergebnis dieser Berechnung wird `result` zugewiesen und ist gleichzeitig das Ergebnis der Attributwertberechnung.

Die Navigation selbst durch das Geflecht von Schnittstellen und Komponenten ist selbstverständlich auch über Schnittstellen hinweg möglich, wie die entsprechende Produktion ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ zeigt. Ausgehend von der aktuellen Schnittstelle, dargestellt durch das Schlüsselwort `self`, kann man über Verbindungen hinweg komplexe Navigationen spezifizieren.

⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ ::=
    `self.` ⟨CONNECTION_END_NAME⟩
    ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ `.` ⟨CONNECTION_END_NAME⟩
    ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ `.`
      ⟨OCLX_COLLECTION_OPERATION_ON_COLLECTION⟩

Die letzte Produktion erlaubt auch noch vordefinierte OCL Operationen auf Mengen auszuführen, die wiederum Mengen zurückliefern (⟨OCLX_COLLECTION_OPERATION_ON_COLLECTION⟩). Damit lassen sich zum Beispiel anhand von Prädikaten bestimmte Untermengen spezifizieren, die dann weiter verwendet werden können. Beispiele solcher OCL Operationen sind `select` oder `collect`. Für eine vollständige Liste der möglichen Operationen sei auf [WK98] verwiesen.

6.2.3 Spezifikation von Bedingungen in Invarianten

Schnittstellenbeschreibungen enthalten Invarianten. Eine Invariante besteht aus einer Bedingungsbeschreibung und einer Verhaltensbeschreibung, die ausgeführt wird, wenn die Bedingung gültig ist. Das nichtterminale Symbol ⟨INVARIANT_CONDITION_SPECIFICATION⟩ legt den syntaktischen Aufbau von Bedingungsbeschreibungen fest.

Bedingungen sind aussagenlogische Ausdrücke. Mit den ersten drei Produktionsregeln werden aussagenlogische Terme aufgebaut. Die Blätter dieser rekursiv aufgebauten, aussagenlogischen Terme sind durch die letzten zwei Produktionsregeln definiert. Blätter sind entweder die logischen Werte `true` oder `false` (⟨OCLX_BOOLEAN_VALUE⟩), oder eine Relation zwischen zwei primitiven Werten, deren Ergebnis ein logischer Wert ist, wie zum Beispiel der Vergleich `9 < 10`.

⟨INVARIANT_CONDITION_SPECIFICATION⟩ ::=
    `(` ⟨INVARIANT_CONDITION_SPECIFICATION⟩ `)`
    ‖ {⟨OCLX_BOOLEAN_UNARY_OPERATOR⟩}⁺ ⟨INVARIANT_CONDITION_SPECIFICATION⟩
    ‖ ⟨INVARIANT_CONDITION_SPECIFICATION⟩⟨OCLX_BOOLEAN_INFIX_OPERATOR⟩
      ⟨INVARIANT_CONDITION_SPECIFICATION⟩
    ‖ ⟨OCLX_BOOLEAN_VALUE⟩
    ‖ ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩⟨OCLX_RELATIONAL_INFIX_OPERATOR⟩
      ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩

Die zwei primitiven Werte in der letzten Produktionsregel sind berechnete Werte, ähnlich zu der Berechnung von Attributwerten in Kapitel 6.2.2. Die dort verwendete Produktion ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩ kann für die Bedingungen in Invarianten aber

nicht verwendet werden. Denn die Bedingung einer Invariante kann auch eine Veränderung des Zustands sein, wie beispielsweise beim Observer Pattern. Soll eine derartige Bedingung spezifiziert werden, so ist ein Zeitbegriff notwendig, der in der Produktion für Attributwertberechnungen nicht vorhanden ist und auch nicht benötigt wird.

In OCL existiert bereits ein sehr rudimentärer Zeitbegriff für die Spezifikation von Zustandsübergängen. Mit dem Schlüsselwort `@pre` kann man in Nachbedingungen von Methoden auf Werte vor der Ausführung dieser Methode zugreifen. Für eine adäquate Spezifikation von Bedingungen in Invarianten ist dieses Konzept aber noch nicht ausreichend. Hierfür wäre es notwendig auf Werte vor dem letzen Ausführungsschritt des Gesamtsystems zugreifen zu können, nicht nur auf die Werte vor der Ausführung der aktuellen Methode bzw. Invariante.

Deshalb haben wir ein neues Schlüsselwort eingeführt: `@past`. Über dieses Schlüsselwort kann man auf den Systemzustand vor dem letzten Ausführungsschritt zugreifen. So referenziert die Spezifikation aus Abbildung 6.1 mit den Ausdruck `self@past.numberNeededDuties` den Wert des Attributes `numberNeededDuties` vor dem letzten Ausführungsschritt.

⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩ ::=
  `(` ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩ `)`
  ‖ {⟨OCLX_BASIC_UNARY_OPERATION⟩}$^{+}$ ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩
  ‖ ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩⟨OCLX_BASIC_INFIX_OPERATION⟩
    ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩
  ‖ ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩ `.` ⟨OCLX_BASIC_VALUE_FEATURE_CALL⟩
  ‖ ⟨OCLX_BASIC_VALUE⟩
  ‖ `self.` ⟨ATTRIBUTE_NAME⟩
  ‖ `self@past.` ⟨ATTRIBUTE_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ `.`
    ⟨OCLX_ELEMENT_OPERATION_ON_COLLECTION⟩⟨ATTRIBUTE_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ `.`
    ⟨OCLX_BASIC_VALUE_OPERATION_ON_COLLECTION⟩

Die zwei Produktionen ⟨OCLX_BASIC_VALUE_TIMED_CALCULATION_EXPRESSION⟩ und ⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ entsprechen den Produktion für die Berechnung von Attributwerten, die wir bereits in Kapitel 6.2.2 vorgestellt haben. An den entsprechenden Stellen wurde nur das neue Schlüsselwort `@past` hinzugefügt. Dies ermöglicht es Bedingungen in Invarianten so zu spezifizieren, dass auch Zustandsänderungen über die Zeit beschrieben werden können. Beispielsweise, der Auslöser der Benachrichtigung aller Observer im Observer Pattern lässt sich damit sehr einfach beschreiben, wie in Abbildung 6.1 bereits gezeigt wurde.

⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ ::=
  `self.` ⟨CONNECTION_END_NAME⟩
  ‖ `self@past.` ⟨CONNECTION_END_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ `.` ⟨CONNECTION_END_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_TIMED_NAVIGATION_EXPRESSION⟩ `.`
    ⟨OCLX_COLLECTION_OPERATION_ON_COLLECTION⟩

6.2.4 Verhaltensspezifikation von Invarianten und Nachrichten

Für die vollständige Grammatik von Komponentenspezifikationen fehlt noch die Syntax der Verhaltensbeschreibungen in Invarianten und Nachrichten, die Produktion ⟨BEHAVIOR_SPECIFICATION⟩. In Analogie zu der Syntax von Attributwertberechnungen aus Kapitel 6.2.2 besteht ⟨BEHAVIOR_SPECIFICATION⟩ aus verschachtelten OCL `let`- und OCL `if`-Blöcken. Zusätzlich kann der Entwickler Sequenzen von einzelnen Anweisung angeben.

⟨BEHAVIOR_SPECIFICATION⟩ ::=
  `let` $\{$⟨OCLX_LOCAL_BASIC_VARIABLE_DECLARATION⟩$\}^{+}_{,}$ `in` ⟨BEHAVIOR_SPECIFICATION⟩
  ‖ `if` ⟨OCLX_BOOLEAN_VALUE_CALCULATION_EXPRESSION⟩
   `then` ⟨BEHAVIOR_SPECIFICATION⟩
   `else` ⟨BEHAVIOR_SPECIFICATION⟩ `endif`
  ‖ $\{$⟨OCLX_BEHAVIOR_EXPRESSION⟩$\}^{+}_{;}$

Eine Einzelanweisung in einer Verhaltensspezifikation ist durch die Produktionsregel ⟨OCLX_BEHAVIOR_EXPRESSION⟩ definiert. Eine Anweisung ist entweder eine Wertzuweisung, das Verschicken einer Nachricht, die Iteration über Mengen von Schnittstellen, das Erzeugen bzw. Löschen von Komponenten, Schnittstellen sowie Verbindungen, oder das Anhalten des Systems in einem Fehlerzustand.

⟨OCLX_BEHAVIOR_EXPRESSION⟩ ::=
  ‖ ⟨OCLX_ASSIGNABLE_VARIABLE⟩ `:=` ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩
  ‖ ⟨OCLX_ACCESSIBLE_INTERFACE⟩ `.` ⟨MESSAGE_NAME⟩ `(` $\{$⟨OCLX_MESSAGE_PARAMETER⟩$\}^{*}_{,}$ `)`
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ `.`
   ⟨OCLX_ITERATION_OPERATION_ON_COLLECTION⟩
  ‖ ⟨OCLX_CREATE_COMPONENT_EXPRESSION⟩
  ‖ ⟨OCLX_CREATE_INTERFACE_EXPRESSION⟩
  ‖ ⟨OCLX_CREATE_CONNECTION_EXPRESSION⟩
  ‖ ⟨OCLX_DELETE_COMPONENT_EXPRESSION⟩
  ‖ ⟨OCLX_DELETE_INTERFACE_EXPRESSION⟩
  ‖ ⟨OCLX_DELETE_CONNECTION_EXPRESSION⟩
  ‖ ⟨OCLX_ERROR_EXPRESSION⟩

Die rechte Seite der Zuweisung in der ersten Produktionsregel, das Schlüsselwort ⟨OCLX_BASIC_VALUE_CALCULATION_EXPRESSION⟩, entspricht der Berechnung von Attributwerten aus Kapitel 6.2.2. Der berechnete Wert kann entweder einer lokalen Variablen oder dem Attribut einer erreichbaren Schnittstelle zugewiesen werden.

⟨OCLX_ASSIGNABLE_VARIABLE⟩ ::=
  ⟨OCLX_LOCAL_BASIC_VARIABLE_NAME⟩
  ‖ ⟨OCLX_ACCESSIBLE_INTERFACE⟩ `.` ⟨ATTRIBUTE_NAME⟩

Erreichbare Schnittstellen werden sowohl bei der Zuweisung in der ersten Produktionsregel, als auch beim Versenden von Nachrichten in der zweiten Produktionsregel verwendet. Das Symbol ⟨OCLX_ACCESSIBLE_INTERFACE⟩ charakterisiert erreichbare Schnittstellen. Die aktuelle Schnittstelle, angesprochen über das Schlüsselwort `self`, und die Schnittstellen ⟨OCLX_LOCAL_INTERFACE_INSTANCE_NAME⟩, die im Kontext dieser Nachricht oder Invariante neu erzeugt wurden, sind erreichbar. Die dritte Produktionsregel beschreibt die Navigation von

der aktuellen Schnittstelle zu anderen Schnittstellen. Die einzelnen Bestandteile dieser Produktion wurden bereits in Kapitel 6.2.2. vorgestellt.

⟨OCLX_ACCESSIBLE_INTERFACE⟩ ::=
  self
  ‖ ⟨OCLX_LOCAL_INTERFACE_INSTANCE_NAME⟩
  ‖ ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ .
    ⟨OCLX_ELEMENT_OPERATION_ON_COLLECTION⟩

Nachrichten, die an erreichbare Schnittstellen verschickt werden, über die Produktion ⟨OCLX_ACCESSIBLE_INTERFACE⟩ . ⟨MESSAGE_NAME⟩ ({⟨OCLX_MESSAGE_PARAMETER⟩}$^{*}_{,}$), können Parameterlisten mit ausschließlich primitiven Werten enthalten. Verweise auf Schnittstellen oder Komponenten können nicht als Parameter einer Nachricht verschickt werden.

Für realistische Verhaltensspezifikationen sind Schleifenkonstrukte, wie sie in den Programmiersprachen verfügbar sind, essentiell. Beispielsweise im Observer Pattern, müssen alle Observer benachrichtigt werden, wenn sich der Zustand des Observable geändert hat. In der Spezifikation aus Abbildung 6.1 wird dies durch den Ausdruck `self.observers.forAll(observer : Observer | observer.update())` beschrieben. Die Operation `forAll` ist dabei eine in OCL vordefinierte Operation zur Iteration über Mengen, mit der man Schleifen in Programmiersprachen simulieren kann.

Mit Hilfe des nichtterminalen Symbols ⟨OCLX_ITERATION_OPERATION_ON_COLLECTION⟩, das in der dritten Produktionsregel für Einzelanweisungen enthalten ist, lassen sich diese, in OCL vordefinierten, Operationen über Mengen erzeugen. Neben der Operation `forAll` existieren weitere Operationen über Mengen in OCL. Die vollständige Liste aller Operationen kann [WK98] entnommen werden.

Die restlichen sieben Produktionsregeln des Symbols ⟨OCLX_BEHAVIOR_EXPRESSION⟩, das Einzelanweisungen in Verhaltensspezifikationen charakterisiert, sind für die syntaktischen Konstrukte zur Erzeugung und zum Löschen von Komponenten, Schnittstellen und Verbindungen sowie für den expliziten Übergang in einen Fehlerzustand zuständig. Die einzelnen Produktionen sind nahezu selbsterklärend und durch das Beispiel in Kapitel 6.1 bereits ausreichend beschrieben. Aus Gründen der Vollständigkeit sind sie im folgenden noch aufgelistet:

⟨OCLX_CREATE_COMPONENT_EXPRESSION⟩ ::=
  ⟨OCLX_LOCAL_COMPONENT_INSTANCE_NAME⟩ : ⟨COMPONENT_NAME⟩ := NEW ⟨COMPONENT_NAME⟩

⟨OCLX_CREATE_INTERFACE_EXPRESSION⟩ ::=
  ⟨OCLX_LOCAL_INTERFACE_INSTANCE_NAME⟩ : ⟨INTERFACE_NAME⟩ := NEW ⟨INTERFACE_NAME⟩
    ASSIGNED TO ⟨OCLX_LOCAL_COMPONENT_INSTANCE_NAME⟩
  ‖ ⟨OCLX_LOCAL_INTERFACE_INSTANCE_NAME⟩ : ⟨INTERFACE_NAME⟩ := NEW ⟨INTERFACE_NAME⟩
    ASSIGNED TO ⟨OCLX_ACCESSIBLE_INTERFACE⟩ . assignedComponent

⟨OCLX_CREATE_CONNECTION_EXPRESSION⟩ ::=
  ⟨OCLX_LOCAL_CONNECTION_INSTANCE_NAME⟩ : ⟨CONNECTION_NAME⟩ :=
    NEW ⟨CONNECTION_NAME⟩ BETWEEN ⟨OCLX_ACCESSIBLE_INTERFACE⟩
    AND ⟨OCLX_ACCESSIBLE_INTERFACE⟩

⟨OCLX_DELETE_COMPONENT_EXPRESSION⟩ ::=
DELETE COMPONENT ⟨OCLX_ACCESSIBLE_INTERFACE⟩ . assignedComponent

⟨OCLX_DELETE_INTERFACE_EXPRESSION⟩ ::=
DELETE INTERFACE ⟨OCLX_ACCESSIBLE_INTERFACE⟩

⟨OCLX_DELETE_CONNECTION_EXPRESSION⟩ ::=
DELETE CONNECTION ⟨OCLX_COLLECTION_OF_INTERFACE_NAVIGATION_EXPRESSION⟩ .
⟨OCLX_ELEMENT_OPERATION_ON_COLLECTION⟩

⟨OCLX_ERROR_EXPRESSION⟩ ::=
ERROR

6.2.5 Spezifikation komponentenbasierter Systeme

In den vorhergehenden Kapiteln haben wir die Produktionen für Komponentenspezifikationen vorgestellt, so wie sie in Abbildung 6.1 und Abbildung 6.2 exemplarisch abgebildet sind. Die Produktionsregel für Systemspezifikationen, wie zum Beispiel das System `BreakPlanner` aus Abbildung 6.3, ist wie folgt definiert.

⟨SYSTEM_SPECIFICATION⟩ ::=
SYSTEM ⟨SYSTEM_NAME⟩ USED COMPONENTS {⟨COMPONENT_NAME⟩}$^{*}_{,}$
{INITIALIZATION ⟨BEHAVIOR_SPECIFICATION⟩}$^{?}$
{FOR EACH INSTANCE OF ⟨COMPONENT_NAME⟩ EXISTS IMPLEMENTATION INSTANCE
⟨SYSTEM_NAME⟩ {WITH VISIBILITY ⟨MAPPING_SPECIFICATION⟩}$^{*}_{,}$ }*

Eine Systemspezifikation besteht demzufolge aus dem Namen des Systems, den verwendeten Komponenten in dem System und einer optionalen Beschreibung zur Initialisierung des Systems. Die Initialisierungsbeschreibung entspricht syntaktisch den Verhaltensspezifikationen in Nachrichten und Invarianten.

⟨MAPPING_SPECIFICATION⟩ ::=
⟨SPECIFIER_NAME⟩ → ⟨SPECIFIER_NAME⟩
‖ ⟨SPECIFIER_NAME⟩ ← ⟨SPECIFIER_NAME⟩

⟨SPECIFIER_NAME⟩ ::=
⟨COMPONENT_NAME⟩
‖ ⟨INTERFACE_NAME⟩
‖ ⟨ATTRIBUTE_NAME⟩
‖ ⟨CONNECTION_NAME⟩
‖ ⟨CONNECTION_END_NAME⟩
‖ ⟨MESSAGE_NAME⟩
‖ ⟨INVARIANT_NAME⟩

Außerdem kann man bei Systemspezifikationen auch noch angeben, für welche Komponenten des Systems, mit der Instanzierung einer dieser Komponenten, ein entsprechendes Subsystem instanziert werden soll, das die Komponentenfunktionalität realisiert. Hierfür kann zusätzlich eine Menge von Einträgen in die Sichtbarkeitsrelation der Komponente über das nichtterminale Symbol ⟨MAPPING_SPECIFICATION⟩ erfolgen. Ein Element aus ⟨MAPPING_SPECIFICATION⟩ ist eine Abbildung zwischen Spezifikationsbezeichnern. Wobei die Spezifikationsbezeichner jeweils von der selben Art sein müssen. Ein gültiger Eintrag in der Abbildung ist beispielsweise eine Abbildung von einem Komponentenbezeichner auf einen anderen Komponentenbezeichner. Eine Abbildung von einem Komponentenbezeichner auf einen Schnittstellenbezeichner ist jedoch nicht erlaubt.

Aus Systemspezifikationen kann man über das syntaktische Konstrukt ⟨ROOT_SYSTEM_INSTANCIATION⟩ Instanzen erzeugen. Diese Instanzen haben einen Namen und einen Verweis auf die zugehörige Systemspezifikation.

⟨ROOT_SYSTEM_INSTANCIATION⟩ ::=
  SYSTEM INSTANCE ⟨SYSTEM_INSTANCE_NAME⟩ OF SYSTEM ⟨SYSTEM_NAME⟩

Damit stehen jetzt alle Bestandteile der Syntax unserer textbasierten Beschreibungstechnik für die Spezifikation von Softwarearchitekturen komponentenbasierter Systeme zur Verfügung. Eine entsprechende Spezifikation besteht aus einer Systeminstanzierung, einer nicht leeren Menge von Systemspezifikationen und einer Menge von Komponentenspezifikationen.

⟨ARCHITECTURE_SPECIFICATION⟩ ::=
  ⟨ROOT_SYSTEM_INSTANCIATION⟩ $\{$⟨SYSTEM_SPECIFICATION⟩$\}^{+}$ $\{$⟨COMPONENT_SPECIFICATION⟩$\}^{*}$

6.3 Semantik von Architekturspezifikationen

In dem vorhergehenden Kapitel haben wir die Syntax von Architekturspezifikationen in Form von Produktionen einer Grammatik beschrieben. Ziel dieses Kapitels ist es die semantische Fundierung dieser Spezifikationen auf Basis des Systemmodells aus Kapitel 5 vorzunehmen. Dabei folgen wir dem, in Kapitel 3 vorgestellten, Ansatz der prädikatenbasierten formalen Semantik.

In den folgenden sechs Unterkapiteln geben wir jeweils die Eigenschaften an, die sich aus den einzelnen Bestandteilen einer Architekturspezifikation ableiten lassen. Das erste Unterkapitel beschreibt kurz die Grundlagen der Formalisierung und legt so die Verbindung zwischen Syntax und Semantik fest. Die Struktur der restlichen Unterkapitel orientiert sich an den Produktionsregelen der Beschreibungstechnik und somit an der Struktur des voranstehenden Kapitel 6.2.

6.3.1 Grundlagen der Formalisierung von Architekturspezifikationen

Die Produktionsregeln aus Kapitel 6.2 legen die Grammatik und somit die Syntax von Architekturspezifikationen fest. Mit jedem nichtterminalen Symbol ⟨NTS⟩ ist auch eine Sprache L(V,T,P,⟨NTS⟩) definiert, eine Menge von Wörtern aus einem Alphabet von terminalen Symbolen (vgl. auch [HU88]). Der syntaktische Aufbau der Wörter ist durch die Produktionsregeln bestimmt. Dementsprechend legt die Sprache L(V,T,P,⟨ARCHITECTURE_SPECIFICATION⟩) die Syntax von Architekturspezifikationen fest und entspricht der Menge aller Spezifikationen (vgl. Kapitel 3.4, Definition 3.10).

Definition 6.1: Grammatik und Sprache von Architekturspezifikationen

Sei ASG *die Grammatik der Architekturspezifikationen komponentenbasierter Systeme und wie folgt definiert:*

$$\mathsf{ASG} =_{def} \left(\mathsf{V},\mathsf{T},\mathsf{P},\langle\mathsf{ARCHITECTURE_SPECIFICATION}\rangle\right)$$

$\langle\mathsf{ARCHITECTURE_SPECIFICATION}\rangle$ *ist das Startsymbol. Die Menge* P *beinhaltet die Produktionsregeln der Grammatik aus Kapitel 6.2. Die Menge* T *besteht aus den terminalen Symbolen und die Menge* V *umfasst die nichtterminalen Symbole. Terminale Symbole sind in den Produktionen durch Unterstreichung gekennzeichnet und die nichtterminalen durch die Umklammerung mit „* $\langle\ \rangle$ *“. Die Sprache* $\mathsf{L}(\mathsf{ASG})$ *ist somit eine Teilmenge aller syntaktisch korrekten Architekturspezifikationen:*

$$\mathsf{L}(\mathsf{ASG}) \subseteq_{def} \mathsf{SPECIFICATION}$$

In der Menge $\mathsf{L}(\mathsf{ASG})$ *sind alle terminalen Wörter der Sprache – alle Architekturspezifikationen – enthalten. Für jedes Element aus der Sprache* $\mathsf{as} \in \mathsf{L}(\mathsf{ASG})$ *existiert eine endliche Folge von Produktionsregeln, die angewendet werden müssen, um das Wort* as *aus dem nichtterminalen Symbol* $\langle\mathsf{ARCHITECTURE_SPECIFICATION}\rangle$ *abzuleiten.*

Während der Ableitung des terminalen Wortes as *durch die Anwendung von Produktionsregeln entstehen zwischenzeitlich Wörter, die nichtterminale Symbole enthalten. Die Menge der terminalen Teilwörter von* as*, die aus einem nichtterminalen Symbol* $\langle\mathsf{NTS}\rangle$ *abgeleitet werden, bezeichnen wir mit* $\mathsf{L}\left(\langle\mathsf{NTS}\rangle_{\mathsf{as}}\right)$*:*

$$\mathsf{L}\left(\langle\mathsf{NTS}\rangle_{\mathsf{as}}\right) =_{def} \left\{\mathsf{w} \in \mathsf{L}\left(\mathsf{V},\mathsf{T},\mathsf{P},\langle\mathsf{NTS}\rangle\right) \middle| \exists \mathsf{n},\mathsf{w},\mathsf{m} \in \mathsf{T}^{*}\,.\, \mathsf{as} = \mathsf{n}\,\mathsf{w}\,\mathsf{m}\right\}$$

Gemäß dieser Definition, ist $\mathsf{as} \in \mathsf{L}(\mathsf{ASG})$ eine beliebige Architekturspezifikation und $\mathsf{L}\left(\langle\mathsf{COMPONENT_NAME}\rangle_{\mathsf{as}}\right)$ die Menge aller Komponentennamen, die in der Architekturspezifikation vorkommen.

Dem Ansatz der prädikatenbasierten formalen Semantik von Spezifikationen aus Definition 3.6 folgend, implementiert eine Menge von Systemen $\mathsf{System}_{\mathsf{as}} \subseteq \mathsf{SYSTEM}$ genau dann eine konsistente und vollständige Architekturspezifikation $\mathsf{as} \in \mathsf{SPECIFICATION}$, wenn gilt:

$$\forall \mathsf{sys} \in \mathsf{System}_{\mathsf{as}}, \forall \mathsf{a} \in \mathsf{assured}(\mathsf{as}) \Rightarrow \mathsf{a}[\mathsf{sys}]$$

Und, nach der These 3.1 entspricht dies ebenfalls dem Ansatz der systemmodellbasierten formalen Semantik von Dokumentenmengen wie folgt:

$$\mathsf{sem}(\mathsf{as}) = \mathsf{System}_{\mathsf{as}}$$

Einfacher ausgedrückt ist somit jedes System $\mathsf{sys} \in \mathsf{System}_{\mathsf{as}} \subseteq \mathsf{SYSTEM}$ genau dann eine korrekte Implementierung einer konsistenten und vollständigen Spezifikation $\mathsf{as} \in \mathsf{SPECIFICATION}$, wenn alle, aus der Spezifikation mittels $\mathsf{assured}(\mathsf{as})$ berechneten Eigenschaften, bei der Belegung mit dem System sys gültig sind und somit die Modellbeziehung $\mathsf{a}[\mathsf{sys}]$ für alle Eigenschaften $\mathsf{a} \in \mathsf{assured}(\mathsf{as})$ gilt.

Entsprechend diesem Ansatz müssen wir für die semantische Fundierung der Spezifikationstechnik alle Eigenschaften angeben, die aus einer beliebigen Spezifikation $\mathsf{as} \in \mathsf{L}(\mathsf{ASG}) \subseteq \mathsf{SPECIFICATON}$ mittels der Funktion $\mathsf{assured}(\mathsf{as})$ berechnet werden können, unter der Annahmen, dass die Spezifikation konsistent und vollständig ist und somit gilt: $\mathsf{consistent}(\mathsf{as}) \wedge \mathsf{complete}(\mathsf{as})$.

Bei einer korrekten Implementierung der Spezifikation as sind diese Eigenschaften dann stets wahr. Die Gültigkeit der Eigenschaften zu gewährleisten ist aber nicht Aufgabe der semantischen Fundierung. Dies kann nur durch formale Entwicklungskalküle, methodische Richtlinien, die Korrektheit des Generators oder den Programmierer selbst sicher gestellt werden.

Die Verbindung zwischen der Menge der syntaktischen Einheiten in Spezifikationen $\mathsf{SPECIFIER}$ und den semantischen Einheiten des Systemmodells $\mathsf{INSTANCE}$ wird über die Funktion $\mathsf{specified}$ modelliert (vgl. Definition 3.5). In einer Architekturspezifikation werden die Eigenschaften der Spezifikationsbezeichner beschrieben, von denen dann während der Ausführung Instanzen erzeugt werden, die den spezifizierten Charakteristika genügen.

Definition 6.2: Spezifikationsbezeichner einer Architekturspezifikation

Sei $\mathsf{as} \in \mathsf{L}(\mathsf{ASG})$ *eine syntaktisch korrekte Architekturspezifikation, so ist die Menge der Spezifikationsbezeichner* $\mathsf{Specifier}_{\mathsf{as}} \subseteq \mathsf{SPECIFIER}$*, die in dem Architekturspezifikation* as *beschrieben sind, wie folgt definiert:*

$$\begin{aligned}\mathsf{Specifier}_{\mathsf{as}} =_{\mathsf{def}} & \mathsf{L}\left(\langle \mathsf{SYSTEM_NAME}\rangle_{\mathsf{as}}\right) \cup \mathsf{L}\left(\langle \mathsf{COMPONENT_NAME}\rangle_{\mathsf{as}}\right) \cup \mathsf{L}\left(\langle \mathsf{INTERFACE_NAME}\rangle_{\mathsf{as}}\right) \cup \\ & \cup \mathsf{L}\left(\langle \mathsf{ATTRIBUTE_NAME}\rangle_{\mathsf{as}}\right) \cup \mathsf{L}\left(\langle \mathsf{CONNECTION_NAME}\rangle_{\mathsf{as}}\right) \cup \mathsf{L}\left(\langle \mathsf{MESSAGE_NAME}\rangle_{\mathsf{as}}\right) \cup \\ & \cup \mathsf{L}\left(\langle \mathsf{INVARIANT_NAME}\rangle_{\mathsf{as}}\right) \cup \mathsf{L}\left(\langle \mathsf{OCLX_BASIC_TYPE}\rangle_{\mathsf{as}}\right)\end{aligned}$$

Die Menge der Instanzen, die in einem System $\mathsf{sys} \in \mathsf{SYSTEM}$ direkt enthalten sind, werden im Systemmodell bereits durch $\mathsf{Instance}_{\mathsf{sys}}$ charakterisiert. Die folgenden Definitionen führen noch die transitive Hülle über die Super-/Subsystem-Beziehung in hierarchischen komponentenbasierten Systemen ein und legen so die Menge aller Instanzen $\widehat{\mathsf{Instance}}_{\mathsf{sys}}$ formal fest.

Definition 6.3: Instanzen eines Systems

Sei $\mathsf{sys} \in \mathsf{SYSTEM}$ *ein System aus dem Systemmodell, so bezeichnet die Menge* $\widehat{\mathsf{Instance}}_{\mathsf{sys}} \subseteq \mathsf{INSTANCE}$ *die Menge aller Instanzen in dem System* sys*:*

$$\widehat{\mathsf{System}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{s} \in \mathsf{SYSTEM} \,\middle|\, \mathsf{s} = \mathsf{sys} \vee \left(\mathsf{s} \in \mathsf{System}_{\mathsf{sub}} \wedge \mathsf{sub} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right) \right\}$$

$$\widehat{\mathsf{Component}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{c} \in \mathsf{Component}_{\mathsf{s}} \,\middle|\, \mathsf{s} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right\}$$

$$\widehat{\mathsf{Interface}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{i} \in \mathsf{Interface}_{\mathsf{s}} \,\middle|\, \mathsf{s} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right\}$$

$$\widehat{\mathsf{Attribute}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{a} \in \mathsf{Attribute}_{\mathsf{s}} \,\middle|\, \mathsf{s} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right\}$$

$$\widehat{\mathsf{Connection}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{c} \in \mathsf{Connection}_{\mathsf{s}} \,\middle|\, \mathsf{s} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right\}$$

$$\widehat{\mathsf{Instance}}_{\mathsf{sys}} =_{\mathsf{def}} \left\{ \mathsf{i} \in \mathsf{Instance}_{\mathsf{s}} \,\middle|\, \mathsf{s} \in \widehat{\mathsf{System}}_{\mathsf{sys}} \right\}$$

Jetzt können wir die eigentliche Formalisierung von Architekturspezifikationen beginnen. Hierfür ordnen wir Architekturspezifikation konkrete Eigenschaften aus der Menge $\mathsf{TERM}^{\mathsf{v}}$ zu. Diese Eigenschaften legen die Verbindung zwischen der Syntax, den Spezifikationsbezeichnern der Architekturspezifikation, und der Semantik, den Instanzen des Systemmodells, fest.

Definition 6.4: Von Instanzen zu den Spezifikationsbezeichnern

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gelten die folgenden Eigenschaften aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$*:*

$$\forall s \in \widehat{System_{sys}} \Rightarrow specified(s) \in L\left(\langle SYSTEM_NAME\rangle_{as}\right)$$

$$\forall c \in \widehat{Component_{sys}} \Rightarrow specified(c) \in L\left(\langle COMPONENT_NAME\rangle_{as}\right)$$

$$\forall i \in \widehat{Interface_{sys}} \Rightarrow specified(i) \in L\left(\langle INTERFACE_NAME\rangle_{as}\right)$$

$$\forall a \in \widehat{Attribute_{sys}} \Rightarrow specified(a) \in L\left(\langle ATTRIBUTE_NAME\rangle_{as}\right)$$

$$\forall c \in \widehat{Connection_{sys}} \Rightarrow specified(c) \in L\left(\langle CONNECTION_NAME\rangle_{as}\right)$$

Diese Eigenschaften legen fest, dass alle Instanzen einer korrekten Implementierung eindeutig einem Spezifikationsbezeichner zugeordnet sind. Da diese Eigenschaften bei einer gültigen Implementierung erfüllt sein müssen, erlaubt uns die Funktion $specified$ jeweils von den semantischen Elementen, den Instanzen des Systemmodells, zu den syntaktischen Elementen, den Spezifikationsbezeichnern der Architekturspezifikation, zu gelangen. Von dieser Möglichkeit werden wir im folgenden exzessiv Gebrauch machen.

6.3.2 Aufbau und Struktur von Komponentenspezifikationen

Kapitel 6.2.1 beschreibt den Aufbau und die Struktur von Komponentenspezifikationen. Die formale Fundierung dieses Teils der Syntax der Spezifikationstechnik konzentriert sich darauf sicherzustellen, dass während der Ausführung die Instanzen des Systems der spezifizierten Struktur genügen.

Dementsprechend muss der Spezifikationsbezeichner einer Schnittstelleninstanz, die einer Komponenteninstanz zugeordnet ist, auch in der Menge der zugesicherten Schnittstellen in der Komponentenbeschreibung enthalten sein. Darüber hinaus muss die Anzahl der aktiven Schnittstelleninstanzen, die einer Komponente zugewiesen sind, zu jedem Zeitpunkt innerhalb der spezifizierten minimalen und maximalen Instanzierungskardinalität liegen.

Definition 6.5: Struktur und Anzahl von Schnittstellen an Komponenten

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gelten die folgenden Eigenschaften aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$*:*

$$\begin{array}{l}
\forall t \in T, s \in \widehat{System_{sys}}, c \in Component_s, i \in Interface_s\,.\\
(c,i) \in assigned_s^t\\
\Rightarrow\\
\exists w \in L\left(\langle COMPONENT_SPECIFICATION\rangle_{as}\right).\\
w = \underline{\texttt{COMPONENT}}\ \ specified(c)\ \ \underline{\texttt{ASSURED}}\ \ \{\langle INTERFACE_SPECIFICATION\rangle\}^*\\
\quad \underline{\texttt{INTERFACE}}\ \ specified(i)\ \ \underline{[}\ \ \langle MINIMUM_CARDINALITY\rangle\ \ \underline{,}\ \ \langle MAXIMUM_CARDINALITY\rangle\ \ \underline{]}\\
\quad \{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*\ \ \{\langle CONNECTION_SPECIFICATION\rangle\}^*\ \ \{\langle MESSAGE_SPECIFICATION\rangle\}^*\\
\quad \{\langle INVARIANT_SPECIFICATION\rangle\}^*\ \ \{\langle INTERFACE_SPECIFICATION\rangle\}^*
\end{array}$$

$$\forall t \in T, s \in \widehat{System_{sys}}, c \in Component_s, in \in L\left(\langle INTERFACE_NAME\rangle_{as}\right).$$
$$alive_s^t(s) \wedge alive_s^t(c)$$
$$\Rightarrow$$
$$\exists n, m \in N, w \in L\left(\langle COMPONENT_SPECIFICATION\rangle_{as}\right).$$
$$n \leq \left|\left\{(c,i) \in assigned_s^t \middle| i \in Interfaces_s \wedge alive_s^t(i) \wedge specified(i) = in\right\}\right| \leq m \wedge$$
$$w = \underline{\texttt{COMPONENT}}\ specified(c)\ \underline{\texttt{ASSURED}}\ \{\langle INTERFACE_SPECIFICATION\rangle\}^*\ \underline{\texttt{INTERFACE}}\ in\ \underline{[}\ n\ \underline{,}\ m\ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*\ \{\langle CONNECTION_SPECIFICATION\rangle\}^*\ \{\langle MESSAGE_SPECIFICATION\rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION\rangle\}^*\ \{\langle INTERFACE_SPECIFICATION\rangle\}^*$$

Außerdem dürfen an Schnittstelleninstanzen nur solche Attributinstanzen existieren, die in der Schnittstellenspezifikation beschrieben sind. Drüber hinaus müssen aber bei einer Schnittstelleninstanz stets alle spezifizierten Attribute existieren.

Definition 6.6: Struktur und Anzahl von Attributen an Schnittstellen

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gelten die folgenden Eigenschaften aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$*:*

$$\forall t \in T, s \in \widehat{System_{sys}}, i \in Interfaces_s, a \in Attribute_s.$$
$$(i,a) \in allocation_s^t$$
$$\Rightarrow$$
$$\exists w \in L\left(\langle INTERFACE_SPECIFICATION\rangle_{as}\right).$$
$$w = \underline{\texttt{INTERFACE}}\ specified(i)\ \underline{[}\ \langle MINIMUM_CARDINALITY\rangle\ \underline{,}\ \langle MAXIMUM_CARDINALITY\rangle\ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*\ \underline{\texttt{ATTRIBUTE}}\ specified(a)\ \underline{:}\ \langle OCLX_BASIC_TYPE\rangle$$
$$\{\underline{\texttt{CALCULATED BY}}\ \langle ATTRIBUTE_CALCULATION_SPECIFICATION\rangle\}^?\ \{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*$$
$$\{\langle CONNECTION_SPECIFICATION\rangle\}^*\ \{\langle MESSAGE_SPECIFICATION\rangle\}^*\ \{\langle INVARIANT_SPECIFICATION\rangle\}^*$$

$$\forall t \in T, s \in \widehat{System_{sys}}, i \in Interfaces_s, w \in L\left(\langle INTERFACE_SPECIFICATION\rangle_{as}\right), an \in L\left(\langle ATTRIBUTE_NAME\rangle_{as}\right).$$
$$alive_s^t(s) \wedge alive_s^t(i) \wedge$$
$$w = \underline{\texttt{INTERFACE}}\ specified(i)\ \underline{[}\ \langle MINIMUM_CARDINALITY\rangle\ \underline{,}\ \langle MAXIMUM_CARDINALITY\rangle\ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*\ \underline{\texttt{ATTRIBUTE}}\ an\ \underline{:}\ \langle OCLX_BASIC_TYPE\rangle$$
$$\{\underline{\texttt{CALCULATED BY}}\ \langle ATTRIBUTE_CALCULATION_SPECIFICATION\rangle\}^?\ \{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*$$
$$\{\langle CONNECTION_SPECIFICATION\rangle\}^*\ \{\langle MESSAGE_SPECIFICATION\rangle\}^*\ \{\langle INVARIANT_SPECIFICATION\rangle\}^*$$
$$\Rightarrow$$
$$\exists a \in Attribute_s.$$
$$alive_s^t(a) \wedge (i,a) \in allocation_s^t \wedge specified(a) = an$$

Der Verbindungsbezeichner, wie auch der Name der rechten und linken Schnittstelleninstanz, einer Verbindungsinstanz muss der Schnittstellenbeschreibung entsprechen. Zusätzlich muss die Anzahl der Verbindungsinstanzen innerhalb der angegebenen Grenzen liegen.

Definition 6.7: Struktur und Anzahl von Verbindungen zwischen Schnittstellen

Sei $\mathsf{as} \in \mathsf{L}(\mathsf{ASG})$ *eine syntaktisch korrekte Architekturspezifikation, so gelten die folgenden Eigenschaften aus der Menge* $\mathsf{assured}(\mathsf{as})$ *bei jeder korrekten Implementierung* $\mathsf{sys} \in \mathsf{SYSTEM}$*:*

$$
\begin{array}{l}
\forall t \in \mathrm{T}, s \in \widehat{\mathsf{System}_{\mathsf{sys}}}, c \in \mathsf{Connection}_{\mathsf{s}}, i \in \mathsf{Interface}_{\mathsf{s}}, in \in \mathsf{L}\left(\left\langle \mathsf{INTERFACE_NAME} \right\rangle_{\mathsf{as}}\right). \\
\mathsf{alive}_{\mathsf{s}}^{\mathsf{t}}(s) \wedge \mathsf{alive}_{\mathsf{s}}^{\mathsf{t}}(c) \wedge \mathsf{alive}_{\mathsf{s}}^{\mathsf{t}}(i) \\
\Rightarrow \\
\exists n, m \in \mathrm{N}, w \in \mathsf{L}\left(\left\langle \mathsf{INTERFACE_SPECIFICATION} \right\rangle_{\mathsf{as}}\right). \\
n \le \left| \left\{ \{i,j\} \in \mathsf{connected}_{\mathsf{s}}^{\mathsf{t}}(c) \,\middle|\, \mathsf{alive}_{\mathsf{s}}^{\mathsf{t}}(j) \wedge \mathsf{specified}(j) = in \right\} \right| \le m \wedge \\
\quad w = \underline{\texttt{INTERFACE}}\ \ \mathsf{specified}(i)\ \ \underline{[}\ \ \langle \mathsf{MINIMUM_CARDINALITY} \rangle\ \ \underline{,}\ \ \langle \mathsf{MAXIMUM_CARDINALITY} \rangle\ \ \underline{]} \\
\qquad \{\langle \mathsf{ATTRIBUTE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^*\ \ \underline{\texttt{CONNECTION}}\ \ \mathsf{specified}(c) \\
\qquad \underline{\texttt{END}}\ \ \langle \mathsf{CONNECTION_END_NAME} \rangle\ \ \underline{:}\ \ in\ \ \underline{[}\ \ n\ \ \underline{,}\ \ m\ \ \underline{]}\ \ \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^* \\
\qquad \{\langle \mathsf{MESSAGE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{INVARIANT_SPECIFICATION} \rangle\}^*
\end{array}
$$

$$
\begin{array}{l}
\forall t \in \mathrm{T}, s \in \widehat{\mathsf{System}_{\mathsf{sys}}}, c \in \mathsf{Connection}_{\mathsf{s}}, i, j \in \mathsf{Interface}_{\mathsf{s}}. \\
\mathsf{connected}_{\mathsf{s}}^{\mathsf{t}}(c) = \{i, j\} \\
\Rightarrow \\
\exists w_1, w_2 \in \mathsf{L}\left(\left\langle \mathsf{INTERFACE_SPECIFICATION} \right\rangle_{\mathsf{as}}\right). \\
w_1 = \underline{\texttt{INTERFACE}}\ \ \mathsf{specified}(i)\ \ \underline{[}\ \ \langle \mathsf{MINIMUM_CARDINALITY} \rangle\ \ \underline{,}\ \ \langle \mathsf{MAXIMUM_CARDINALITY} \rangle\ \ \underline{]} \\
\quad \{\langle \mathsf{ATTRIBUTE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^*\ \ \underline{\texttt{CONNECTION}}\ \ \mathsf{specified}(c) \\
\quad \underline{\texttt{END}}\ \ \langle \mathsf{CONNECTION_END_NAME} \rangle\ \ \underline{:}\ \ \mathsf{specified}(j)\ \ \underline{[}\ \ \langle \mathsf{MINIMUM_CARDINALITY} \rangle\ \ \underline{,}\ \ \langle \mathsf{MAXIMUM_CARDINALITY} \rangle\ \ \underline{]} \\
\quad \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{MESSAGE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{INVARIANT_SPECIFICATION} \rangle\}^* \\
\wedge \\
w_2 = \underline{\texttt{INTERFACE}}\ \ \mathsf{specified}(j)\ \ \underline{[}\ \ \langle \mathsf{MINIMUM_CARDINALITY} \rangle\ \ \underline{,}\ \ \langle \mathsf{MAXIMUM_CARDINALITY} \rangle\ \ \underline{]} \\
\quad \{\langle \mathsf{ATTRIBUTE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^*\ \ \underline{\texttt{CONNECTION}}\ \ \mathsf{specified}(c) \\
\quad \underline{\texttt{END}}\ \ \langle \mathsf{CONNECTION_END_NAME} \rangle\ \ \underline{:}\ \ \mathsf{specified}(i)\ \ \underline{[}\ \ \langle \mathsf{MINIMUM_CARDINALITY} \rangle\ \ \underline{,}\ \ \langle \mathsf{MAXIMUM_CARDINALITY} \rangle\ \ \underline{]} \\
\quad \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{MESSAGE_SPECIFICATION} \rangle\}^*\ \ \{\langle \mathsf{INVARIANT_SPECIFICATION} \rangle\}^*
\end{array}
$$

Und schließlich, die letzte Bedingung, die sich aus der Struktur von Komponentenspezifikationen ableiten lässt, besagt, dass an Schnittstelleninstanzen nur Nachrichteninstanzen versendet werden dürfen, wenn der entsprechende Nachrichtenbezeichner bzw. Invariantenbezeichner in der Schnittstellenbeschreibung vorkommt.

Definition 6.8: Versenden von Nachrichten an Schnittstellen

Sei $\mathsf{as} \in \mathsf{L}(\mathsf{ASG})$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $\mathsf{assured}(\mathsf{as})$ *bei jeder korrekten Implementierung* $\mathsf{sys} \in \mathsf{SYSTEM}$*:*

$$\forall t \in T, s \in \widehat{System_{sys}}, i \in Interface_s, m \in M, a, b \in M^* .$$
$$alive_s^t(s) \wedge alive_s^t(i) \wedge evaluation_s^t(i) = a\ m\ b$$
$$\Rightarrow$$
$$\exists w \in L(\langle INTERFACE_SPECIFICATION \rangle_{as}) .$$
$$w = \underline{\texttt{INTERFACE}}\ \ specified(i)\ \ \underline{[}\ \ \langle MINIMUM_CARDINALITY \rangle\ \ \underline{,}\ \ \langle MAXIMUM_CARDINALITY \rangle\ \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION \rangle\}^*\ \ \{\langle CONNECTION_SPECIFICATION \rangle\}^*\ \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*$$
$$\underline{\texttt{MESSAGE}}\ \ specified(m)\ \ \underline{(}\ \ \{\langle MESSAGE_PARAMETER_NAME \rangle\ \ \underline{:}\ \ \langle OCLX_BASIC_TYPE \rangle\}^*_{\underline{,}}\ \ \underline{)}$$
$$\langle BEHAVIOR_SPECIFICATION \rangle\ \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*\ \ \{\langle INVARIANT_SPECIFICATION \rangle\}^*$$
$$\vee$$
$$w = \underline{\texttt{INTERFACE}}\ \ specified(i)\ \ \underline{[}\ \ \langle MINIMUM_CARDINALITY \rangle\ \ \underline{,}\ \ \langle MAXIMUM_CARDINALITY \rangle\ \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION \rangle\}^*\ \ \{\langle CONNECTION_SPECIFICATION \rangle\}^*\ \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION \rangle\}^*\ \ \underline{\texttt{INVARIANT}}\ \ specified(m)\ \ \underline{()}\ \ \langle INVARIANT_CONDITION_SPECIFICATION \rangle$$
$$\underline{\texttt{implies}}\ \ \langle BEHAVIOR_SPECIFICATION \rangle\ \ \{\langle INVARIANT_SPECIFICATION \rangle\}^*$$

6.3.3 Spezifikation von Attributwertberechnungen

Der Wert eines Attributes kann durch eine Berechnungsvorschrift bestimmt sein (vgl. Kapitel 6.2.2). Für die prädikatenbasierte Charakterisierung der formalen Semantik von Attributwertberechnungen verwenden wir eine Interpretationsfunktion. Diese berechnet das Ergebnis einer Berechnungsvorschrift bei einer konkreten Belegung mit einer Systeminstanz.

Definition 6.9: Interpretation von Attributwertberechnungen

Die Interpretation einer Attributwertberechnung $w \in L(\langle ATTRIBUTE_CALCULATION_SPECIFICATION \rangle_{as})$ *in einer Spezifikation* $as \in L(ASG)$*, im Kontext einer Schnittstelleninstanz* $i \in INTERFACE$ *bei einem gegebenen Systemzustand* $snapshot \in SNAPSHOT$*, ergibt einen primitiven Wert* $v \in VALUE$*, der durch die Funktion* $interpret(w,i,snapshot)$ *berechnet wird:*

$$interpret : (L(V,T,P,\langle ATTRIBUTE_CALCULATION_SPECIFICATION \rangle) \times INTERFACE \times SNAPSHOT) \rightarrow VALUE$$

Auf eine vollständige formale Spezifikation der Funktion $interpret$ wird im Rahmen dieser Arbeit verzichtet. Es sei aber auf die Arbeiten [Sche00] und [HDF00] verwiesen. Die folgenden Ausschnitte einer induktiven Formalisierung sollen als Beispiel genügen:

$$\forall as \in L(ASG), w \in L(\langle ATTRIBUTE_CALCULATION_SPECIFICATION \rangle_{as}), i \in INTERFACE,$$
$$snapshot = (alive, assigned, allocation, connected, valuation, evaluation) \in SNAPSHOT, n \in VALUE .$$
$$alive(i) \wedge w = \underline{result}\ \ \underline{:=}\ \ n\ \ \underline{;}$$
$$\Rightarrow$$
$$interpret(w,i,snapshot) = n$$

$$\forall as \in L(ASG), w \in L(\langle ATTRIBUTE_CALCULATION_SPECIFICATION \rangle_{as}), i \in INTERFACE,$$
$$snapshot = (alive, assigned, allocation, connected, valuation, evaluation) \in SNAPSHOT, a \in ATTRIBUTE .$$
$$alive(i) \wedge alive(a) \wedge (i,a) \in allocation \wedge (a,v) \in valuation \wedge w = \underline{result}\ \ \underline{:=}\ \ \underline{self.}\ \ a\ \ \underline{;}$$
$$\Rightarrow$$
$$interpret(w,i,snapshot) = v$$

$$\forall as \in L(ASG), w \in L\left(\langle \text{ATTRIBUTE_CALCULATION_SPECIFICATION} \rangle_{as}\right), i, j \in \text{INTERFACE},$$
$$\quad snapshot = (alive, assigned, allocation, connected, valuation, evaluation) \in \text{SNAPSHOT}, c \in \text{CONNECTION}.$$
$$alive(i) \wedge alive(j) \wedge alive(a) \wedge connected(\{i,j\}) = c \wedge (j,a) \in allocation \wedge (a,v) \in valuation \wedge w = \underline{result}\ \underline{:=}\ \underline{self.}\ c\ \underline{.}\ a\ \underline{;}$$
$$\Rightarrow$$
$$interpret(w, i, snapshot) = v$$

Mit Hilfe dieser Interpreterfunktion können wir jetzt die, für die Formalisierung der Attributwertberechnung, notwendige Eigenschaft angeben. Der Attributwert entspricht zu jedem Zeitpunkt dem durch die Funktion $interpret$ berechnetem Wert.

Definition 6.10: Attributwertberechnung

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in \text{SYSTEM}$*:*

$$\forall t \in T, s \in \widehat{System_{sys}}, i \in Interface_s, w_1 \in L\left(\langle \text{INTERFACE_SPECIFICATION} \rangle_{as}\right), a \in Attribute_s,$$
$$\quad w_2 \in L\left(\langle \text{ATTRIBUTE_CALCULATION_SPECIFICATION} \rangle_{as}\right).$$
$$alive_s^t(s) \wedge alive_s^t(i) \wedge alive_s^t(a) \wedge$$
$$\quad w_1 = \underline{\texttt{INTERFACE}}\ specified(i)\ \underline{[}\ \langle \text{MINIMUM_CARDINALITY} \rangle\ \underline{,}$$
$$\quad \langle \text{MAXIMUM_CARDINALITY} \rangle\ \underline{]}\ \{\langle \text{ATTRIBUTE_SPECIFICATION} \rangle\}^*\ \underline{\texttt{ATTRIBUTE}}\ specified(a)$$
$$\quad \underline{:}\ \langle \text{OCLX_BASIC_TYPE} \rangle\ \underline{\texttt{CALCULATED BY}}\ w_2\ \{\langle \text{ATTRIBUTE_SPECIFICATION} \rangle\}^*$$
$$\quad \{\langle \text{CONNECTION_SPECIFICATION} \rangle\}^*\ \{\langle \text{MESSAGE_SPECIFICATION} \rangle\}^*\ \{\langle \text{INVARIANT_SPECIFICATION} \rangle\}^*$$
$$\Rightarrow$$
$$\left(a, interpret\left(w_2, i, snapshot_s^t\right)\right) \in valuation_s^t$$

6.3.4 Spezifikation von Bedingungen in Invarianten

Komponentenschnittstellen beinhalten Invarianten. Eine Invariante besteht aus einer Bedingung und einer Verhaltensbeschreibung, die ausgeführt wird, wenn die Bedingung wahr ist (vgl. Kapitel 6.2.3). In Analogie zum vorhergehenden Kapitel verwenden wir für die prädikatenbasierte Charakterisierung der formalen Semantik eine Interpretationsfunktion. Diese berechnet das Ergebnis der Bedingungsauswertung von Invarianten bei einer konkreten Belegung mit einer Systeminstanz.

Definition 6.11: Interpretation von Bedingungen in Invarianten

Die Interpretation der Bedingung einer Invariante $w \in L\left(\langle \text{INVARIANT_CONDITION_SPECIFICATION} \rangle_{as}\right)$ *in einer Spezifikation* $as \in L(ASG)$*, im Kontext einer Schnittstelleninstanz* $i \in \text{INTERFACE}$ *bei einem gegebenen Systemzustand* $snapshot \in \text{SNAPSHOT}$*, ergibt einen logischen Wert* $b \in \text{BOOLEAN}$*, der durch die Funktion* $interpret(w, i, snapshot)$ *berechnet wird:*

$$interpret : \left(L\left(V, T, P, \langle \text{INVARIANT_CONDITION_SPECIFICATION} \rangle\right) \times \text{INTERFACE} \times \text{SNAPSHOT}\right) \rightarrow \text{BOOLEAN}$$

Auf eine vollständige formale Spezifikation der Funktion $interpret$ wird im Rahmen dieser Arbeit wiederum verzichtet. Es sei aber auf die Arbeiten [Sche00] und [HDF00] verwiesen. Die folgenden Ausschnitte einer induktiven Formalisierung sollen als Beispiel genügen:

$$\forall as \in L(ASG), w \in L(\langle INVARIANT_CONDITION_SPECIFICATION\rangle_{as}), i \in INTERFACE,$$
$$snapshot = (alive, assigned, allocation, connected, valuation, evaluation) \in SNAPSHOT, b \in BOOLEAN.$$
$$alive(i) \wedge w = b$$
$$\Rightarrow$$
$$interpret(w, i, snapshot) = b$$

$$\forall as \in L(ASG), w \in L(\langle INVARIANT_CONDITION_SPECIFICATION\rangle_{css}), i \in INTERFACE, a_1, a_2 \in ATTRIBUTE,$$
$$snapshot = (alive, assigned, allocation, connected, valuation, evaluation) \in SNAPSHOT, v_1, v_2 \in VALUE.$$
$$alive(i) \wedge alive(a_1) \wedge alive(a_2) \wedge (i, a_1) \in allocation \wedge (i, a_2) \in allocation \wedge (a_1, v_1) \in valuation \wedge (a_2, v_2) \in valuation \wedge$$
$$\wedge w = \underline{self.}\ a_1\ \underline{=}\ \underline{self.}\ a_2$$
$$\Rightarrow$$
$$interpret(w, i, snapshot) = (v_1 = v_2)$$

Mit Hilfe dieser Interpreterfunktion können wir jetzt die, für die Formalisierung der Bedingungsauswertung von Invarianten, notwendige Eigenschaft angeben. Ist die Invariantenbedingung gültig, so liegt an der Schnittstelleninstanz eine entsprechende Nachricht an, die dann zur Ausführung der Invariante führt.

Definition 6.12: Bedingungsauswertung von Invarianten

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$ *:*

$$\forall t \in T, s \in \widehat{System_{sys}}, i \in Interface_s, w_1 \in L(\langle INTERFACE_SPECIFICATION\rangle_{as}), in \in L(\langle INVARIANT_NAME\rangle_{as}),$$
$$w_2 \in L(\langle INVARIANT_CONDITION_SPECIFICATION\rangle_{as}), m \in M.$$
$$alive_s^t(s) \wedge alive_s^t(i) \wedge specified(m) = in \wedge$$
$$w_1 = \underline{\mathtt{INTERFACE}}\ \ specified(i)\ \ \underline{[}\ \ \langle MINIMUM_CARDINALITY\rangle\ \ \underline{,}\ \ \langle MAXIMUM_CARDINALITY\rangle\ \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION\rangle\}^*\ \ \{\langle CONNECTION_SPECIFICATION\rangle\}^*\ \ \{\langle MESSAGE_SPECIFICATION\rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION\rangle\}^*\ \ \underline{\mathtt{INVARIANT}}\ \ in\ \ \underline{()}\ \ w_2\ \ \underline{\mathtt{implies}}\ \ \langle BEHAVIOR_SPECIFICATION\rangle$$
$$\{\langle INVARIANT_SPECIFICATION\rangle\}^*$$
$$\Rightarrow$$
$$interpret(w_2, i, snapshot_s^t) \Leftrightarrow \exists a, b \in M^*.\ (i, a\ m\ b) \in evaluation_s^t$$

6.3.5 Verhaltensspezifikation von Invarianten und Nachrichten

Verhaltensspezifikationen beschreiben den erwarteten Effekt der Ausführung einer Invariante oder eine Nachricht: Das Ändern von Attributwerten, das Löschen und Erzeugen von Verbindungen, Schnittstellen und Komponenten, sowie das Versenden von Nachrichten (vgl. Kapitel 6.4.2). In Analogie zum vorhergehenden Kapitel verwenden wir für die prädikatenbasierte Charakterisierung der formalen Semantik eine Interpretationsfunktion.

Definition 6.13: Interpretation von Verhaltensbeschreibungen

Die Interpretation von Verhaltensbeschreibungen $w \in L(\langle BEHAVIOR_SPECIFICATION\rangle_{as})$ *in einer Spezifikation* $as \in L(ASG)$*, im Kontext einer Schnittstelleninstanz* $i \in INTERFACE$ *bei einem gegebenen Systemzustand* $snapshot \in SNAPSHOT$*, ergibt einen Schnappschuss* $snapshot' \in SNAPSHOT$*, der den geforderten Ausschnitt des nächsten Systemschnappschusses repräsentiert und durch die Funktion* $interpret(w, i, snapshot)$ *berechnet wird:*

$$\mathrm{interpret}: \left(L\left(V,T,P,\langle \mathrm{BEHAVIOR_SPECIFICATION}\rangle\right) \times \mathrm{INTERFACE} \times \mathrm{SNAPSHOT}\right) \rightarrow \mathrm{SNAPSHOT}$$

Auf eine vollständige formale Spezifikation der Funktion $\mathrm{interpret}$ wird im Rahmen dieser Arbeit wiederum verzichtet. Es sei aber auf die Arbeiten [Sche00] und [HDF00] verwiesen. Die folgenden Ausschnitte einer induktiven Formalisierung sollen als Beispiel genügen:

$$\begin{array}{l}
\forall as \in L(\mathrm{ASG}), w \in L\left(\langle \mathrm{BEHAVIOR_SPECIFICATION}\rangle_{as}\right), i \in \mathrm{INTERFACE}, a \in \mathrm{ATTRIBUTE}, \\
\quad \mathrm{snapshot} = (\mathrm{alive}, \mathrm{assigned}, \mathrm{allocation}, \mathrm{connected}, \mathrm{valuation}, \mathrm{evaluation}) \in \mathrm{SNAPSHOT}, v \in \mathrm{VALUE}. \\
\mathrm{alive}(i) \wedge \mathrm{alive}(a) \wedge (i,a) \in \mathrm{allocation} \wedge w = \underline{\mathrm{self}} \;\; \underline{.} \;\; a \;\; \underline{:=} \;\; v \;\; \underline{;} \\
\Rightarrow \\
\mathrm{interpret}(w, i, \mathrm{snapshot}) = \left(\varnothing, \varnothing, \varnothing, \varnothing, \{(a,v)\}, \varnothing\right)
\end{array}$$

$$\begin{array}{l}
\forall as \in L(\mathrm{ASG}), w \in L\left(\langle \mathrm{BEHAVIOR_SPECIFICATION}\rangle_{as}\right), i \in \mathrm{INTERFACE}, mn \in L\left(\langle \mathrm{MESSAGE_NAME}\rangle_{as}\right), \\
\quad \mathrm{snapshot} = (\mathrm{alive}, \mathrm{assigned}, \mathrm{allocation}, \mathrm{connected}, \mathrm{valuation}, \mathrm{evaluation}) \in \mathrm{SNAPSHOT}, m \in M. \\
\mathrm{alive}(i) \wedge \mathrm{specified}(m) = mn \wedge w = \underline{\mathrm{self.}} \;\; mn \;\; \underline{()} \;\; \underline{;} \\
\Rightarrow \\
\exists m \in M. \\
\mathrm{specified}(m) = mn \wedge \mathrm{interpret}(w, i, \mathrm{snapshot}) = \left(\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \{(i,m)\}\right)
\end{array}$$

Mit Hilfe dieser Interpreterfunktion können wir jetzt die, für die Formalisierung der Ausführung von Verhaltensspezifikationen, notwendige Eigenschaft angeben. Der Effekt der Verhaltensspezifikation wird dabei durch die Funktion $\mathrm{interpret}$ berechnet.

Definition 6.14: Ausführung von Invarianten und Nachrichten

Sei $as \in L(\mathrm{ASG})$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $\mathrm{assured}(as)$ *bei jeder korrekten Implementierung* $sys \in \mathrm{SYSTEM}$*:*

$$\begin{array}{l}
\forall t \in T, s \in \widehat{\mathrm{System}_{sys}}, c \in \mathrm{Component}_s, i \in \mathrm{Interface}_s, m \in M, a, b \in M^*, w \in L\left(\langle \mathrm{COMPONENT_SPECIFICATION}\rangle_{as}\right), \\
\quad bs \in L\left(\langle \mathrm{BEHAVIOR_SPECIFICATION}\rangle_{as}\right). \\
\mathrm{alive}_s^t(s) \wedge \mathrm{alive}_s^t(c) \wedge \mathrm{alive}_s^t(i) \wedge (i,m) \in \mathrm{evaluation}_s^t \wedge \\
\left(\begin{array}{l}
w = \underline{\mathrm{COMPONENT}} \;\; \mathrm{specified}(c) \;\; \underline{\mathrm{ASSURED}} \;\; \{\langle \mathrm{INTERFACE_SPECIFICATION}\rangle\}^* \\
\quad \underline{\mathrm{INTERFACE}} \;\; \mathrm{specified}(i) \;\; \underline{[} \;\; \langle \mathrm{MINIMUM_CARDINALITY}\rangle \;\; \underline{,} \;\; \langle \mathrm{MAXIMUM_CARDINALITY}\rangle \;\; \underline{]} \\
\quad \{\langle \mathrm{ATTRIBUTE_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{CONNECTION_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{MESSAGE_SPECIFICATION}\rangle\}^* \\
\quad \underline{\mathrm{MESSAGE}} \;\; \mathrm{specified}(m) \;\; \underline{(} \;\; \{\langle \mathrm{MESSAGE_PARAMETER_NAME}\rangle \;\; \underline{:} \;\; \langle \mathrm{OCLX_BASIC_TYPE}\rangle\}^*_{\underline{,}} \;\; \underline{)} \;\; bs \\
\quad \{\langle \mathrm{MESSAGE_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{INVARIANT_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{INTERFACE_SPECIFICATION}\rangle\}^* \\
\vee \\
w = \underline{\mathrm{COMPONENT}} \;\; \mathrm{specified}(c) \;\; \underline{\mathrm{ASSURED}} \;\; \{\langle \mathrm{INTERFACE_SPECIFICATION}\rangle\}^* \\
\quad \underline{\mathrm{INTERFACE}} \;\; \mathrm{specified}(i) \;\; \underline{[} \;\; \langle \mathrm{MINIMUM_CARDINALITY}\rangle \;\; \underline{,} \;\; \langle \mathrm{MAXIMUM_CARDINALITY}\rangle \;\; \underline{]} \\
\quad \{\langle \mathrm{ATTRIBUTE_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{CONNECTION_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{MESSAGE_SPECIFICATION}\rangle\}^* \\
\quad \{\langle \mathrm{INVARIANT_SPECIFICATION}\rangle\}^* \;\; \underline{\mathrm{INVARIANT}} \;\; \mathrm{specified}(m) \;\; \underline{()} \;\; \langle \mathrm{INVARIANT_CONDITION_SPECIFICATION}\rangle \\
\quad \underline{\mathrm{implies}} \;\; bs \;\; \{\langle \mathrm{INVARIANT_SPECIFICATION}\rangle\}^* \;\; \{\langle \mathrm{INTERFACE_SPECIFICATION}\rangle\}^*
\end{array}\right) \\
\Rightarrow \\
\exists (\mathrm{snapshot}_c, \mathrm{snapshot}'_c) \in \mathrm{behavior}_c. \\
\mathrm{snapshot}_c = \left(\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \{(i, a\,m\,b)\}\right) \wedge \mathrm{snapshot}'_c = \mathrm{interpret}\left(bs, i, \mathrm{snapshot}_s^t\right)
\end{array}$$

6.3.6 Spezifikation komponentenbasierter Systeme

Die Beschreibungstechnik für komponentenbasierter Systeme erlaubt es aus einzelnen Komponentenbeschreibungen Systembeschreibungen zu erstellen. Systemspezifikationen bestehen dabei aus einer Liste von Komponenten, die verwendet werden, einer Initialisierungsbeschreibung des Systems, und einer Beschreibung der Sichtbarkeitsregeln der Komponenten (vgl. Kapitel 6.2.5). Dementsprechend dürfen in einer Systeminstanz nur Komponenteninstanzen erzeugt werden, die als erlaubte Subkomponenten auch spezifiziert wurden.

Definition 6.15: Komponenteninstanzen in Systemen

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$ *:*

$$
\begin{aligned}
&\forall t \in T, s \in \widehat{System_{sys}}, c \in Component_s \,.\\
&alive_s^t(s) \wedge alive_s^t(c)\\
&\Rightarrow\\
&\exists w \in L\Big(\langle \text{SYSTEM_SPECIFICATION} \rangle_{as}\Big).\\
&w = \underline{\texttt{SYSTEM}}\ \ specified(s)\ \ \underline{\texttt{USED COMPONENTS}}\ \ \big\{\langle \text{COMPONENT_NAME} \rangle\big\}^{*}_{\underline{,}}\ \ specified(c)\ \ \underline{,}\\
&\quad \big\{\langle \text{COMPONENT_NAME} \rangle\big\}^{*}_{\underline{,}}\ \ \big\{\underline{\texttt{INITIALIZATION}}\ \ \langle \text{BEHAVIOR_SPECIFICATION} \rangle\big\}^{?}\\
&\quad \left\{\begin{array}{l}\underline{\texttt{FOR EACH INSTANCE OF}}\ \ \langle \text{COMPONENT_NAME} \rangle\ \ \underline{\texttt{EXISTS IMPLEMENTATION INSTANCE}}\\ \quad \langle \text{SYSTEM_NAME} \rangle\ \ \big\{\underline{\texttt{WITH VISIBILITY}}\ \ \langle \text{MAPPING_SPECIFICATION} \rangle\big\}^{*}_{\underline{;}}\end{array}\right\}^{*}
\end{aligned}
$$

Der Startzustand eines Systems zum Zeitpunkt $t = 0$ wird durch die folgende Eigenschaft festgelegt.

Definition 6.16: Startzustand eines Systems

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$ *:*

$$
\begin{aligned}
&\forall s \in \widehat{System_{sys}}, w \in L\Big(\langle \text{SYSTEM_SPECIFICATION} \rangle_{as}\Big), bs \in L\Big(\langle \text{BEHAVIOR_SPECIFICATION} \rangle_{as}\Big).\\
&w = \underline{\texttt{SYSTEM}}\ \ specified(s)\ \ \underline{\texttt{USED COMPONENTS}}\ \ \big\{\langle \text{COMPONENT_NAME} \rangle\big\}^{*}_{\underline{,}}\ \ \underline{\texttt{INITIALIZATION}}\ \ bs\\
&\quad \left\{\begin{array}{l}\underline{\texttt{FOR EACH INSTANCE OF}}\ \ \langle \text{COMPONENT_NAME} \rangle\ \ \underline{\texttt{EXISTS IMPLEMENTATION INSTANCE}}\\ \quad \langle \text{SYSTEM_NAME} \rangle\ \ \big\{\underline{\texttt{WITH VISIBILITY}}\ \ \langle \text{MAPPING_SPECIFICATION} \rangle\big\}^{*}_{\underline{;}}\end{array}\right\}\\
&\Rightarrow\\
&snapshot_s^0 = interpret(bs, \varepsilon, \varnothing)
\end{aligned}
$$

Außerdem beschreiben Systemspezifikationen für welche Subkomponenten bei der Instanzierung gleichzeitig entsprechende Subsysteminstanzen erzeugt werden sollen, damit die Realisierung der Subkomponente gewährleistet ist. Insbesondere müssen dabei die spezifizierten Sichtbarkeitsregeln in die Sichtbarkeitsrelation des Systemmodells aufgenommen werden.

Definition 6.17: Subsysteme eines Supersystems

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $assured(as)$ *bei jeder korrekten Implementierung* $sys \in SYSTEM$ *:*

$$\forall s \in \widehat{\mathrm{System}_{sys}}, c \in \mathrm{HirarchicalComponent}_s, w \in L\left(\langle \mathrm{SYSTEM_SPECIFICATION} \rangle_{as}\right), sn \in L\left(\langle \mathrm{SYSTEM_NAME} \rangle_{as}\right),$$
$$ms \in L\left(\left\{\langle \mathrm{MAPPING_SPECIFICATION} \rangle\right\}^{*}_{\underline{;}\,as}\right).$$
$$w = \underline{\texttt{SYSTEM}} \quad \mathrm{specified}(s) \quad \underline{\texttt{USED COMPONENTS}} \quad \left\{\langle \mathrm{COMPONENT_NAME} \rangle\right\}^{*}_{\underline{,}} \quad \mathrm{specified}(c)$$
$$\left\{\langle \mathrm{COMPONENT_NAME} \rangle\right\}^{*}_{\underline{,}} \quad \left\{\underline{\texttt{INITIALIZATION}} \quad \langle \mathrm{BEHAVIOR_SPECIFICATION} \rangle\right\}^{?}$$
$$\left\{\begin{matrix}\underline{\texttt{FOR EACH INSTANCE OF}} \quad \langle \mathrm{COMPONENT_NAME} \rangle \quad \underline{\texttt{EXISTS IMPLEMENTATION INSTANCE}} \\ \langle \mathrm{SYSTEM_NAME} \rangle \quad \left\{\underline{\texttt{WITH VISIBILITY}} \quad \langle \mathrm{MAPPING_SPECIFICATION} \rangle\right\}^{*}_{\underline{;}}\end{matrix}\right\}^{*}$$
$$\underline{\texttt{FOR EACH INSTANCE OF}} \quad \mathrm{specified}(c) \quad \underline{\texttt{EXISTS IMPLEMENTATION INSTANCE}} \quad sn \quad \underline{\texttt{WITH VISIBILITY}} \quad ms$$
$$\left\{\begin{matrix}\underline{\texttt{FOR EACH INSTANCE OF}} \quad \langle \mathrm{COMPONENT_NAME} \rangle \quad \underline{\texttt{EXISTS IMPLEMENTATION INSTANCE}} \\ \langle \mathrm{SYSTEM_NAME} \rangle \quad \left\{\underline{\texttt{WITH VISIBILITY}} \quad \langle \mathrm{MAPPING_SPECIFICATION} \rangle\right\}^{*}_{\underline{;}}\end{matrix}\right\}^{*}$$
$$\Rightarrow$$
$$\exists sub \in \mathrm{System}_s .$$
$$\mathrm{specified}(sub) = sn \wedge \mathrm{implements}(c) = sub$$

$$\wedge$$

$$\forall n \in \mathbb{N}, me_i \in L\left(\langle \mathrm{MAPPING_SPECIFICATION} \rangle_{as}\right).$$
$$i \in \mathbb{N} \wedge 1 \le i \le n \wedge ms = me_1 \ \underline{;} \ \ldots \ me_n \ \underline{;}$$
$$\Rightarrow$$
$$me_i = t_1 \ \underline{\rightarrow} \ t_2 \Leftrightarrow \forall i_1 \in \mathrm{Instance}_s . \mathrm{specified}(i_1) = t_1 \Rightarrow \exists i_2 \in \mathrm{Instance}_{sub} . \mathrm{specified}(i_2) = t_2 \wedge (i_1, i_2) \in \mathrm{InstanceMapping}_{(s,c,sub)}$$
$$\vee$$
$$me_i = t_1 \ \underline{\leftarrow} \ t_2 \Leftrightarrow \forall i_2 \in \mathrm{Instance}_{sub} . \mathrm{specified}(i_2) = t_2 \Rightarrow \exists i_1 \in \mathrm{Instance}_s . \mathrm{specified}(i_1) = t_1 \wedge (i_2, i_1) \in \mathrm{InstanceMapping}_{(s,c,sub)}$$

Schließlich enthält jede Spezifikation eines komponentenbasierten Systems eine Instanzierung des Wurzelsystems. Somit steht mit dem Start immer eine Systeminstanz zur Verfügung.

Definition 6.18: Instanzierung des Wurzelsystems

Sei $as \in L(ASG)$ *eine syntaktisch korrekte Architekturspezifikation, so gilt die folgende Eigenschaft aus der Menge* $\mathrm{assured}(as)$ *bei jeder korrekten Implementierung* $sys \in \mathrm{SYSTEM}$*:*

$$\forall sn \in L\left(\langle \mathrm{SYSTEM_NAME} \rangle_{as}\right).$$
$$\mathrm{specified}(sn) = sys$$
$$\Rightarrow$$
$$as = \underline{\texttt{SYSTEM INSTANCE}} \quad sys \quad \underline{\texttt{OF SYSTEM}} \quad sn \quad \left\{\langle \mathrm{SYSTEM_SPECIFICATION} \rangle\right\}^{+} \quad \left\{\langle \mathrm{COMPONENT_SPECIFICATION} \rangle\right\}^{*}$$

Damit sind die einzelnen Bestandteile von Architekturspezifikationen komponentenbasierter Systeme aus Kapitel 6.2 mit Hilfe der prädikatenbasierten formalen Semantik vollständig formalisiert. Die drei $\mathrm{interpret}$ Funktionen wurden dabei nur exemplarisch formalisiert. Ein weitergehende Betrachtung der Formalisierung ist bei den Arbeiten [Sche00] und [HDF00] zu finden. Für eine Implementierung eines entsprechenden Werkzeuges ist diese Tiefe der Formalisierung ausreichend, wie wir in Kapitel 8 zeigen werden.

6.4 UML-basierte grafische Spezifikationstechnik

In den letzten Jahren, angetrieben durch die Objektorientierung und die Verbesserungen in der Werkzeugunterstützung, haben sich grafische Beschreibungs-, Modellierungs- und Spezifikationstechniken verstärkt durchgesetzt. Letztlich mündete dieser Trend in der Verbreitung und

Etablierung der UML in Forschung und Industrie als der internationale Standard für objektorientierte Modellierung [BJR98, RJB98, OMG00a]. Die UML hat zu einer Vereinheitlichung der Notation und punktuell auch zu einem besseren Verständnis der Semantik von Modellierungselementen geführt.

Gerade im Hinblick auf die praktischen Relevanz und die neuesten Entwicklungen rund um die UML ist es sinnvoll eine UML-basierte grafische Variante der textbasierten Spezifikationstechnik zur Verfügung zu stellen, die wir in den voranstehenden Kapiteln erarbeitet haben. Hierfür verwenden wir die, von der UML vorgesehenen, Mechanismen zur Erweiterung der UML. Über sogenannte Stereotypen können neue Metamodellelemente in das Metamodell der UML integriert werden. Mit Hilfe von Tagged Values und Constraints lassen sich existierende sowie neue Metamodellelemente mit zusätzlichen Eigenschaften und Bedingungen anreichern (vgl. auch [DSB99] und [KRSW01]).

In den folgenden Unterkapiteln stellen wir die, von uns erarbeitete, UML-basierte grafische Beschreibungstechnik für Softwarearchitekturen komponentenbasierter Systeme vor. Eine Formalisierung dieser Beschreibungstechnik würde einerseits den Rahmen dieser Arbeit sprengen, andererseits keine neuen, wesentlichen Erkenntnisse erzielen. Darüber hinaus lassen sich bereits mit dem in Kapitel 8 vorgestellten Werkzeug aus den UML Beschreibungen textbasierte Spezifikationen generieren. Aus diesem Grund beschränken wir uns in diesem Kapitel darauf, exemplarisch die UML-basierte Beschreibungstechnik vorzustellen. Dabei leiten wir aus der textbasierten Spezifikation des Pausenplaners in Kapitel 6.1 eine UML-basierte Beschreibung ab.

Kapitel 6.4.1 illustriert mit Hilfe von Komponentendiagrammen grafisch die Struktur komponentenbasierter Systeme. Im nächsten Kapitel 6.4.2 werden Verhaltensbeschreibungen in diese Komponentendiagramme integriert. Systemdiagramme mit Instanzierungsbeschreibungen in Kapitel 6.4.3 zeigen die grafische Spezifikation von Systemen, die aus einzelnen Komponenten zusammengesetzt sind, und deren Initialisierung. Abschließend, in Kapitel 6.4.4 skizzieren wir noch Sequenzdiagramme und deren methodische Verwendung im Rahmen des evolutionären Architekturentwurfs.

6.4.1 Komponentendiagramme

Komponentendiagramme sind spezielle statische UML Klassendiagramme, in denen nur Modellierungselemente mit den Stereotypen `«component»`, `«assured»`, `«interface»`, `«attribute»`, `«message»`, `«invariant»` und `«connection»` verwendet werden. Komponenten und Schnittstellen werden dabei als UML Klassen mit den entsprechenden Stereotypen `«component»` und `«interface»` dargestellt. Eine von einer Komponenten zur Verfügung gestellte Schnittstelle, wird durch eine gerichtete Assoziation mit dem Stereotyp `«assured»` repräsentiert. Am Ende der Assoziation ist die minimale und maximale Instanzierungskardinalität der Schnittstelle vermerkt. So hat die Komponente `StaffOrganizer`, wie in Abbildung 6.4 dargestellt, stets genau eine Instanz der Schnittstelle `StaffManager` und beliebig viele Instanzen der Schnittstelle `Teacher`.

Das Komponentendiagramm in Abbildung 6.4 illustriert die UML-basierte Beschreibungstechnik für die statische Struktur komponentenbasierter Systeme. Das Komponentendiagramm entspricht dabei direkt den textbasierten Beschreibungen aus Kapitel 6.1.

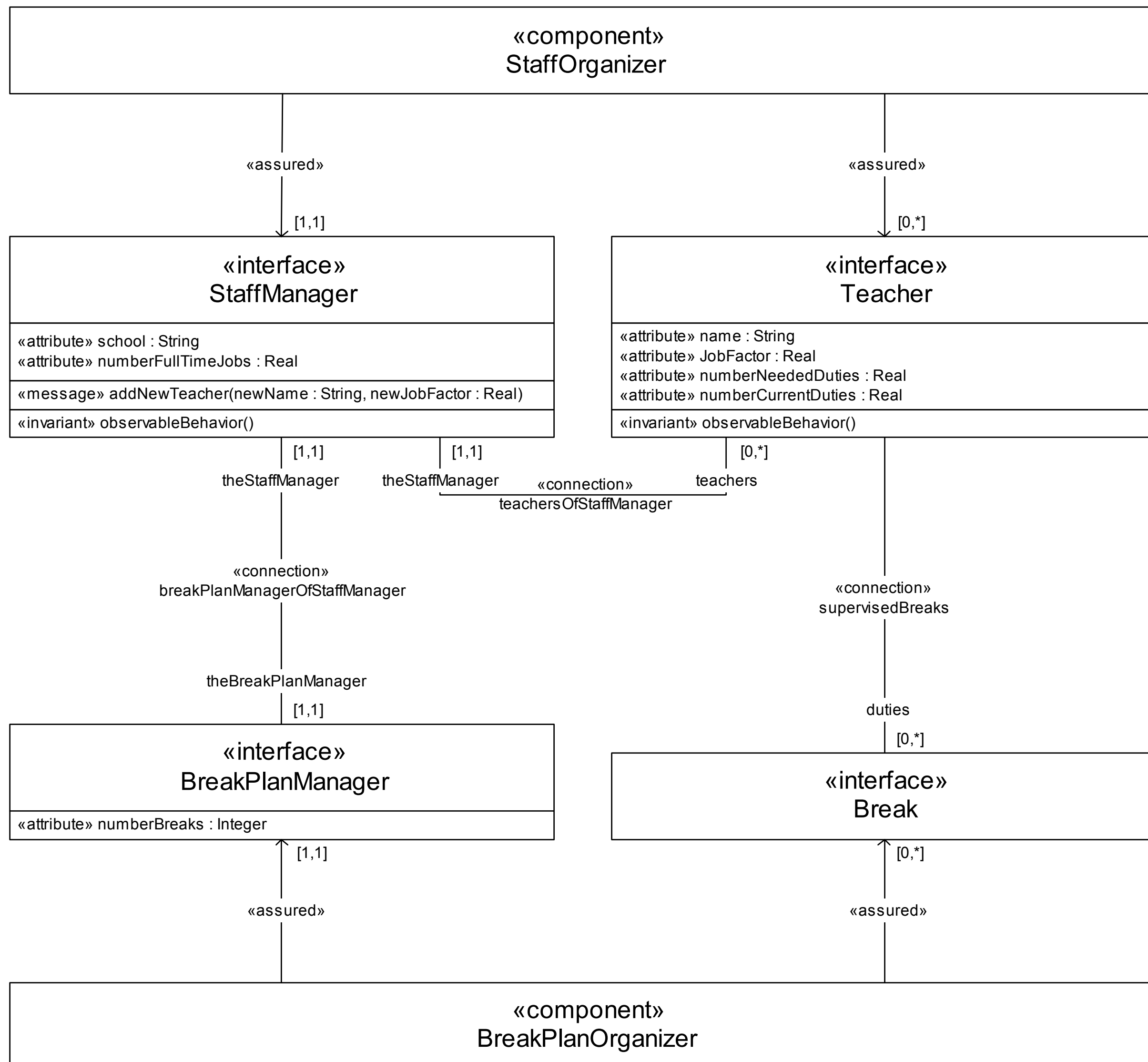

Abbildung 6.4: UML-basiertes Komponentendiagramm

6.4.2 Verhaltensbeschreibung in Komponentendiagrammen

Neben der Strukturbeschreibung enthalten die Spezifikationen in Kapitel 6.1 auch die Beschreibung des Verhaltens eines komponentenbasierten Systems. In Komponentendiagrammen können diese Verhaltensbeschreibungen integriert werden, wie Abbildung 6.5 zeigt.

Die Stereotypen «`calculated by`», «`invariant condition`» und «`behavior description`» kennzeichnen bei Attributen, Nachrichten und Invarianten die zusätzlichen Verhaltensspezifikationen. In Komponentendiagrammen können diese Beschreibungen direkt verwendet und dargestellt werden. Abbildung 6.5 zeigt einen Ausschnitt des Komponentendiagramms aus Abbildung 6.4, der eine vollständige Verhaltensspezifikation der Schnittstelle `StaffManager` enthält. Gekennzeichnet durch die entsprechenden Stereotypen beinhaltet die Schnittsstellenspezifikation Verhaltensbeschreibungen für das Attribut `numberFullTimeJobs`, die Nachricht `addNewTeacher` und die Invariante `observableBehavior`.

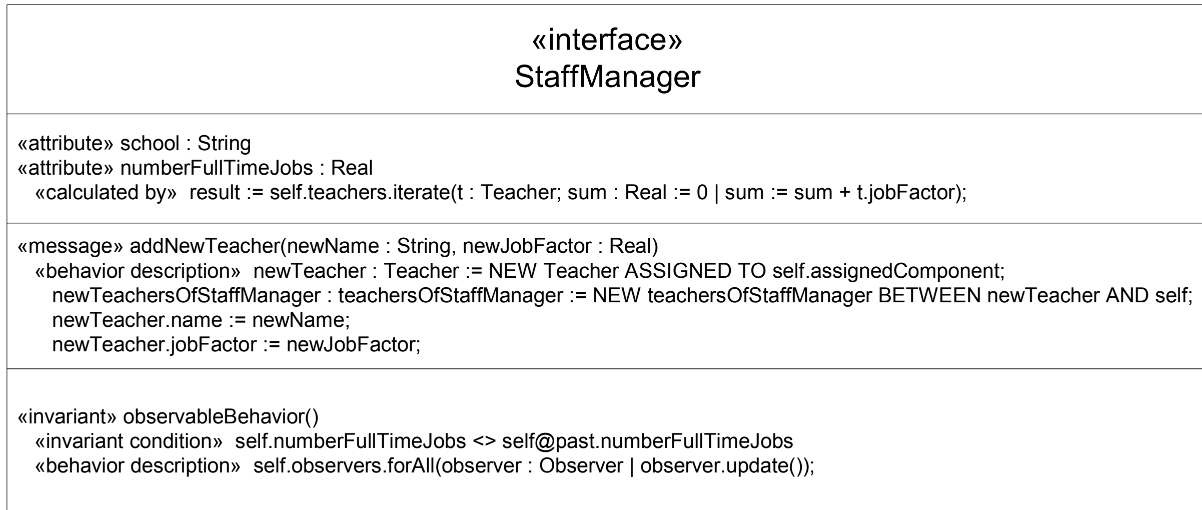

Abbildung 6.5: Komponentendiagramm mit integrierter Verhaltensbeschreibung

6.4.3 Systemdiagramme mit Instanzierungsbeschreibungen

Hat der Architekt mit Hilfe von Komponentendiagrammen alle Komponenten spezifiziert, so kann er mit Systemdiagrammen aus diesen Komponenten komponentenbasierte Systeme formen. Ein Systemdiagramm beschreibt die verwendeten Komponenten und die Instanzierung des Systems.

Abbildung 6.6 zeigt in einem UML-basiertes Systemdiagramm die grafische Repräsentation der textbasierten Spezifikation des Pausenplaners aus Kapitel 6.1. Komponentenbasierte Systeme werden hierbei als UML Pakete mit dem Stereotyp «`system`» dargestellt. Analog zu textbasierten Spezifikationen kann jedes System aus zwei weiteren Bestandteilen bestehen, eingeleitet durch die Stereotypen «`used components`» und «`initialization`».

Innerhalb der ersten Sektion werden die Komponenten aufgeführt, die in dem System verwendet werden. Für jede Komponente kann spezifiziert werden, welches Subsystem diese Komponente realisiert, eingeleitet durch den Stereotyp «`implemented by`». Über den Stereotyp «`mapping specification`» wird die Abbildung zwischen dem Super- und dem Subsystem definiert und somit die Sichtbarkeitsregeln der Komponente festgelegt. In Abbildung 6.6 beispielsweise wird jede Instanz der Komponente `StaffOrganizer` durch eine Instanz des Systems `StaffOrganizerImpl` implementiert. Dabei gelten die im Diagramm spezifizierten Sichtbarkeitsregeln, die in Kapitel 5.5.2 bereits vorgestellt und diskutiert wurden.

Die zweiten Sektion der Systemspezifikation, gekennzeichnet durch den Stereotyp «`initialization`», beinhaltet ein spezielles statisches UML Instanzendiagramm, das den Initialisierungszustand einer Systeminstanz beschreibt. Diese Sektion enthält Instanzen, die wiederum nur die Stereotypen «`component`», «`assured`», «`interface`», «`attribute`», «`message`», «`invariant`» und «`connection`» haben. Außerdem muss dieses Instanzendiagramm, wie jedes Instanzendiagramm in UML, den strukturellen Vorgaben seines entsprechenden Komponentendiagramms genügen. In unserem Beispiel zeigt das Instanzendiagramm in Abbildung 6.6 eine gültige Instanzierung des Komponentendiagramms aus Abbildung 6.4.

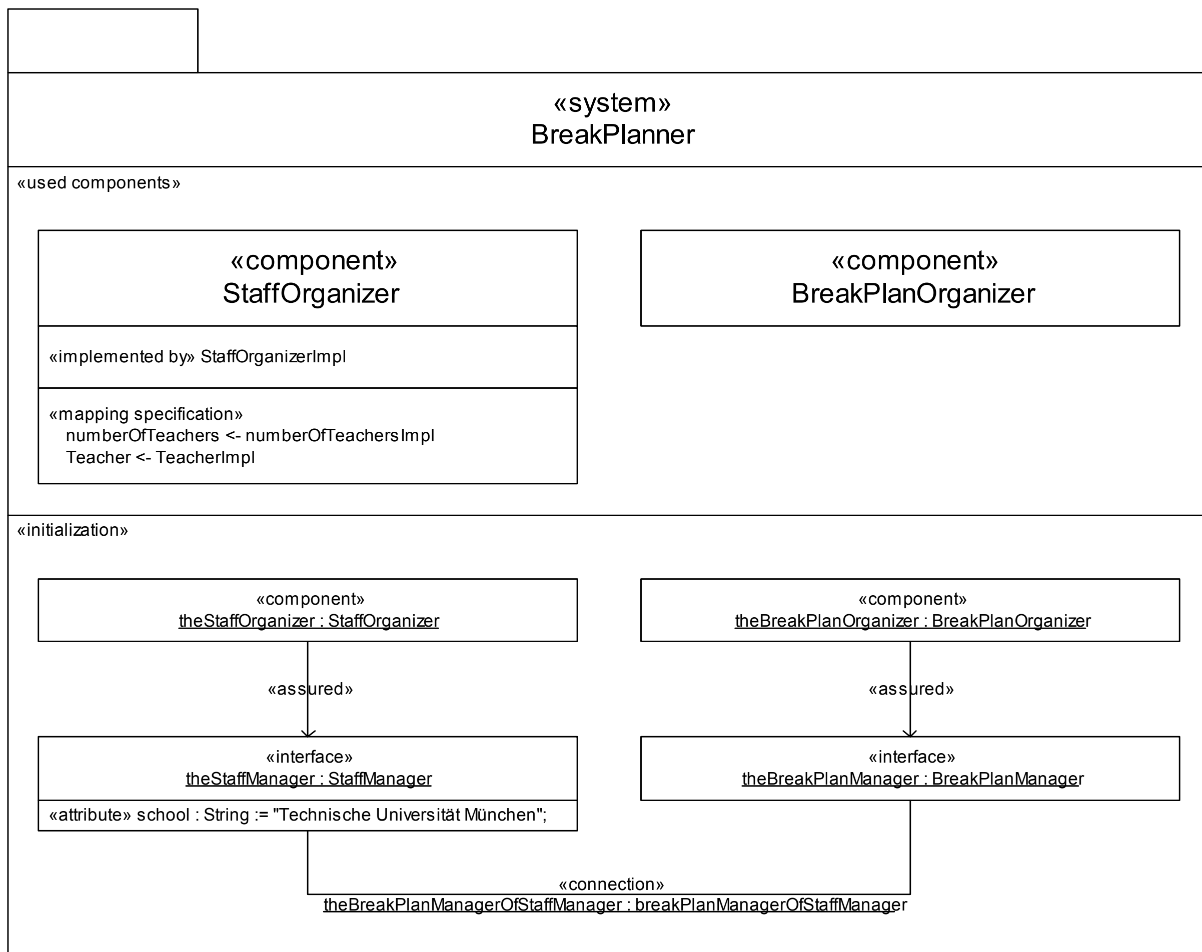

Abbildung 6.6: UML-basiertes Systemdiagramm

6.4.4 Sequenzdiagramme mit Ablaufbeschreibungen

Neben den bereits vorgestellten statischen UML Klassen- und Instanzendiagrammen erfreuen sich Sequenzdiagramme zunehmender Beliebtheit. Allerdings unterscheidet sich in der Praxis der methodischer Einsatz von Sequenzdiagrammen noch sehr stark. So werden Sequenzdiagramme einerseits dazu verwendet um exemplarische Abläufe zu beschreiben [BRS97], andererseits aber auch um daraus vollständige Systemspezifikationen abzuleiten [Krüg00].

Unsere textbasierte Spezifikationssprache für komponentenbasierter Systeme beinhaltet keine Repräsentation von Sequenzdiagrammen. Auf Grund der verbreiteten Anwendung von Sequenzdiagrammen in der Praxis setzen wir uns aber dennoch mit dieser Diagrammart am Rande auseinander. Abbildung 6.7 zeigt eine entsprechende Erweiterung der UML Sequenzdiagramme, die uns im Rahmen des evolutionären Architekturentwurfs sinnvoll erscheint.

Dieses Sequenzdiagramm besteht zuerst aus einem Instanzendiagramm, das den Systemzustand zum Startzeitpunkt der beschriebenen Sequenz festlegt. Dann folgt eine beliebige Folge von Ausführungsschritten jeweils getrennt durch gestichelte horizontale Linien. Innerhalb eines Ausführungsschrittes können Attributwerte verändert, Nachrichten verschickt, Invarianten ausgelöst sowie Instanzen erzeugt und gelöscht werden. Am Ende des Sequenzdiagramms wird unter Verwendung eines weiteren Instanzendiagramms der Systemzustand nach einer korrekten Ausführung der Ablaufsequenz spezifiziert.

Abbildung 6.7 zeigt ein Sequenzdiagramm, in dem die Nachricht `addNewTeacher` an die Schnittstelle `StaffManager` geschickt wird. Durch die Verarbeitung der Nachricht wird eine neue Lehrerschnittstelle erzeugt. Dem Observer Pattern entsprechend wird die Nachricht `update` an alle Observer verschickt. Am Schluss des Sequenzdiagramms ist der Zustand des Systems nach Ausführung der spezifizierten Schritte beschrieben.

Wie bereits angesprochen ist der methodische Einsatz von Sequenzdiagrammen an vielen Stellen noch nicht einheitlich festgelegt. Im Rahmen des evolutionären Architekturentwurfs bieten sich aber zwei Varianten an: Einerseits können Sequenzdiagramme dazu verwendet werden, um das Verhalten von Komponenten und deren Schnittstellen zu spezifizieren. Wir werden dies im Rahmen dieser Arbeit nicht näher untersuchen und verweisen hierbei aber auf [Krüg00]. Andererseits können Sequenzdiagramme sehr nützlich sein, um Testfälle für die modellierte Softwarearchitektur eines komponentenbasierten Systems zu spezifizieren. Im Kapitel 8 werden wir dieses Thema noch eingehender diskutieren.

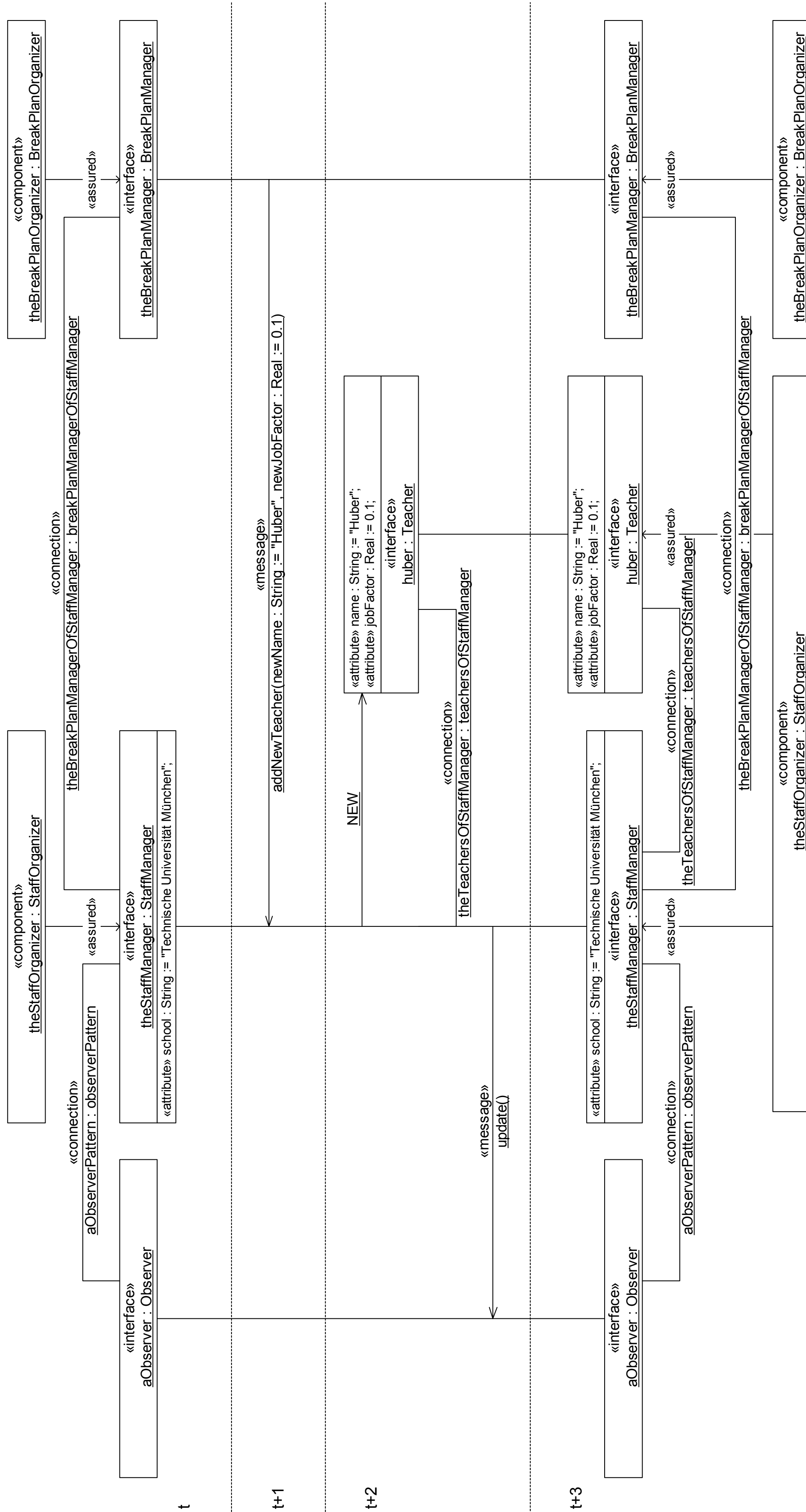

Abbildung 6.7: Sequenzdiagramm eines komponentenbasierten Systems

6.5 Zusammenfassung

Für die Modellierung und Spezifikation von Softwarearchitekturen komponentenbasierter Systeme sind adäquate Beschreibungstechniken zentral. Nur so ist es dem Architekten möglich eine Softwarearchitektur auf dem notwendigen Abstraktionsniveau zu beschreiben, ohne dabei wesentliche Elemente der Architektur zu vernachlässigen. Die in Forschung und Industrie bekannten und eingesetzten Beschreibungstechniken weisen hier noch eklatante Defizite auf.

Aus diesem Grund haben wir eine neuartige, textbasierte Beschreibungstechnik für komponentenbasierte Systeme entworfen, die speziell auf die Bedürfnisse des Architekturentwurfs zugeschnitten ist. Die vorgestellte Grammatik für Architekturspezifikationen unterteilt sich dabei in zwei wesentliche Spezifikationsarten: Komponentenspezifikationen und Systemspezifikationen.

In Komponentenspezifikationen werden einzelne Komponenten beschrieben, die zugesicherten Schnittstellen und die darin enthaltenen Attribute, Methoden und Invarianten. Die Teile der Spezifikation, die Verhalten beschreiben, sind stark an die Sprache OCL angelehnt. Dabei wurde OCL so erweitert, dass das Versenden von Nachrichten und das Erzeugen und Löschen von Komponenten, Schnittstellen sowie Verbindungen formuliert werden kann.

Systemspezifikationen dagegen beschreiben die hierarchische Struktur komponentenbasierter Systeme. Diese beinhaltet die Komponenten aus denen ein System zusammengesetzt ist und den Initialisierungszustand des Systems. Darüber hinaus wird die Zuordnung der hierarchischen Komponenten zu ihren Subsystemen und die dabei gültigen Sichtbarkeitsregeln festgelegt.

Damit die spezifizierten Modelle eindeutig und maschinell verarbeitbar sind, wurden diese Beschreibungstechniken auf Basis des Systemmodells aus Kapitel 5 formal fundiert. Dabei haben wir uns den grundlegenden Konzepten einer prädikatenbasierten formalen Semantik aus Kapitel 3 bedient.

Aus Gründen der praktischen Relevanz wurde zusätzlich eine grafische, UML-basierte Variante der textbasierten Beschreibungstechnik entwickelt. Auf eine formale Fundierung der UML-basierte Beschreibungstechnik oder einer Abbildung der grafischen in die textbasierte Beschreibungstechnik haben wir verzichtet. Anhand der Spezifikation des Pausenplaners wurde die Umsetzung der textbasierten in die UML-basierte Spezifikation exemplarisch vorgeführt. Im Kapitel 8 werden wir ein Werkzeug vorstellten, das aus den UML-basierten zuerst textbasierte Spezifikationen erzeugt und dann daraus ablauffähige Architekturprototypen generiert.

7 Erweiterte Spezifikationen für den evolutionären Entwurf

Mit Hilfe der Spezifikationstechniken, die im vorhergehenden Kapitel 6 erarbeitet wurden, lassen sich Architekturen komponentenbasierter Systeme vollständig und präzise beschreiben. Aus einem entsprechenden Architekturmodell können bereits ausführbare Prototypen generiert werden. Bei der Ausführung dieser Prototypen werden typischerweise eine Vielzahl von Fehlern, Verbesserungsmöglichkeiten und Erweiterungspotentialen identifiziert und diskutiert. Teilweise werden diese dann in der nächsten Evolutionsstufe der Modellierung berücksichtigt und dementsprechend einige der bestehenden Komponentenspezifikationen verändert.

Lokale Änderungen und Erweiterungen an Komponentenspezifikationen können jedoch dazu führen, dass die Gesamtfunktionalität des Systems wesentlich gestört, das System unter Umständen sogar funktionsuntüchtig und somit der Projektfortschritt negativ beeinflusst wird. Hintergrund ist meist, dass die Spezifikation und die komplexen Zusammenhänge zwischen den Komponenten in ihrer Gesamtheit von den einzelnen Bearbeitern nicht mehr durchdrungen werden können. Deshalb nehmen Softwareingenieure beim Entwurf und bei der Realisierung implizit vereinfachende Annahmen von Teilen des Systems an. Treffen diese Annahmen nicht zu, so führt dies zu Problemen und Fehlern.

Ursächlich ist aber die fehlende Möglichkeit in den gängigen Modellierungssprachen Annahmen und Abhängigkeiten zwischen den einzelnen Teilen eines Softwaresystems, den Komponenten, explizit und präzise formulieren zu können. Die UML beispielsweise bietet hier nur die Relation „uses“ an. In Java steht den Programmierern das Schlüsselwort „import“ zur Verfügung. Notwendig wären jedoch wesentlich weitreicherende Konzepte, eingebettet in die Spezifikationssprache.

In diesem Kapitel erarbeiten wir eine entsprechende Erweiterung der Spezifikationstechnik aus Kapitel 6. Mit diesen erweiterten Spezifikationen können einerseits vollständige und unabhängige Komponentenspezifikationen entworfen und andererseits explizite und eigenständige Spezifikationen der Abhängigkeiten zwischen den Komponenten entwickelt werden. Diese erweiterte Modellierungstechnik erlaubt es, frühzeitig durch lokale Änderungen an Komponentenspezifikationen verursachte Fehler in der Systemarchitekturspezifikation zu identifizieren und darauf entsprechend zu reagieren.

Ein erstes Anschauungsbeispiel in Kapitel 7.1 bringt uns der grundsätzlichen Problematik näher, die mit den existierenden Modellierungstechniken verbunden ist. Im nächsten Kapitel 7.2 stellen wir dann anhand des diskutierten Evolutionsszenarios eine verbesserte, erweiterte Spezifikationstechnik vor. Dabei wird deutlich, dass diese Spezifikationstechnik es ermöglicht, bestimmte Integrationsprobleme bereits im Vorfeld zu identifizieren und zu umgehen. Im anschließenden Kapitel 7.3 präsentieren wir die wesentlichen Bestandteile der Grammatik der erweiterten Spezifikationstechnik. Die formale Fundierung im Kapitel 7.4 sorgt für eine präzise Semantik der erweiterten Modellierungstechnik. In Analogie zum vorhergehenden Kapitel präsentieren wir im Kapitel 7.5 noch eine UML-basierte grafische Darstellung der zuvor erweiterten und formal fundierten textbasierten Spezifikationstechnik.

7.1 Evolution einer Architekturspezifikation – Ein Beispiel

Die Spezifikationstechnik aus Kapitel 6 erlaubt es uns die Architektur von komponentenbasierten Systemen vollständig und präzise zu beschreiben. Im Laufe des evolutionären Entwurfs werden einzelne Komponentenspezifikationen weiter entwickelt. Diese lokalen Änderungen und Erweiterungen können dazu führen, dass die Systemarchitekturspezifikation nicht mehr konsistent ist und das System unter Umständen funktionsuntüchtig wird.

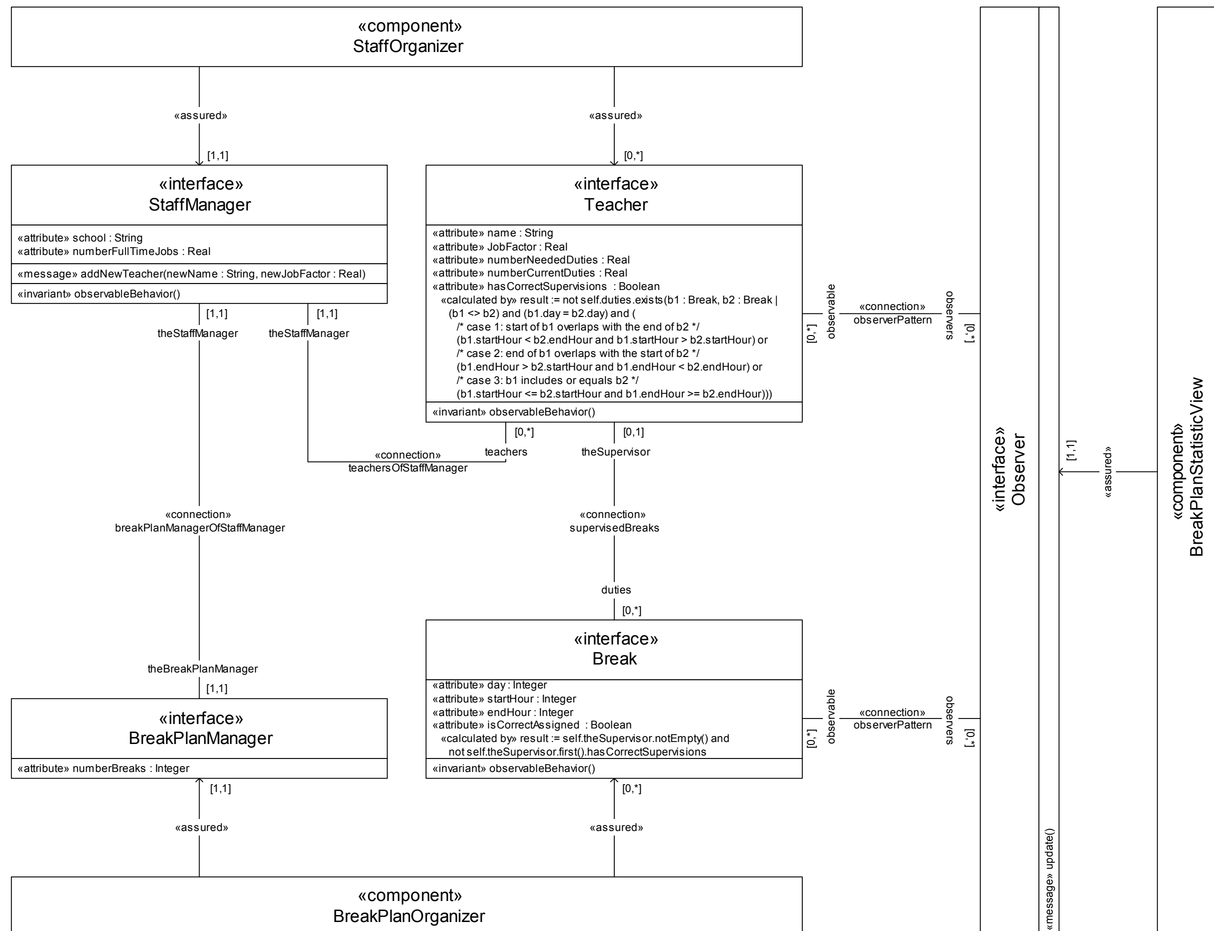

Abbildung 7.1: Komponentendiagramm des Pausenplaners mit Pausenstatistik I

Ein kleines, konstruiertes Beispiel illustriert die grundlegende Problematik. Abbildung 6.4 in Kapitel 6 beinhaltet mit den zwei Komponenten `StaffOrganizer` und `BreakPlanOrganizer` eine erste Spezifikation des Pausenplaners. Eine noch nicht realisiert Anforderungen des Systems ist, dass eine weitere Komponente `BreakPlanStatisticView` dem Benutzer aktuelle Statistikinformationen zur Verfügung stellen soll (vgl. Kapitel 4.2). In der nächsten Iteration soll diese Anforderung modelliert werden. Die Statistik soll dabei folgende Informationen beinhalten:

- Liste der Pausen, die entweder keinem Lehrer zugewiesen sind oder einem Lehrer, der mindestens eine weitere Pausenaufsicht zur gleichen Zeit hat.
- Liste der Lehrer, die zur gleichen Zeit mindestens zwei Pausenaufsichten haben.

Abbildung 7.1 zeigt eine entsprechende Evolution des Modells aus Abbildung 6.4. Die Komponente `BreakPlanStatisticView` mit der Schnittstelle `Observer` und den zugehörigen Verbindungen wurde integriert. Entsprechend dem Observer-Pattern wird die `update` Nachricht genau dann an die `Observer` Schnittstelle geschickt, wenn sich die Zustand der obser-

vierten `Teacher` und `Break` Schnittstellen ändert. Liegt die Nachricht `update` an, so kann die Komponente `BreakPlanStatisticView` die Pausenstatistik neu berechnen und so dem Benutzer eine aktuelle Statistik anzeigen.

Für die einfache Ermittlung der Lehrerliste, steht das neue, berechnete Attribut `hasCorrectSupervisions` in der `Teacher` Schnittstelle zur Verfügung. Die Berechungsvorschrift dieses Attributes ermittelt, ob der Lehrer mindestens zwei Pausenaufsichten zur gleichen Zeit hat. Ist der Wert dieses Attributes `false`, dann wird der Lehrer in die Liste der Lehrer aufgenommen, deren Pausenaufsichtspläne noch zu überarbeiten sind.

Für die einfache Ermittlung der Pausenliste, steht das neue, berechnete Attribut `isCorrectAssigned` der `Break` Schnittstelle zur Verfügung. Die Berechungsvorschrift dieses Attributes ermittelt, ob die Pause einem Lehrer zugewiesen ist, und ob der zugewiesene Lehrer mindestens eine weitere Pausenaufsicht zur gleichen Zeit hat. Pausen bei denen dieses Attribut `false` ist, werden in die Liste der Pausen aufgenommen, die noch zu überarbeiten sind.

Damit ist die evolutionäre weiter entwickelte Spezifikation des Systems abgeschlossen. Die neuen Anforderungen wurden in das Architekturmodell aufgenommen. Ein Prototyp kann generiert und dem Kunden vorgeführt werden.

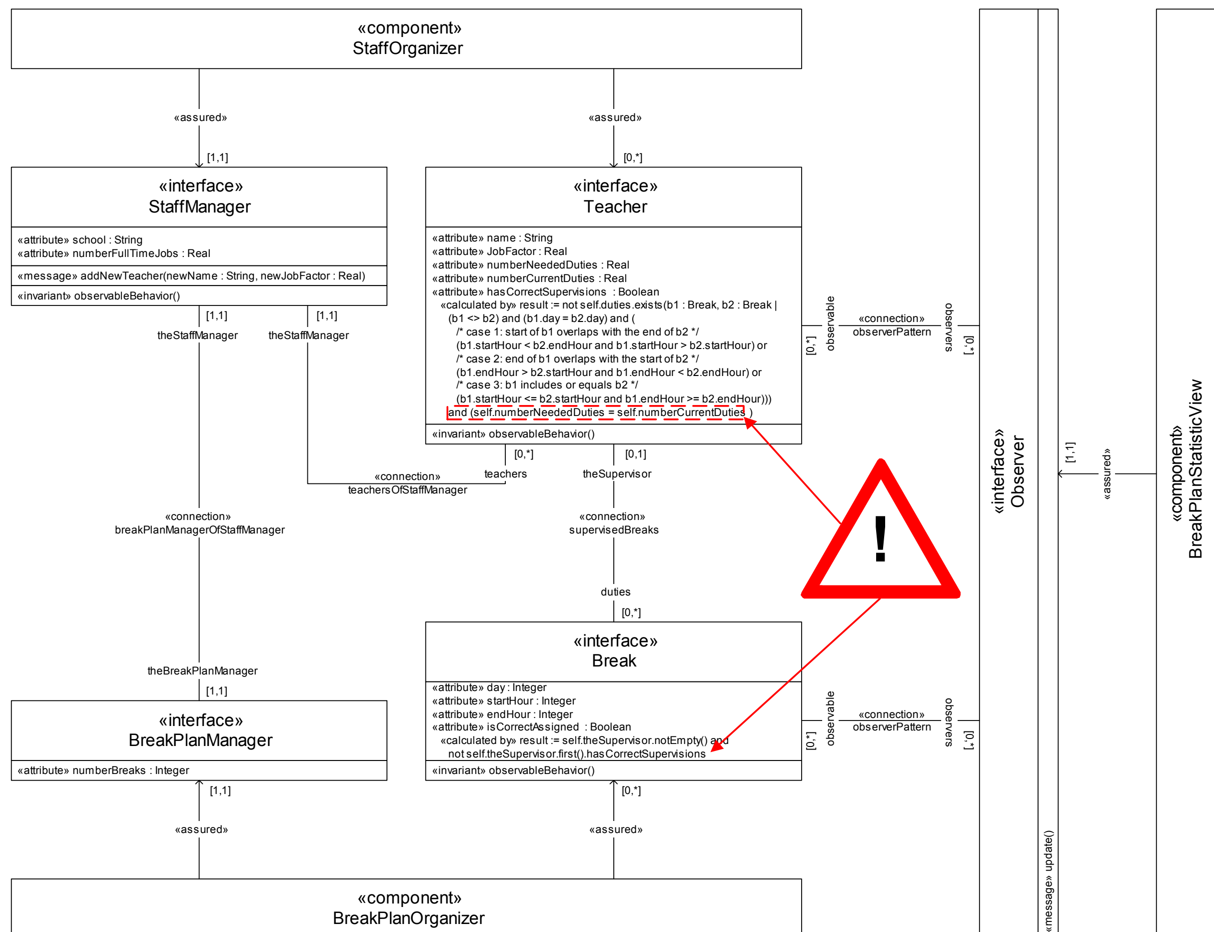

Abbildung 7.2: Komponentendiagramm des Pausenplaners mit Pausenstatistik II

In der nächsten Iteration soll die Pausenstatistik jetzt so verändert werden, dass die Pausenaufsichten gleichmäßig über alle Lehrer verteilt werden (vgl. Kapitel 4.2). Deshalb sollen in der Lehrerliste, deren Pausenaufsichtspläne noch zu überarbeiten sind, zusätzlich alle Lehrer

aufgenommen werden, deren aktuelle Anzahl der Pausenaufsichten nicht mit der benötigten Anzahl übereinstimmt.

Der Entwickler, der diese Änderungen durchführen soll, ändert hierfür die Berechung des Attributes `hasCorrectSupervisions` der `Teacher` Schnittstelle. Er fügt eine weitere Bedingung ein, die in dem gestrichelten Rechteck in Abbildung 7.2 dargestellt ist. Damit ist das berechnete Attribut `hasCorrectSupervisions` eines Lehrers nur noch dann `true`, wenn auch die Anzahl der zugewiesenen Pausenaufsichten mit den benötigten übereinstimmt. Dadurch wird die Lehrerliste in der Pausenstatistik jetzt entsprechend den neuen Anforderungen berechnet. Nach erfolgreich durchgeführten Testfällen wird der Entwickler diese Evolutionsstufe des Architekturmodells freigeben.

Es hat sich aber unbemerkt ein kleiner Fehler bei den Erweiterung eingeschlichen. Das Attribut `hasCorrectSupervisions`, dessen Berechnung der Entwickler modifiziert hat, wird auch für die Berechnung des Attributes `isCorrectAssigned` in der `Break` Schnittstelle verwendet. Somit hat sich auch die Berechnung dieses Attributes verändert (vgl. Abbildung 7.2), mit der Konsequenz, dass sich die Berechnung der Pausenliste in der Statistik verändert hat. Dies war weder beabsichtigt, noch entspricht es den Anforderungen. Das neue Modell ist somit nicht mehr korrekt. Hat man nicht entsprechende Testfälle im Vorfeld vorgesehen, so sind derartige Fehler meist sehr schwer zu finden. In der Regel führt dies zur Auslieferung eines fehlerhaften Systems. Früher oder später tritt dann beim Kunden der Fehler auf. Bis er identifiziert, reproduziert, diagnostiziert sowie behoben ist und eine verbesserte Version ausgeliefert wird, entstehen zusätzliche Kosten, die unter Umständen extrem hoch sind. Je früher ein Fehler entdeckt wird, desto geringer sind die Kosten um diesen Fehler zu beheben.

Die eigentliche Ursache für Fehler dieser Art ist, dass die komplexen Zusammenhänge zwischen den Komponenten in ihrer Vollständigkeit dem einzelnen Entwickler nicht bewusst sind und auf Grund der Größe des Systems auch überhaupt nicht bekannt sein können. Die Entwickler treffen deshalb implizit Annahmen über Teile des Systems. Treffen diese Annahmen nicht zu, führt dies zu Fehlern in der Modellierung, die wiederum in einem fehlerhaften Verhalten des gesamten Systems resultieren.

Das vorliegende Spielbeispiel hat dies bereits deutlich gemacht. Es gibt aber auch viele bekannte Beispiel aus der Industrie mit dramatischen Auswirkungen, wie zum Beispiel die Explosion der Ariane 5, der Verlust des Mars Climate Orbiters, die Probleme bei Sojourner's Mars-Auto Pathfinder oder die fehlerhaften Euro-Umrechnungen (siehe auch [Huck01]). Alle diese Beispiele sind darauf zurückzuführen, dass Komponenten des Systems Annahmen über andere Komponenten des Systems getroffen haben, die nicht zutrafen.

Dabei kann man nicht alleine den Softwareingenieuren die Schuld anlasten. Gängige Modellierungstechniken und Programmiersprachen stellen dem Ingenieur nur sehr rudimentäre Sprachmittel zur Verfügung, um Annahmen und Abhängigkeiten zwischen einzelnen Teilen eines Softwaresystems explizit und präzise formulieren zu können.

Die UML beispielsweise bietet nur die Relation „uses" um syntaktische Abhängigkeiten zwischen Modellelementen zu spezifizieren. In Java steht den Programmierern das Schlüsselwort „import" zur Verfügung, um Abhängigkeiten auf syntaktischer Ebene zu beschreiben. Abhängigkeiten auf semantischer Ebene lassen sich mit diesen Konzepten aber nicht abbilden. So kann man beispielsweise modellieren, dass eine Komponente A eine andere Komponente B benutzt. Was aber die Komponente A von der Komponente B erwartet, wenn die Komponente

A die Komponente B benutzt, kann mit den existierenden Modellierungskonzepten nicht spezifiziert werden.

7.2 Beispiel einer erweiterten Architekturspezifikation

Die Notwendigkeit immer komplexere Modelle evolutionär weiter zu entwickeln, stellt neue, zusätzliche Anforderungen an die verwendeten Modellierungssprachen. Die Abhängigkeiten zwischen einzelnen Teilen des Modells müssen für alle Beteiligten sichtbar, präzise und explizit modelliert werden können. Diese Transparenz erlaubt es dann, frühzeitig Aussagen über die Auswirkungen von lokalen Änderungen auf das gesamte Modell zu treffen. Für eine erfolgreiche evolutionäre Modellierung größerer Systeme ist diese Fähigkeit essentiell.

Einige Modellierungs- und Programmiersprachen beinhalten bereits erweiterte Konzepte für die Beschreibung der Abhängigkeiten zwischen den einzelnen Elementen eines Modells. Hierzu zählen die Konzepte von „Design by Contract“ [Meye92], die beispielsweise in Eiffel [Meye97] und in der Java Modeling Language [LBR99] zur Verfügung stehen.

Dabei werden in jeder Klasse bzw. Methode die Annahmen und Zusicherungen dieser Klasse bzw. Methode explizit spezifiziert. Diese Spezifikation repräsentiert einen Vertrag, den der Aufrufer zu erfüllen hat. Sind die Annahmen der Klasse bzw. Methode erfüllt, dann kann der Aufrufer sicher sein, dass die Klasse bzw. Methode die spezifizierten Zusicherungen einhält.

Diese Spezifikationstechnik ermöglicht es das Verhalten von Klassen bzw. Methoden unabhängig von anderen Klassen bzw. Methoden zu spezifizieren, in einer sogenannten „island specification“. Eine vollständige Beschreibung des Verhaltens ist aber mit diesen Spezifikationstechniken noch nicht möglich (siehe [BW97]). Ein weiterer wesentlicher Nachteil ist, dass bei einer Veränderung der Spezifikation keine Aussagen über die Beeinträchtigung bereits existierender Nutzer getroffen werden können. Hierfür müsste man noch zusätzlich die Abhängigkeiten und Nutzungsbeziehungen zwischen den Modellelemente beschreiben in einer sogenannten „behavioral dependencies specification“.

Einen anderer Ansatz wird mit den sogenannten „Interaction Contracts“ verfolgt (siehe [HHG90, Holl93]). Die Abhängigkeiten und Nutzungsbeziehungen zwischen den einzelnen Komponenten werden hier explizit modelliert, jedoch vermischt mit den eigentlichen Komponentenspezifikationen. Dies hat wiederum den Nachteil, dass unabhängige „island specifications“ für die einzelnen Komponenten nicht mehr existieren. Eine Komponente muss immer in ihrer tatsächlichen Umgebung modelliert werden. Wesentliche Eigenschaften von Componentware gehen dadurch verloren, wie zum Beispiel Wiederverwendung oder frühzeitiges, unabhängiges Testen einzelner Komponenten.

Damit die Abhängigkeiten zwischen den Komponenten eines Systems präzise und explizit modelliert werden können, ohne die Vorteile komponentenbasierter Spezifikationen zu verlieren benötigen wir für den evolutionären Architekturentwurf eine Spezifikationssprache, die

- einerseits unabhängige und vollständige Komponentenspezifikation erlaubt, sogenannte „island specification“, und
- andererseits eigenständige und präzise Abhängigkeitsspezifikationen ermöglicht, sogenannte „behavioral dependencies specification“.

In diesem Kapitel erweitern wir anhand des Evolutionsszenarios aus dem vorhergehenden Kapitel die Spezifikationstechnik aus Kapitel 6, damit sie diesen Anforderungen genügt.

Damit das Beispiel überschaubar bleibt, werden wir im folgenden nur noch die für das Beispiel relevanten Ausschnitte der Spezifikation betrachten. Abbildung 7.3 zeigt die entsprechend vereinfachte, textbasierte Spezifikation der Komponente `StaffOrganizer`. Diese beinhaltet nur noch die Schnittstelle `Teacher` mit dem berechneten Attribut `hasCorrectSupervisions` und einer Verbindung zu den Pausenaufsichten.

Neben der Sektion ASSURED kann eine Komponentenbeschreibung jetzt noch eine zusätzliche, neue Sektion beinhalten, die mit dem Schlüsselwort REQUIRED beginnt. ASSURED leitet den Anteil der Komponentenbeschreibung ein, der die Zusicherungen der Komponente beschreibt (vgl. Kapitel 6). REQUIRED hingegen beinhaltet die Bestandteile der Komponentenspezifikationen, mit der die Annahmen über die Umgebung der Komponente beschrieben werden. Um in diesem Beispiel eine einheitliche Nomenklatur zu verwenden, besitzen alle Spezifikationsbezeichner, die in der REQUIRED Sektion definiert werden, das Post-Fix „R“.

```
COMPONENT StaffOrganizer

  ASSURED

    INTERFACE Teacher [0,*]

      ATTRIBUTE hasCorrectSupervisions : Boolean CALCULATED BY
        result := not self.duties.exists(b1 : BreakR, b2 : BreakR |
          (b1 <> b2) and (b1.dayR = b2.dayR) and (
            /* case 1: start of b1 overlaps with the end of b2 */
            (b1.startHourR < b2.endHourR and b1.startHourR > b2.startHourR) or
            /* case 2: end of b1 overlaps with the start of b2 */
            (b1.endHourR > b2.startHourR and b1.endHourR < b2.endHourR) or
            /* case 3: b1 includes or equals b2 */
            (b1.startHourR <= b2.startHourR and b1.endHourR >= b2.endHourR)
          ))

      CONNECTION supervisedBreaks END duties : BreakR [0,*]

  REQUIRED

    INTERFACE BreakR [0,*]

      ATTRIBUTE dayR : Integer
      ATTRIBUTE startHourR : Integer
      ATTRIBUTE endHourR : Integer
```

Abbildung 7.3: Erweiterte textbasierte Spezifikation der Komponente `StaffOrganizer`

Wie in Abbildung 7.3 dargestellt, erwartet die Komponente `StaffOrganizer` von ihrer Umgebung eine Schnittstelle `BreakR` mit den Attributen `dayR`, `startHourR` und `endHourR`. Die Komponente `BreakPlanOrganizer` aus Abbildung 7.4 hingegen stellt eine Schnittstelle `Break` mit den Attributen `day`, `startHour`, `endHour` und `isCorrectAssigned` zur Verfügung. Im Gegenzug erwartet sie eine Schnittstelle `TeacherR` mit dem berechneten Attribut `hasCorrectSupervisionsR` und der Verbindung `dutiesR` zu den Pausenaufsichten. Insbesondere wird dabei auch die Berechnung des erwarteten Attributes `hasCorrectSupervisionsR` spezifiziert (siehe Abbildung 7.4).

Jede der Spezifikationen aus Abbildung 7.3 und Abbildung 7.4 beinhaltet eine unabhängige und in sich vollständige Komponentenbeschreibungen („island specification“). Die Zusicherungen der Komponente und die Annahmen über die Umgebung der Komponente werden dabei vollständig und präzise spezifiziert.

```
COMPONENT BreakPlanOrganizer

  ASSURED

    INTERFACE Break [0,*]

      ATTRIBUTE day : Integer
      ATTRIBUTE startHour : Integer
      ATTRIBUTE endHour : Integer
      ATTRIBUTE isCorrectAssigned : Boolean CALCULATED BY
        result := self.theSupervisor.notEmpty() and
          not self.theSupervisor.first().hasCorrectSupervisionsR

      CONNECTION supervisedBreaksR END theSupervisor : TeacherR [0,1]

  REQUIRED

    INTERFACE TeacherR [0,*]

      ATTRIBUTE hasCorrectSupervisionsR : Boolean CALCULATED BY
        result := not self.dutiesR.exists(b1 : Break, b2 : Break |
          (b1 <> b2) and (b1.day = b2.day) and (
            /* case 1: start of b1 overlaps with the end of b2 */
            (b1.startHour < b2.endHour and b1.startHour > b2.startHour) or
            /* case 2: end of b1 overlaps with the start of b2 */
            (b1.endHour > b2.startHour and b1.endHour < b2.endHour) or
            /* case 3: b1 includes or equals b2 */
            (b1.startHour <= b2.startHour and b1.endHour >= b2.endHour)
          ))

      CONNECTION supervisedBreaksR END dutiesR : Break [0,*]
```

Abbildung 7.4: Erweiterte textbasierte Spezifikation der Komponente `BreakPlanOrganizer`

Um aus den einzelnen Komponentenspezifikationen die Architektur eines komponentenbasierten Systems zu formen, haben wir in Kapitel 6 Systembeschreibungen eingeführt. Diese Systembeschreibungen werden jetzt so erweitert, dass zusätzlich die Abhängigkeiten zwischen den Komponenten im Kontext des Systems spezifiziert werden können („behavioral dependencies specification“). Damit setzen wir in den Systembeschreibungen das bereits vorgestellte Konzept der Annahme-/Zusicherungsverträge um (vgl. auch Abbildung 3.2 in Kapitel 3.4).

Abbildung 7.5 beinhaltet eine entsprechend erweiterte Beschreibung des Systems `BreakPlanner`. Zwischen den bereits bekannten Abschnitten `USED COMPONENTS` und `INITIALIZATION` sind zwei neue Abschnitte in der Systembeschreibungen enthalten, eingeleitet durch die Schlüsselworte `SPECIFIER MAPPING` und `REQUIREMENT ASSURANCE CONTRACT OF`.

Der erste Abschnitt beschreibt eine Abbildung zwischen Spezifikationsbezeichnern. Der Eintrag `BreakR → Break` bedeutet beispielsweise, dass die von der Komponente `StaffOrganizer` benötigte Schnittstelle `BreakR` durch die Schnittstelle `Break` der Komponente `BreakPlanOrganizer` zur Verfügung gestellt wird.

Dieser Spezifikationsbezeichnerabbildung folgt eine Menge von Annahme-/Zusicherungsverträgen. Für jede im System verwendete Komponente ist ein Annahme-/Zusicherungsvertrag vorhanden. Zum Beispiel wird der Annahme-/Zusicherungsvertrag der Komponente `BreakPlanOrganizer` in der Spezifikation durch die Zeile `REQUIREMENT ASSURANCE`

CONTRACT OF BreakPlanOrganizer eingeleitet. Dabei werden die Annahmen der Komponente entsprechenden Zusicherungen anderer Komponenten des Systems zugewiesen.

Jeder Annahme-/Zusicherungsvertrag besteht aus einer Menge von „Beweisen“. Beweise können aber müssen nicht mathematisch formal sein. Entscheidend ist, dass Beweisziel und verwendete Axiome explizit aufgeführt werden. Beweisziel ist jeweils eine angenommene Schnittstelle, die durch den entsprechenden Schnittstellenbezeichner referenziert wird. So wird beispielsweise mit PROOF OF GOAL TeacherR als Beweisziel die Schnittstelle TeacherR der Komponente BreakPlanOrganizer mit allen ihren Eigenschaften festgehalten.

Im konkreten Beispiel wird für den Beweis nur ein Axiom verwendet: USED AXIOM Teacher. Die verwendeten Axiome eines Beweises sind eine Menge von zugesicherten Schnittstellen mit allen ihren Eigenschaften, wiederum referenziert durch die Liste der zugehörigen Schnittstellenbezeichner.

```
SYSTEM BreakPlanner

  USED COMPONENTS StaffOrganizer, BreakPlanOrganizer

  SPECIFIER MAPPING

    BreakR → Break
    dayR → day
    startHourR → startHour
    endHourR → endHour
    TeacherR → Teacher
    hasCorrectSupervisionsR → hasCorrectSupervisions
    supervisedBreaksR → supervisedBreaks
    dutiesR → duties

  REQUIREMENT ASSURANCE CONTRACT OF StaffOrganizer

    PROOF OF GOAL BreakR

    USED AXIOM Break

    PROOF BODY

      /* no proof body required as axiom directly match goal */

  REQUIREMENT ASSURANCE CONTRACT OF BreakPlanOrganizer

    PROOF OF GOAL TeacherR

    USED AXIOM Teacher

    PROOF BODY

      /* no proof body required as axiom directly match goal */

  INITIALIZATION

    theStaffOrganizer : StaffOrganizer := NEW StaffOrganizer;
    theBreakPlanOrganizer : BreakPlanOrganizer := NEW BreakPlanOrganizer;
```

Abbildung 7.5: Erweiterte textbasierte Spezifikation des Systems BreakPlanner

Der Annahme-/Zusicherungsvertrag ist dabei genau dann gültig, wenn mit Hilfe der zuvor definierten Spezifikationsbezeichnerabbildung und einem optionalen Beweiskörper (PROOF

BODY) aus den Axiomen das Beweisziel ableitbar ist, wie das folgende Beispiel illustriert: Mit dem Axiom USED AXIOM Teacher in Abbildung 7.5 wird der in Abbildung 7.6 dargestellte Teil der Zusicherungen aus der Spezifikation der Komponente StaffOrganizer adressiert:

```
INTERFACE Teacher [0,*]

  ATTRIBUTE hasCorrectSupervisions : Boolean CALCULATED BY
    result := not self.duties.exists(b1 : BreakR, b2 : BreakR |
      (b1 <> b2) and (b1.dayR = b2.dayR) and (
        /* case 1: start of b1 overlaps with the end of b2 */
        (b1.startHourR < b2.endHourR and b1.startHourR > b2.startHourR) or
        /* case 2: end of b1 overlaps with the start of b2 */
        (b1.endHourR > b2.startHourR and b1.endHourR < b2.endHourR) or
        /* case 3: b1 includes or equals b2 */
        (b1.startHourR <= b2.startHourR and b1.endHourR >= b2.endHourR)
      ))

  CONNECTION supervisedBreaks END duties : BreakR [0,*]
```

Abbildung 7.6: Ausgangsaxiom des Beweises

Ersetzt man in diesem Spezifikationsartefakt die Spezifikationsbezeichner anhand der Abbildung, die in der Systemspezifikation beschrieben ist, so resultiert dies in dem Spezifikationsartefakt, das in der folgenden Abbildung 7.7 enthalten ist:

```
INTERFACE Teacher [0,*]

  ATTRIBUTE hasCorrectSupervisions : Boolean CALCULATED BY
    result := not self.duties.exists(b1 : Break, b2 : Break |
      (b1 <> b2) and (b1.day = b2.day) and (
        /* case 1: start of b1 overlaps with the end of b2 */
        (b1.startHour < b2.endHour and b1.startHour > b2.startHour) or
        /* case 2: end of b1 overlaps with the start of b2 */
        (b1.endHour > b2.startHour and b1.endHour < b2.endHour) or
        /* case 3: b1 includes or equals b2 */
        (b1.startHour <= b2.startHour and b1.endHour >= b2.endHour)
      ))

  CONNECTION supervisedBreaks END duties : Break [0,*]
```

Abbildung 7.7: Axiom des Beweises mit ersetzten Spezifikationsbezeichnern

Auf der anderen Seite wird mit PROOF OF GOAL TeacherR als Beweisziel das in Abbildung 7.8 dargestellte Spezifikationsartefakt referenziert:

```
INTERFACE TeacherR [0,*]

  ATTRIBUTE hasCorrectSupervisionsR : Boolean CALCULATED BY
    result := not self.dutiesR.exists(b1 : Break, b2 : Break |
      (b1 <> b2) and (b1.day = b2.day) and (
        /* case 1: start of b1 overlaps with the end of b2 */
        (b1.startHour < b2.endHour and b1.startHour > b2.startHour) or
        /* case 2: end of b1 overlaps with the start of b2 */
        (b1.endHour > b2.startHour and b1.endHour < b2.endHour) or
        /* case 3: b1 includes or equals b2 */
        (b1.startHour <= b2.startHour and b1.endHour >= b2.endHour)
      ))

  CONNECTION supervisedBreaksR END dutiesR : Break [0,*]
```

Abbildung 7.8: Beweisziel des Beweises

Führt man auf diesem Beweisziel ebenfalls die Ersetzung der Spezifikationsbezeichner durch, so erhält man das folgende, in Abbildung 7.9 dargestellte, Spezifikationsartefakt:

```
INTERFACE Teacher [0,*]

  ATTRIBUTE hasCorrectSupervisions : Boolean CALCULATED BY
    result := not self.dutiesR.exists(b1 : Break, b2 : Break |
      (b1 <> b2) and (b1.day = b2.day) and (
        /* case 1: start of b1 overlaps with the end of b2 */
        (b1.startHour < b2.endHour and b1.startHour > b2.startHour) or
        /* case 2: end of b1 overlaps with the start of b2 */
        (b1.endHour > b2.startHour and b1.endHour < b2.endHour) or
        /* case 3: b1 includes or equals b2 */
        (b1.startHour <= b2.startHour and b1.endHour >= b2.endHour)
      ))

  CONNECTION supervisedBreaks END duties : Break [0,*]
```

Abbildung 7.9: Beweisziel des Beweises mit ersetzten Spezifikationsbezeichnern

Mit ersetzten Spezifikationsbezeichnern sind Axiome und Ziel des Beweises identisch (vgl. Abbildung 7.7 und Abbildung 7.9). Ein zusätzlicher Beweiskörper ist in der Spezifikation nicht vorhanden und auch nicht mehr notwendig. Der Annahme-/Zusicherungsvertrag ist gültig. Der Spezifikationsbezeichner, der im Beweisziel angegeben ist, und alle im zugehörigen Spezifikationsartefakt enthaltenen Eigenschaften, wurden durch entsprechende zugesicherte Spezifikationsbezeichner und deren Eigenschaften abgedeckt.

Die Abhängigkeiten zwischen den Komponentenspezifikationen werden so durch diese Annahme-/Zusicherungsverträge explizit und nochvollziehbar beschrieben. Ändert sich eine Komponentenspezifikation im Zuge der evolutionären Modellierung so können resultierende Inkonsistenzen früher erkannt werden.

Wird beispielsweise die Spezifikation der Komponente `StaffOrganizer` wie im Kapitel 7.1 beschrieben weiter entwickelt, so ändern sich die Axiome des vorgestellten Beweises. Ein entsprechendes Werkzeug kann feststellen, dass die Annahmen der Komponente `BreakPlanOrganizer` nicht mehr zutreffen. Der Annahme-/Zusicherungsvertrag und der zugehörige Beweis ist nicht mehr gültig. Die Inkonsistenzen im System würden erkannt. Ein nachfolgende teure Fehlerbehebung wird verhindert.

Wesentlicher Nachteil ist ein erhöhter Spezifikationsaufwand. Der Übergang zwischen der einfachen und der erweiterten Spezifikationstechnik ist jedoch relativ schematisch und kann auch noch nachträglich erfolgen. Entfernt man aus den erweiterten Komponentenspezifikationen die `REQUIRED` Sektion sowie aus den Systemspezifikationen die `SPECIFIER MAPPING` und `REQUIREMENT ASSURANCE CONTRACT OF` Abschnitte und führt man die Spezifikationsbezeichnerersetzung auf der restlichen Spezifikation durch, so erhält man die einfachen Spezifikationen. Umgekehrt kann man auf diesem Wege aus den einfachen Spezifikationen Grundgerüste für erweiterte Spezifikationen erzeugen. Der Entwickler kann also zuerst mit den einfachen Spezifikationstechniken beginnen. Nachdem eine gewisse Stabilität erreicht ist, kann er dann zu der erweiterten Spezifikationstechnik übergehen. Dies entspricht auch dem in Kapitel 2.4 empfohlenen Vorgehen des evolutionären Architekturentwurfs.

7.3 Erweiterte Syntax von Architekturspezifikationen

Das Beispiel aus dem vorhergehenden Kapitel zeigt bereits relativ deutlich die erweiterte Syntax von Architekturspezifikationen komponentenbasierter Systeme. In diesem Kapitel werden wir die Produktionsregeln aus Kapitel 6.2 so erweitern, dass die Syntax eindeutig festgelegt ist und somit eine Basis für die anschließende Formalisierung zur Verfügung steht.

Für die Erweiterung der Komponentenspezifikationen muss nur eine einzige Produktionsregel angepasst werden. Die erweiterte Produktion des Symbols ⟨COMPONENT_SPECIFICATION⟩ enthält nun zusätzlichen einen optionalen Abschnitt mit einer Menge von Schnittstellenspezifikationen, der eingeleitet wird durch das Schlüsselwort REQUIRED :

⟨COMPONENT_SPECIFICATION⟩ ::=
COMPONENT ⟨COMPONENT_NAME⟩ ASSURED {⟨INTERFACE_SPECIFICATION⟩}*
{REQUIRED {⟨INTERFACE_SPECIFICATION⟩}*}?

Die notwendigen Erweiterungen bei Systemspezifikationen umfassen die Änderung einer Produktionsregel und die Einführung einiger neuer Produktionsregeln. Eine Systemspezifikation enthält jetzt zusätzlich noch eine optionale Abbildung zwischen Spezifikationsbezeichnern (⟨SPECIFIER_MAPPING_SPECIFICATION⟩) und eine optionale Menge von Annahme-/Zusicherungsverträgen (⟨REQUIREMENT_ASSURANCE_CONTRACT_SPECIFICATION⟩).

⟨SYSTEM_SPECIFICATION⟩ ::=
SYSTEM ⟨SYSTEM_NAME⟩ USED COMPONENTS {⟨COMPONENT_NAME⟩}*;
{⟨SPECIFIER_MAPPING_SPECIFICATION⟩ {⟨REQUIREMENT_ASSURANCE_CONTRACT_SPECIFICATION⟩}*}?
{INITIALIZATION ⟨BEHAVIOR_SPECIFICATION⟩}?
{FOR EACH INSTANCE OF ⟨COMPONENT_NAME⟩ EXISTS IMPLEMENTATION INSTANCE
⟨SYSTEM_NAME⟩ {WITH VISIBILITY ⟨MAPPING_SPECIFICATION⟩}*;}*

Die Abbildung zwischen Spezifikationsbezeichnern wird durch eine Liste von konkreten Einträgen in diese Abbildung beschrieben. Ein Eintrag in dieser Liste besteht aus zwei Spezifikationsbezeichnern, die jeweils Komponenten, Schnittstellen, Attribute, Verbindungen, Verbindungsendpunkte, Nachrichten oder Invarianten sein können.

⟨SPECIFIER_MAPPING_SPECIFICATION⟩ ::=
MAPPING SPECIFICATION {⟨SPECIFIER_MAPPING_ENTRY⟩}*

⟨SPECIFIER_MAPPING_ENTRY⟩ ::=
⟨SPECIFIER_NAME⟩ → ⟨SPECIFIER_NAME⟩

⟨SPECIFIER_NAME⟩ ::=
  ⟨COMPONENT_NAME⟩
  ‖ ⟨INTERFACE_NAME⟩
  ‖ ⟨ATTRIBUTE_NAME⟩
  ‖ ⟨CONNECTION_NAME⟩
  ‖ ⟨CONNECTION_END_NAME⟩
  ‖ ⟨MESSAGE_NAME⟩
  ‖ ⟨INVARIANT_NAME⟩

Und schließlich ein Annahme-/Zusicherungsvertrag besteht aus einem Komponentenbezeichner dem der Vertrag zugeordnet ist und einer Menge von Beweisen. Jeder Beweis enthält ein Beweisziel, das durch einen Schnittstellenbezeichner angegeben wird, eine Menge von Axiomen, in Form einer Menge von Schnittstellenbezeichnern, und einem optionalen Beweiskörper.

Die exakte Syntax eines Beweiskörpers, dargestellt durch das Symbol ⟨PROOF_DESCRIPTION⟩, ist für die Arbeit nicht von weiterem Interesse, da die Korrektheit eines Beweises nicht formal erfasst werden soll. Es sollen bewusst auch informelle Beweise im Rahmen der erweiterten Spezifikationstechnik für Softwarearchitekturen möglich sein, die beispielsweise aus reinem Prosa bestehen. Wichtig ist nur, dass die Abhängigkeiten zwischen den Spezifikationsbezeichner explizit spezifiziert werden. Die Überprüfung der Abhängigkeiten muss nicht zwingend vollständig automatisiert sein (vgl. Kapitel 8).

⟨REQUIREMENT_ASSURANCE_CONTRACT_SPECIFICATION⟩ ::=
  <u>REQUIREMENT ASSURANCE CONTRACT OF</u> ⟨COMPONENT_NAME⟩ {⟨PROOF_SPECIFICATION⟩}*

⟨PROOF_SPECIFICATION⟩ ::=
  <u>PROOF OF GOAL</u> ⟨INTERFACE_NAME⟩ <u>USED AXIOM</u> {⟨INTERFACE_NAME⟩}*,
  {<u>PROOF BODY</u> ⟨PROOF_DESCRIPTION⟩}?

7.4 Semantik von erweiterten Architekturspezifikationen

In dem vorhergehenden Kapitel haben wir die Syntax von erweiterten Architekturspezifikationen in Form von Produktionen einer Grammatik beschrieben. Ziel dieses Kapitels ist es die semantische Fundierung dieser erweiterten Spezifikationen analog zu Kapitel 6.3 vorzunehmen und dabei möglichst viel von Kapitel 6.3 direkt zu übernehmen.

Das erste Unterkapitel beschreibt deshalb die Grundlagen und Ziele dieser Formalisierung. Dabei wird insbesondere der Zusammenhang zwischen den erweiterten und den einfachen Architekturspezifikationen festgelegt. Dies erlaubt es uns die Formalisierung von Kapitel 6.3 direkt zu übernehmen und in den nächsten Unterkapiteln sogar noch auszubauen.

7.4.1 Grundlagen der Formalisierung erweiterter Architekturspezifikationen

Analog zu Kapitel 6.3.1 ist der Ausgangspunkt für die Formalisierung von erweiterten Architekturspezifikationen deren Grammatik und die zugehörige Sprache.

Definition 7.1: Grammatik und Sprache von erweiterten Architekturspezifikationen

Sei EASG *die Grammatik der erweiterten Architekturspezifikationen komponentenbasierter Systeme und wie folgt definiert:*

$$\text{EASG} =_{\text{def}} \left(\text{V}, \text{T}, \text{P}, \langle\text{ARCHITECTURE_SPECIFICATION}\rangle\right)$$

$\langle\text{ARCHITECTURE_SPECIFICATION}\rangle$ *ist das Startsymbol. Die Menge* P *beinhaltet die Produktionsregeln der Grammatik aus Kapitel 6.2 und die zugehörigen Erweiterungen aus Kapitel 7.3. Die Menge* T *besteht aus den terminalen Symbolen und die Menge* V *umfasst die nichtterminalen Symbole. Terminale Symbole sind in den Produktionen durch Unterstreichung gekennzeichnet und die nichtterminalen durch die Umklammerung mit „⟨ ⟩“. Die Sprache* $\text{L}(\text{EASG})$ *ist somit die Menge aller syntaktisch korrekten erweiterten Architekturspezifikationen:*

$$\text{L}(\text{EASG}) =_{\text{def}} \text{SPECIFICATION}$$

In der Menge $\text{L}(\text{EASG})$ *sind alle terminalen Wörter der Sprache – alle erweiterten Architekturspezifikationen – enthalten. Für jedes Element aus der Sprache* $\text{eas} \in \text{L}(\text{EASG})$ *existiert eine endliche Folge von Produktionsregeln, die angewendet werden müssen, um das Wort* eas *aus dem nichtterminalen Symbol* $\langle\text{ARCHITECTURE_SPECIFICATION}\rangle$ *abzuleiten.*

Wie bereits im Kapitel 7.2 skizziert, kann aus einer erweiterten Spezifikation eine entsprechende einfache Spezifikation durch Entfernen der zusätzlichen Abschnitte und Austauschen von Spezifikationsbezeichnern erzeugt werden. Dieser Ansatz erlaubt es uns die bestehende Formalisierung aus Kapitel 6.3 direkt zu übernehmen.

Definition 7.2: Zusammenhang zwischen erweiterten und einfachen Spezifikationen

Mit jeder syntaktisch korrekten erweiterten Architekturspezifikation $\text{eas} \in \text{L}(\text{EASG})$ *ist auch eine Abbildung zwischen Spezifikationsbezeichnern* $\text{specifier_mapping}_{\text{eas}}$ *gegeben:*

$$\text{specifier_mapping}_{\text{eas}} : \text{L}\left(\langle\text{SPECIFIER_NAME}\rangle_{\text{eas}}\right) \rightarrow \text{L}\left(\langle\text{SPECIFIER_NAME}\rangle_{\text{eas}}\right)$$

Dabei gilt:

$$\text{specifier_mapping}_{\text{eas}}(\text{s}_1) = \text{s}_2 \Leftrightarrow_{\text{def}} \text{s}_1 \ \underline{\rightarrow} \ \text{s}_2 \in \text{L}\left(\langle\text{SPECIFIER_MAPPING_ENTRY}\rangle_{\text{eas}}\right)$$

Aus einer syntaktisch korrekten erweiterten Architekturspezifikation $\text{eas} \in \text{L}(\text{EASG})$ *erzeugt die totale Abbildung* eas_2_as *eine syntaktisch korrekte einfache Architekturspezifikation* $\text{eas_2_as}(\text{eas})$*:*

$$\text{eas_2_ea} : \text{L}(\text{EASG}) \rightarrow \text{L}(\text{ASG})$$

Dabei werden in allen Komponentenspezifikationen die `REQUIRED` *Abschnitte entfernt sowie in allen Systemspezifikationen die* `SPECIFIER MAPPING` *und* `REQUIREMENT ASSURANCE CONTRACT OF` *Abschnitte. Außerdem werden in der resultierenden Spezifikation alle Spezifikationsbezeichner anhand der Abbildung* $\text{specifier_mapping}_{\text{eas}}$ *ersetzt.*

Mit Hilfe der Abbildung eas_2_as können wir aus einer erweiterten Architekturspezifikation $\text{eas} \in \text{L}(\text{EASG})$ eine einfache erzeugen. Ist die erweiterte Architekturspezifikation vollständig und konsistent (es gilt $\text{consistent}(\text{eas})$ und $\text{complete}(\text{eas})$ aus Definition 3.11), so kann aus der zugehörigen einfachen Spezifikation $\text{eas_2_as}(\text{eas})$ entsprechend der semantischen Fundierung aus Kapitel 6.3 ein Architekturprototyp generiert werden. Dieser kann dann in einer Laufzeitumgebung ausgeführt und getestet werden.

7.4.2 Erweiterte Komponentenspezifikationen

Gegenstand der vollständigen Formalisierung erweiterter Architekturspezifikationen sind damit nur die verbleibenden zusätzlichen Spezifikationselemente. Komponentenspezifikationen können einen neuen Spezifikationsabschnitt beinhalten, eingeleitet durch das Schlüsselwort REQUIRED. Dieser Abschnitt enthält alle Schnittstellen sowie deren Attribute, Verbindungen, Nachrichten und Invarianten, die von der Umgebung der Komponente zur Verfügung gestellt werden müssen.

Dementsprechend beschreibt dieser Abschnitt alle Eigenschaften, die von der Komponente angenommen werden. Wie in Kapitel 3.4 bereits diskutiert, werden diese Eigenschaften durch die Funktion $\mathsf{required}$ modelliert. Die Formalisierung sollte somit die Berechnung dieser Eigenschaften beschreiben. Auf die explizite Angabe der Eigenschaften können wir aber verzichten. Denn diese Eigenschaften entsprechen denen, die bereits im Kapitel 6.3 vorgestellt wurden. Einziger Unterschied ist, dass sie jetzt der Funktion $\mathsf{required}$ zugeordnet werden und nicht wie in Kapitel 6.3 der Funktion $\mathsf{assured}$.

Die folgende Definition illustriert dies exemplarisch. Dabei entspricht diese Definition der Definition 6.5 aus Kapitel 6.3. Demgemäß dürfen Schnittstelleninstanzen nur einer Komponenteninstanz zugeordnet werden, wenn sie auch in der Menge der angenommenen Schnittstellen in der Komponentenbeschreibung enthalten sind. Darüber hinaus muss die Anzahl der aktiven angenommenen Schnittstelleninstanzen, die einer Komponente zugewiesen sind, zu jedem Zeitpunkt innerhalb der spezifizierten minimalen und maximalen Instanzierungskardinalität liegen.

Definition 7.3: Struktur und Anzahl von Schnittstellen an Komponenten

Sei $\mathsf{eas} \in \mathsf{L}(\mathsf{EASG})$ *eine syntaktisch korrekte erweiterte Architekturspezifikation, so sind in der Menge der benötigten Eigenschaften* $\mathsf{required}(\mathsf{eas})$ *die folgenden Eigenschaften enthalten, soweit sie nicht durch entsprechende Annahme-/Zusicherungsverträgen bereits zugesichert sind:*

$$\begin{aligned}
&\forall t \in T, s \in \widehat{\mathsf{System}_{sys}}, c \in \mathsf{Component}_s, i \in \mathsf{Interface}_s \, . \\
&(c,i) \in \mathsf{assigned}_s^t \\
&\Rightarrow \\
&\exists w \in L\left(\langle \mathsf{COMPONENT_SPECIFICATION} \rangle_{eas}\right) . \\
&w = \underline{\texttt{COMPONENT}} \;\; \mathsf{specified}(c) \;\; \underline{\texttt{REQUIRED}} \;\; \{\langle \mathsf{INTERFACE_SPECIFICATION} \rangle\}^* \\
&\quad \underline{\texttt{INTERFACE}} \;\; \mathsf{specified}(i) \;\; \underline{[} \;\; \langle \mathsf{MINIMUM_CARDINALITY} \rangle \;\; \underline{,} \;\; \langle \mathsf{MAXIMUM_CARDINALITY} \rangle \;\; \underline{]} \\
&\quad \{\langle \mathsf{ATTRIBUTE_SPECIFICATION} \rangle\}^* \;\; \{\langle \mathsf{CONNECTION_SPECIFICATION} \rangle\}^* \\
&\quad \{\langle \mathsf{MESSAGE_SPECIFICATION} \rangle\}^* \;\; \{\langle \mathsf{INVARIANT_SPECIFICATION} \rangle\}^* \;\; \{\langle \mathsf{INTERFACE_SPECIFICATION} \rangle\}^*
\end{aligned}$$

$$\forall t \in T, s \in \widehat{System_{sys}}, c \in Component_s, in \in L\left(\langle INTERFACE_NAME \rangle_{eas}\right).$$
$$alive_s^t(s) \wedge alive_s^t(c)$$
$$\Rightarrow$$
$$\exists n, m \in N, w \in L\left(\langle COMPONENT_SPECIFICATION \rangle_{eas}\right).$$
$$n \leq \left|\left\{(c,i) \in assigned_s^t \middle| i \in Interface_s \wedge alive_s^t(i) \wedge specified(i) = in\right\}\right| \leq m \wedge$$
$$w = \underline{\texttt{COMPONENT}}\ \ specified(c)\ \ \underline{\texttt{REQUIRED}}\ \ \{\langle INTERFACE_SPECIFICATION \rangle\}^*$$
$$\underline{\texttt{INTERFACE}}\ \ in\ \ \underline{[}\ \ n\ \ \underline{,}\ \ m\ \ \underline{]}\ \ \{\langle ATTRIBUTE_SPECIFICATION \rangle\}^*\ \ \{\langle CONNECTION_SPECIFICATION \rangle\}^*$$
$$\{\langle MESSAGE_SPECIFICATION \rangle\}^*\ \ \{\langle INVARIANT_SPECIFICATION \rangle\}^*\ \ \{\langle INTERFACE_SPECIFICATION \rangle\}^*$$

Auf eine vollständige Formalisierung aller angenommener Eigenschaften, die aus Komponentenspezifikationen berechnet werden können, wird verzichtet. Das vorliegende Beispiel zeigt bereits die notwendigen syntaktische Umformung, die an allen Definitionen aus Kapitel 6.3 vorzunehmen sind, für die vollständige Formalisierung der Funktion $required$.

7.4.3 Erweiterte Systemspezifikationen

Erweiterte Architekturspezifikationen können die zwei neuen Abschnitte SPECIFIER MAPPING und REQUIREMENT ASSURANCE CONTRACT OF in einer Systemspezifikation enthalten. Die Formalisierung des ersten neuen Abschnittes wurde bereits im Kapitel 7.4.1 beschrieben. Der zweite Abschnitt, REQUIREMENT ASSURANCE CONTRACT OF beschreibt die Annahme-/Zusicherungsverträge der Komponenten eines Systems.

Definition 7.4: Annahme-/Zusicherungsverträge von Komponenten

Sei $eas \in L(EASG)$ *eine syntaktisch korrekte erweiterte Architekturspezifikation, so sind die Annahme-/Zusicherungsverträge von den Komponenten der Spezifikation wie folgt definiert:*

$$\forall c \in L\left(\langle COMPONENT_NAME \rangle_{eas}\right), w \in L\left(\langle REQUIREMENT_ASSURANCE_CONTRACT_SPECIFICATION \rangle_{eas}\right),$$
$$n \in N, g, a_1, \ldots, a_n \in L\left(\langle INTERFACE_NAME \rangle_{eas}\right).$$
$$w = \underline{\texttt{REQUIREMENT ASSURANCE CONTRACT OF}}\ \ c\ \ \{\langle PROOF_SPECIFICATION \rangle\}^*$$
$$\underline{\texttt{PROOF OF GOAL}}\ \ g\ \ \underline{\texttt{USED AXIOM}}\ \ a_1, \ldots, a_n\ \ \{\underline{\texttt{PROOF BODY}}\ \ \langle PROOF_DESCRIPTION \rangle\}^?$$
$$\{\langle PROOF_SPECIFICATION \rangle\}^*$$
$$\Rightarrow$$
$$contract_of_{eas}(c) = \left\{(r,x,a) \middle| r \in required(g) \wedge x \in \{a_1, \ldots, a_n\} \wedge a \in assured(x)\right\}$$

Die einzelnen Einträge in einem Annahme-/Zusicherungsvertrag einer Komponente bestehen dabei aus den angenommenen Eigenschaften der Beweisziele und den zugesicherten Eigenschaften der Axiome, wie in Definition 7.4 festgelegt ist. Die Zuordnung dieser Eigenschaften ist gültig, wenn die einzelnen Beweise korrekt sind. Ein Beweis ist dabei genau dann korrekt, wenn aus den Spezifikationsartefakten, die durch die Beweisaxiome referenziert werden, über den Beweiskörper die Spezifikationsartefakte abgeleitet werden können, die durch das Beweisziel vorgegeben sind (vgl. Definition 7.5). Die Funktion $exchange_specifier_{eas}$ nimmt dabei ein Spezifikationsartefakt und tauscht die darin enthaltenen Spezifikationsbezeichner entsprechend der Spezifikationsbezeichnerabbildung $specifier_mapping_{eas}$ aus.

Definition 7.5: Gültigkeit von Annahme-/Zusicherungsverträgen

Sei $eas \in L(EASG)$ *eine syntaktisch korrekte erweiterte Architekturspezifikation, so sind die Annahme-/Zusicherungsverträge genau dann gültig, wenn die entsprechenden Beweise korrekt sind:*

$$\forall c \in L(\langle COMPONENT_NAME \rangle_{eas}), w \in L(\langle REQUIREMENT_ASSURANCE_CONTRACT_SPECIFICATION \rangle_{eas}),$$
$$n \in \mathbb{N}, g, a_1, \ldots, a_n \in L(\langle INTERFACE_NAME \rangle_{eas}), w_g, w_{a_1}, \ldots, w_{a_n} \in L(\langle INTERFACE_SPECIFICATION \rangle_{eas}),$$
$$w_b \in L(\langle PROOF_DESCRIPTION \rangle_{eas}), (r, x, a) \in contract_of_{eas}(c).$$

$$w = \underline{\texttt{REQUIREMENT ASSURANCE CONTRACT OF}} \ c \ \{\langle PROOF_SPECIFICATION \rangle\}^*$$
$$\underline{\texttt{PROOF OF GOAL}} \ g \ \underline{\texttt{USED AXIOM}} \ a_1, \ldots, a_n \ \{\underline{\texttt{PROOF BODY}} \ w_b\}^? \ \{\langle PROOF_SPECIFICATION \rangle\}^*$$
$$\wedge$$
$$w_g = \underline{\texttt{INTERFACE}} \ g \ \underline{[} \ \langle MINIMUM_CARDINALITY \rangle \ \underline{,} \ \langle MAXIMUM_CARDINALITY \rangle \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION \rangle\}^* \ \{\langle CONNECTION_SPECIFICATION \rangle\}^* \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION \rangle\}^*$$
$$\wedge$$
$$w_{a_1} = \underline{\texttt{INTERFACE}} \ a_1 \ \underline{[} \ \langle MINIMUM_CARDINALITY \rangle \ \underline{,} \ \langle MAXIMUM_CARDINALITY \rangle \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION \rangle\}^* \ \{\langle CONNECTION_SPECIFICATION \rangle\}^* \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION \rangle\}^*$$
$$\wedge \ldots \wedge$$
$$w_{a_n} = \underline{\texttt{INTERFACE}} \ a_n \ \underline{[} \ \langle MINIMUM_CARDINALITY \rangle \ \underline{,} \ \langle MAXIMUM_CARDINALITY \rangle \ \underline{]}$$
$$\{\langle ATTRIBUTE_SPECIFICATION \rangle\}^* \ \{\langle CONNECTION_SPECIFICATION \rangle\}^* \ \{\langle MESSAGE_SPECIFICATION \rangle\}^*$$
$$\{\langle INVARIANT_SPECIFICATION \rangle\}^*$$
$$\Rightarrow$$
$$holds(a,r) \Leftrightarrow (exchange_specifier_{eas}(w_{a_1}) \ldots exchange_specifier_{eas}(w_{a_n}) \Rightarrow w_b \Rightarrow exchange_specifier_{eas}(w_g))$$

7.5 Erweiterte UML-basierte grafische Spezifikationstechnik

Die UML hat sich als der internationale Standard für objektorientierte Modellierung in Forschung und Industrie etabliert [BJR98, RJB98, OMG00a]. Sie beinhaltet bereits eine Reihe Erweiterungsmechanismen (vgl. auch [DSB99] und [KRSW01]). Aus Gründen der praktischen Relevanz haben wir in Kapitel 6.4 eine UML-basierte grafische Variante der einfachen Beschreibungstechnik für Softwarearchitekturen vorgestellt. In diesem Kapitel stellen wir eine entsprechende Weiterentwicklung dieser grafischen Beschreibungstechnik vor, die zusätzlich die Konzepte erweiterter textbasierter Architekturspezifikationen beinhaltet.

Abbildung 7.10 zeigt das Komponentendiagramm der Komponente `StaffOrganizer`. Dieses Diagramm entspricht direkt der textbasierten Spezifikation aus Abbildung 7.3. Im Vergleich zu dem ursprünglichen Komponentendiagramm in Abbildung 6.4 werden in der neuen Diagrammart den Komponenten nicht nur eine Menge zugesicherter Schnittstellen sondern auch eine Menge angenommener Schnittstelle zugeordnet. Angenommene Schnittstellen werden durch Assoziationen mit dem Stereotyp «`required`» dargestellt und zugesicherte durch Assoziationen mit dem Stereotyp «`assured`». Für die Schnittstellenbeschreibungen selbst, gleichgültig ob zugesicherte oder angenommene Schnittstellen, werden die bereits vorgestellten Modellierungselemente mit den Stereotypen «`attribute`», «`message`», «`invariant`» und «`connection`» verwendet.

In Abbildung 7.11 ist das UML-basierte Systemdiagramm des Systems `BreakPlanner` dargestellt. Dieses Diagramm entspricht direkt der textbasierten Spezifikation aus Abbildung 7.5. Betrachtet man das Systemdiagramm in Abbildung 6.6, so sind in dem neuen Diagramm nur Erweiterungen im Abschnitt mit dem Stereotyp «`used components`» vorhanden. Dieser Abschnitt enthält jetzt nicht nur eine Liste verwendeter Komponenten, sondern auch deren zugesicherte und angenommene Schnittstellen mit ihren Attributen, Nachrichten, Invarianten und Verbindungen. Die Verhaltensbeschreibungen bei Attributen, Nachrichten und Invarianten werden dabei ausgeblendet. Wichtig ist, dass die vollständige Struktur jeder Komponente des Systems dargestellt ist. Mit Hilfe von Assoziationen, dargestellt durch gestrichelte Pfeile mit dem Stereotypen «`specifier mapping`», kann dann die Abbildung zwischen den angenommenen und den zugesicherten Spezifikationsbezeichnern grafisch spezifiziert werden.

Darüber hinaus hat jede Komponente innerhalb des Abschnittes «`used components`» einen zusätzlichen Abschnitt, eingeleitet durch den Stereotyp «`requirement assurance contract`». In diesem Abschnitt ist der Annahme-/Zusicherungsvertrage der Komponente mit dem zugehörigen Beweis enthalten. Jeder Beweis besteht aus den drei Teilen, die jeweils durch die Stereotypen «`proof of goal`», «`used axiom`» und «`proof body`» eingeleitet werden und das Beweisziel, die verwendeten Axiome und den Beweiskörper beinhalten.

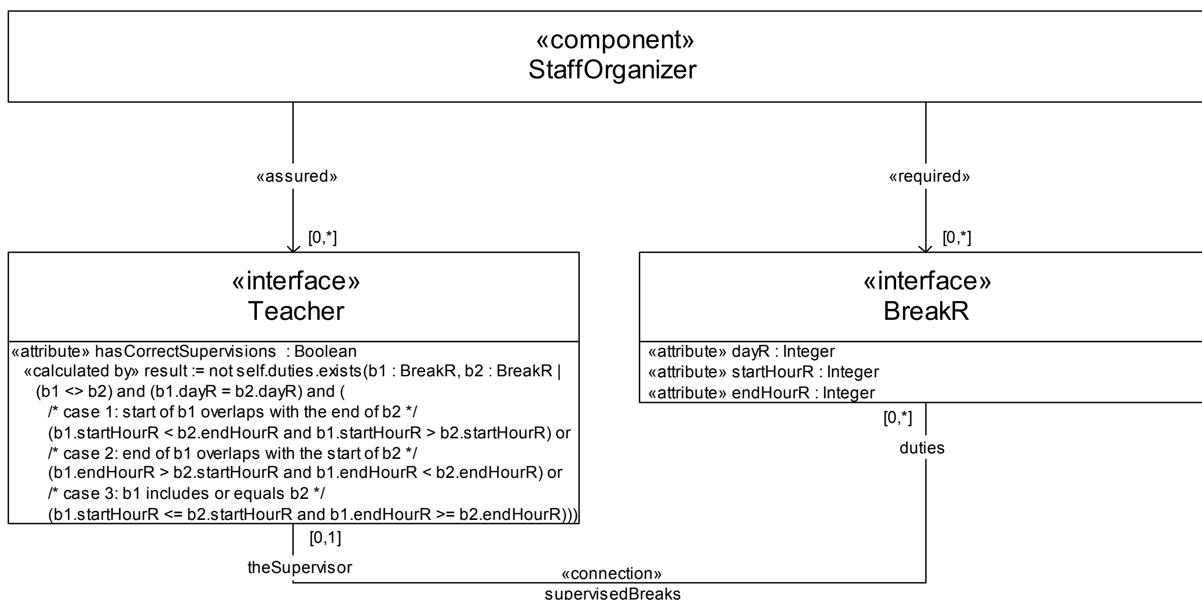

Abbildung 7.10: Erweitertes UML-basiertes Komponentendiagramm

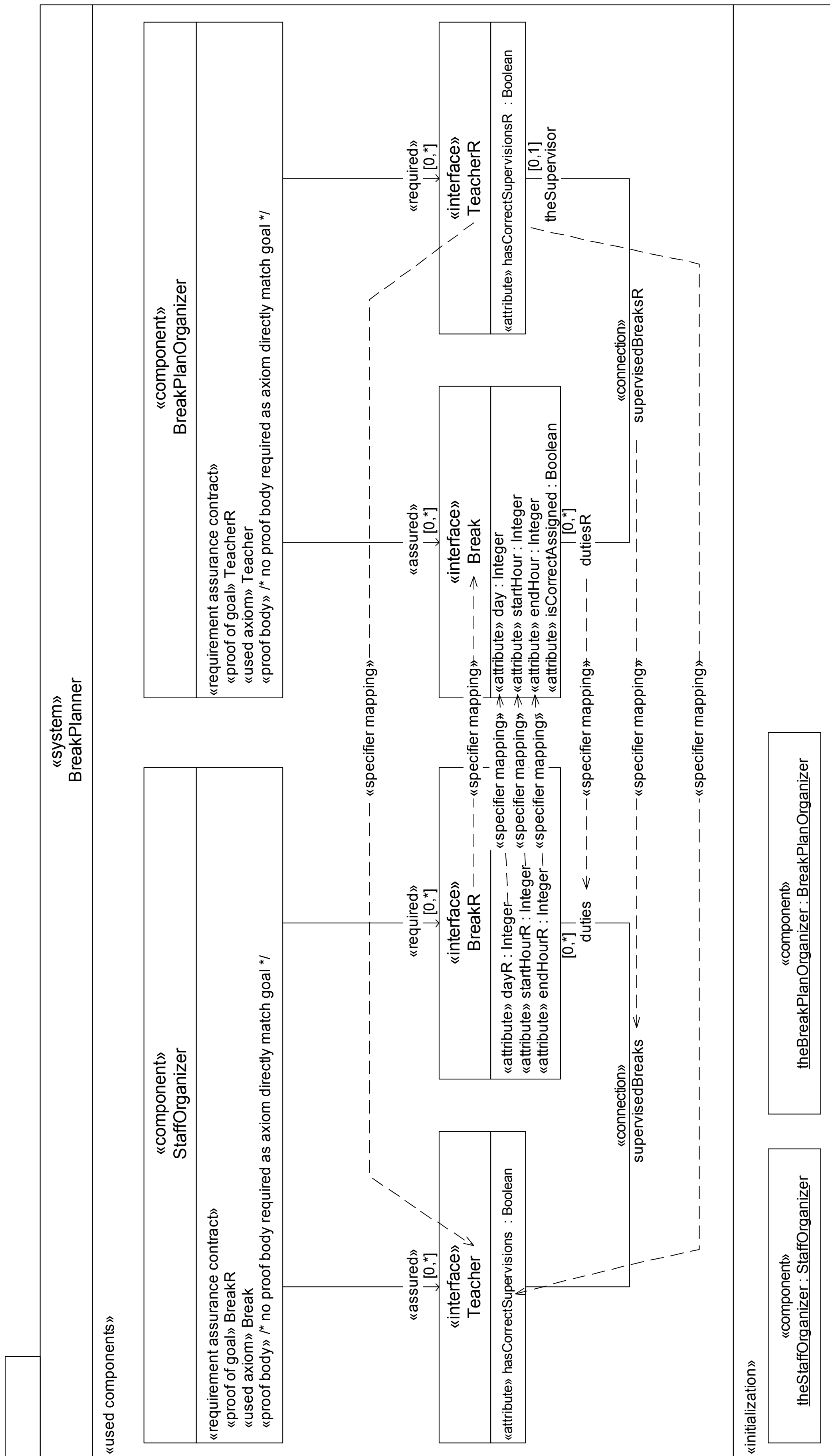

Abbildung 7.11: Erweitertes UML-basiertes Systemdiagramm

7.6 Zusammenfassung

Softwaresysteme und deren Architekturen sind sehr komplex. In der Fülle ihrer Details können sie von einzelnen Menschen meist nicht mehr vollständig durchdrungen werden. Dies ist keine neue Erkenntnis. Viele Softwareingenieure mussten dies bereits in Projekten erfahren: Steht man beispielsweise vor der Aufgabe, an einem bestehenden System ein kleines, neues Feature zu integrieren, so greift nicht selten folgendes Szenario: Nachdem man die neue Funktionalität, spezifiziert, entworfen, realisiert und getestet hat, wird sie in das System integriert. Nach der Integration arbeitet das System bezüglich der neuen Funktionalität korrekt, jedoch treten Probleme bei der Ausführung alter Funktionen auf.

Korrekte lokale Änderungen, die im Zuge der evolutionären Modellierung durchgeführt werden, können zu einem fehlerhaften Verhalten des Gesamtsystems führen, was unter Umständen zunächst nicht bemerkt wird. Der Grund hierfür ist, dass Entwickler die komplexen Zusammenhänge zwischen den einzelnen Komponenten des Systems in ihrer Vollständigkeit nicht mehr verstehen können. Existierende Modellierungstechniken bieten ihnen nur sehr rudimentäre Sprachmittel um diese Zusammenhänge zu beschreiben.

Für eine optimale Unterstützung des evolutionären Entwurfs komponentenbasierter Systeme haben wir deshalb in diesem Kapitel die Spezifikationstechniken aus Kapitel 6 entsprechend verbessert. Mit der erweiterten Modellierungstechnik können die Abhängigkeiten zwischen den Komponenten eines Systems explizit und präzise beschrieben werden.

Dazu wurden die Komponentenspezifikationen so erweitert, dass jede Komponente unabhängig von anderen Spezifikationsartefakten vollständig und präzise in einer sogenannten „island specification“ beschrieben werden kann. Darüber hinaus sind eigenständige und präzise Spezifikationen der Abhängigkeiten zwischen den Komponenten innerhalb der Systemspezifikationen möglich. Diese „behavioral dependencies specification“ basieren auf dem Konzept der Annahme-/Zusicherungsverträge. Ändern sich bestimmte Komponentenspezifikationen im Rahmen des evolutionären Entwurfs, so können diese Annahme-/Zusicherungsverträge erneut überprüft werden. Damit wird der Integrationstest bereits auf der Modellebene durchgeführt. Fehler in der Spezifikation können so früher erkannt und behoben werden.

Für diese erweiterte Spezifikationstechnik haben wir entsprechende Erweiterungen der ursprünglichen Grammatik aus Kapitel 6 definiert. Infolgedessen wurde die zugehörige semantische Fundierung auf Basis der prädikatenbasierten formalen Semantik angepasst. Damit steht ein präzises formales Verständnis der erweiterten Spezifikationstechniken zur Verfügung. Abschließend haben wir anhand eines Beispieles gezeigt, wie diese formal fundierte, textbasierte Spezifikationstechnik in eine UML-basierte grafische Variante abgebildet werden kann.

8 Werkzeugunterstützung

Der evolutionäre Architekturentwurf komponentenbasierter Systeme besteht, wie jede Methodik, aus einer Menge von einzelnen Entwicklungsschritten. Erst durch das methodische Vorgehen werden diese Entwicklungsschritte zu einer zielgerichteten Vorgehensweise kombiniert. Der Erfolg einer Methodik in der industriellen Praxis wird maßgeblich von der Qualität und Durchgängigkeit der Werkzeugunterstützung bestimmt. Denn eine möglichst weitreichende Werkzeugunterstützung kann die Effektivität einer Methode entscheidend beeinflussen.

Deshalb ist heute die Notwendigkeit des Einsatzes maschineller Unterstützung zur Durchführung von Softwareentwicklung mehr denn je unbestritten. Entsprechende Werkzeuge sind in nahezu allen Bereichen der Softwareentwicklung sinnvoll und kommerziell verfügbar. Diese Werkzeuge werden in der industriellen Praxis erfolgreich eingesetzt, beispielsweise für die Modellierung, zur Erstellung von Dokumentation, für das Konfigurationsmanagement, für das Änderungsmanagement, für die Programmgenerierung, für das Tailoring von Prozessmodellen, zur Unterstützung des Prozessmodells selbst oder für die Projektplanung.

Für die in dieser Arbeit vorgestellten Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme existiert jedoch noch keine entsprechende Werkzeugunterstützung. Die verfügbaren Werkzeugen liefern allerdings bereits einen Baukasten, auf dessen Basis eine durchgängige und weitreichende Werkzeugunterstützung konzipiert und entwickelt werden kann.

Im Kapitel 8.1 werden wir grundlegende Konzepte einer umfassenden und durchgängigen Werkzeugunterstützung im Kontext des Vorgehensmodells des evolutionären Architekturentwurfs erarbeiten. Dabei können wir fünf elementare Bausteine einer entsprechenden Werkzeugunterstützung identifizieren: Modellierungs- und Spezifikationswerkzeuge, Konsistenz- und Integrationsüberprüfung, Generierung von Programmcode, Ausführungs- und Testumgebung sowie Versions- und Migrationsunterstützung.

In den folgenden fünf Kapiteln, Kapitel 8.2 bis 8.6, diskutieren wir diese Bausteine detaillierter. Die funktionalen Anforderungen an diese Werkzeuge werden erarbeitet. Wir präsentieren die grundlegenden Konzepte und Technologien der Werkzeuge. Außerdem stellen wir bereits existierende Komponenten und Teillösungen vor und demonstrieren, wie diese in den von uns entwickelten Werkzeugprototyp DesignIt integriert werden können [Desi01].

8.1 Konzept einer umfassenden Werkzeugunterstützung

Eingangs, in Kapitel 2.4 haben wir die einzelnen Entwicklungsschritte des evolutionären Architekturentwurfs komponentenbasierter Systeme vorgestellt: Spezifikation der Softwarearchitektur (*Software Architecture Specification*), Validierung der evolutionären Modelländerungen (*Validate Evolutionary Model Changes*), Generierung und Implementierung eines Architekturprototypen (*Generate / Implement Architectural Prototype*) und schließlich Testen der Softwarearchitektur (*Test Software Architecture*).

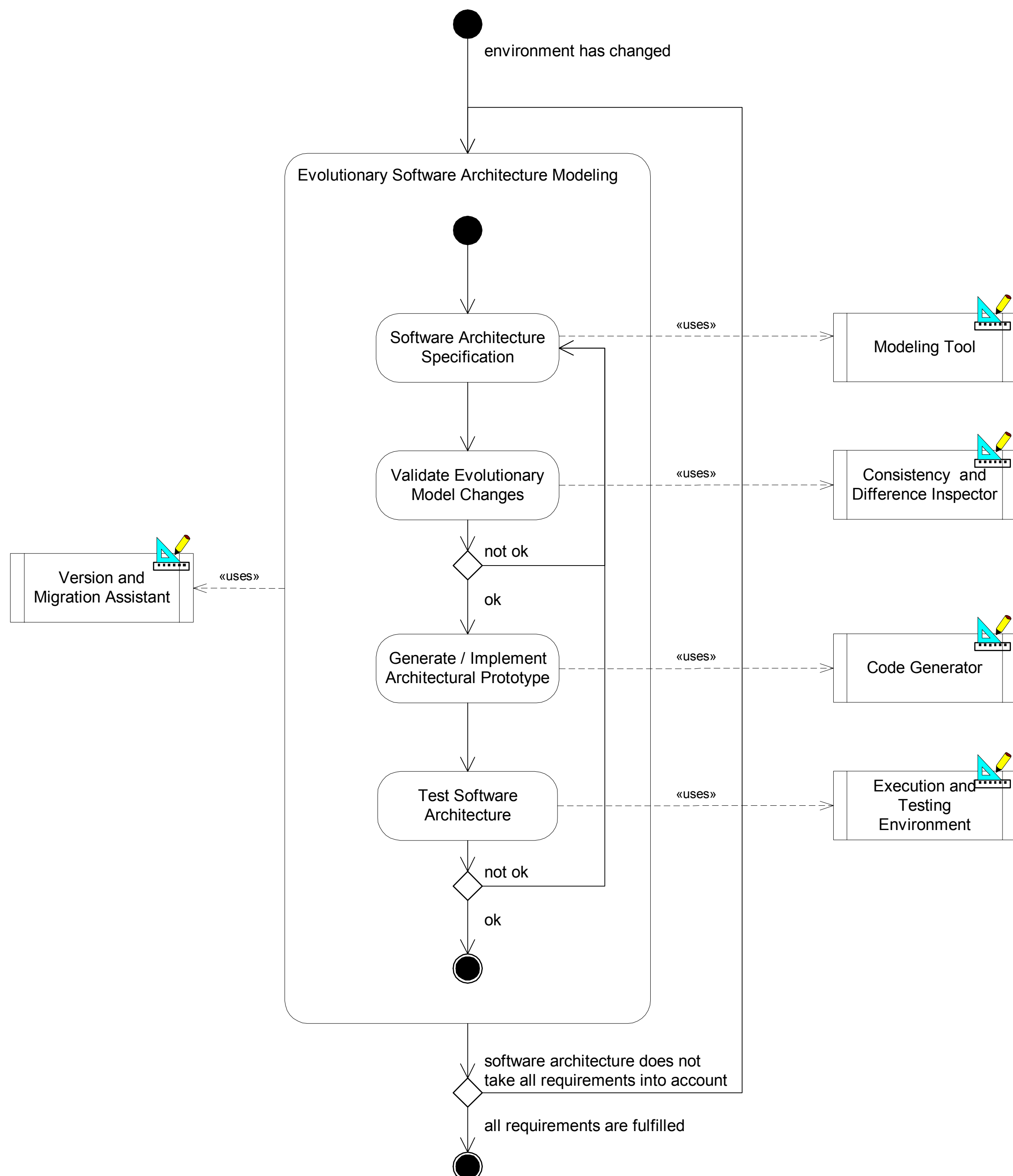

Abbildung 8.1: Werkzeugunterstützung für den evolutionären Architekturentwurf

Das UML Aktivitätsdiagramm in Abbildung 8.1 zeigt diese einzelnen Entwicklungsschritte nochmals im Zusammenhang. Jedem Entwicklungsschritt können bestimmte Werkzeuge zuordnet werden, die der Softwareingenieur für eine effiziente Umsetzung einsetzen wird:

- Modellierungs- und Spezifikationswerkzeuge (*Modeling Tool*) werden für die Erstellung, Bearbeitung und Verwaltung von grafischen wie auch textbasierten Architekturspezifikationen verwendet.
- Die Konsistenz- und Integrationsüberprüfung (*Consistency and Difference Inspector*) von den so erstellten Architekturspezifikationen besteht aus automatisierten syntaktischen Prüfmechanismen sowie semantischen Konsistenzüberprüfungen, für die unter Umständen zusätzliche Interaktion mit dem Entwickler notwendig sind.
- Für die Generierung von Programmcode (*Code Generator*) aus den erstellten und validierten Architekturspezifikationen sind spezielle Verfahren zur Programmgenerierung notwendig. Programme sollten für unterschiedliche Ausführungsplattformen in verschiedenen Programmiersprachen erzeugt werden können.
- Eine Ausführungs- und Testumgebung (*Execution and Testing Environment*) ist notwendig, um die generierten Programme kontrolliert ausführen zu können. Sowohl Ad-hoc-Testfälle als auch Regressionstests sollten möglichst weitgehend unterstützt und automatisiert durchgeführt werden können.
- Schließlich, ist eine Versions- und Migrationsunterstützung (*Migration Assistant*) erforderlich, um unterschiedliche Modellversionen zu verwalten. Die mit der evolutionäre Modellierung einhergehenden Änderungen an Modellen und deren Auswirkungen sind durch eine entsprechende Werkzeugunterstützung abzufedern.

Für eine optimale Unterstützung des evolutionären Architekturentwurfs sind aber nicht allein diese Werkzeuge ausschlaggebend. Die Durchgängigkeit und Integration der Werkzeugunterstützung ist ein zentraler Faktor. Wir unterscheiden hierbei vier verschiedene Integrationsstufen, die jeweils aufeinander aufbauen (vgl. auch [Keie99b]):

- Systemtechnische Integration
- Modelltechnische Integration
- Verfahrenstechnische Integration
- Prozesstechnische Integration

Die systemtechnische Integration von Werkzeugen ist vergleichbar mit der Verwendung einer gemeinsamen Sprache in der herkömmlichen zwischenmenschlichen Kommunikation. Durch die Verwendung einer gemeinsamen Sprache oder einer geeigneten Übersetzung können alle Beteiligten sich untereinander verständigen. Diese Integrationsstufe dient der Umsetzung einer gemeinsamen Kommunikationsinfrastruktur. Sie schafft ein gemeinsames, von allen Werkzeugen nutzbares Kommunikations- und Organisationsmedium.

Die modelltechnische Integration ist dafür verantwortlich, dass alle Benutzer eines Informationsmodells nicht nur dieselbe Sprache sprechen, sondern dass sie auch das gleiche Verständnis der ausgetauschten Informationen haben. Die modelltechnische Integration entspricht der Realisierung eines Produktmodells, das alle Werkzeuge als gemeinsame Datenbasis verwenden. Die unterschiedlichen, unter Umständen interdisziplinären, Sichten der Werkzeuge und der Projektteilnehmer auf die Informationen werden durch fachspezifischen Sichten auf ein gemeinsames Modell realisiert.

Die verfahrenstechnische Integration setzt sich zum Ziel durch Parallelisierung und Nebenläufigkeit von Arbeitsschritten innerhalb des Entwurfs- und Entwicklungsprozesses (Simultane-

ous and Concurrent Engineering), unter Einhaltung der geforderten Qualitätsansprüche, die Entwicklungszeiten und -kosten zu minimieren. Für die verfahrenstechnische Integration sind den Werkzeugen eine Reihe allgemeiner technischer Koordinationsfunktionen auf dem gemeinsamen Produktmodell zur Verfügung zu stellen. Beispiele solcher Koordinationsfunktionen sind Versionierung, Sicherheits- und Rechtekonzepte, Transaktionsverwaltung, und Sperrmechanismen.

Unter der prozesstechnischen Integration wird die Unterstützung der effizienten Zusammenarbeit eines Teams verstanden. Dies erfordert die Verteilung und Koordination von Teilaufgaben, die zu einem gemeinsamen Ergebnis zusammengestellt werden. Hierfür bedarf es einer Prozessmanagementkomponente, welche die Prozesse und die einzelnen, eventuell mit Werkzeugunterstützung, durchzuführenden Teilaufgaben plant, steuert und überwacht. Mit der prozesstechnischen Integration wird eine durchgängige und weitreichende Werkzeugunterstützung für einen integrierten und weitgehend automatisierten Softwareentwicklungsprozess angestrebt.

In den folgenden Kapiteln werden wir die einzelnen Werkzeuge, deren Integrationsstufen und entsprechende Design- und Architekturvarianten eingehender diskutieren. DesignIt ist ein erster Prototyp einer Referenzimplementierung [Desi01]. DesignIt wird uns in den folgenden Kapitel als Anschauungsbeispiel dienen. Die aktuelle Implementierung von DesignIt beinhaltet eine erste Version der Komponenten: Modellierungs- und Spezifikationswerkzeuge, Generierung von Programmcode und Ausführungs- und Testumgebung.

Im Vordergrund bei der Realisierung von DesignIt stand dabei stets möglichst viele existierende Konzepte, Techniken, Komponenten und Infrastrukturen wiederzuverwenden. Nur so ist es möglich eine umfassende und durchgängige Werkzeugunterstützung für den evolutionären Architekturentwurf aufzubauen die in der Praxis eingesetzt werden kann und deren Entwicklung und Wartung nicht zuviel Zeit und Aufwand in Anspruch nimmt.

8.2 Modellierungs- und Spezifikationswerkzeuge

Das Modellierungs- und Spezifikationswerkzeug ist das zentrale Werkzeug für den evolutionären Architekturentwurf. Mit ihm kann der Softwarearchitekt die grafischen sowie textbasierten Architekturspezifikationen erstellen und bearbeiten. Außerdem erlaubt dieses Werkzeug die Verwaltung von Projekten, deren Daten und die darin enthaltenen Spezifikationen. Und schließlich ist das Modellierungs- und Spezifikationswerkzeug der gemeinsame Rahmen und der zentrale Einstiegspunkt für alle anderen Werkzeuge, die der Benutzer verwendet.

Mit dem Siegeszug der Objektorientierung und der UML sind eine Reihe von Entwicklungs- und Modellierungswerkzeugen entstanden, die verstärkt auch in der Industrie eingesetzt werden. Rational Rose [Rati01] und Together [Toge01] sind die wohl bekanntesten Vertreter dieser Werkzeuge. Jahrelange Entwicklungsarbeit und große Investitionen sind in diese Werkzeuge geflossen. Die Ergebnisse können sich durchaus sehen lassen. Die Werkzeuge werden inzwischen erfolgreich in industriellen Projekten eingesetzt und haben sich dort bewährt. Trotzdem weisen sie noch eine Reihe von Schwächen auf. So können sie beispielsweise meist nur unter großem Anpassungs- und Konfigurationsaufwand effektiv eingesetzt werden (vgl. auch [OHJ+99, BS00]).

Auf Grund dieser nicht zu vermeidenden individuellen Anpassung und Konfiguration des Modellierungswerkzeuges für ein konkretes Projekt, sehen sich die Hersteller von Werkzeugen immer stärker den Kundenforderungen nach Standardisierung und Offenheit gegenüber.

Dieser Druck hat inzwischen dazu geführt, dass einige Standards entstanden sind und die Werkzeuge eine Reihe von dokumentierten Schnittstellen und Erweiterungsmöglichkeiten unterstützen.

So steht beispielsweise mit XMI [OMG00c, OMG00b] ein breit akzeptierter Standard für den Datenaustausch zwischen Modellierungswerkzeugen zur Verfügung, der von fast allen Produkten und Herstellern unterstützt wird. Darüber hinaus bietet fast jedes Werkzeug dokumentierte Schnittstellen an, die es ermöglichen, neue Modellierungselemente und Beschreibungstechniken sowie zusätzliche Funktionsbausteine, wie zum Beispiel Generatoren und Prüfmechanismen zu integrieren.

Offensichtlich steigt dabei der Aufwand entsprechend der anvisierten Integrationsstufe (vgl. Kapitel 8.1). Generell kann aber jede Integrationsstufe erreicht werden, wie in den Arbeiten von Alexander Egyed und Robert Balzer am Beispiel der Integration von Rational Rose und Matlab/Stateflow anschaulich demonstriert wird (siehe [EB01]).

Im Gegensatz zu einer eigenständigen Implementierung eines Modellierungs- und Spezifikationswerkzeuges ist die Anpassung und Erweiterung existierender Werkzeuge meist günstiger. Die umfangreichen und aufwendigen Funktionen der Projektverwaltung und Diagrammeditoren können in der Regel direkt übernommen werden.

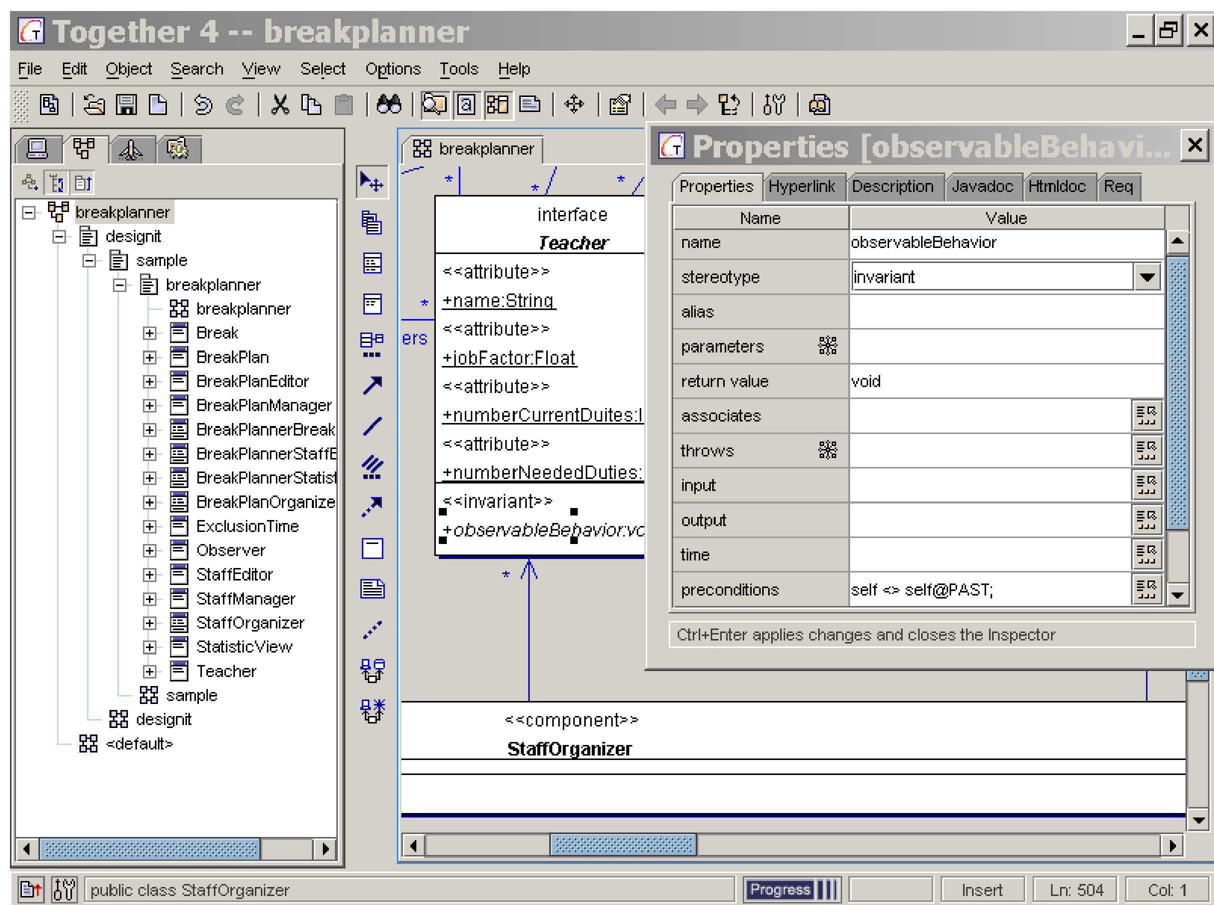

Abbildung 8.2: Architekturspezifikation des Pausenplaners in Together

Für die Realisierung unseres Werkzeugprototyp DesignIt haben wir uns deshalb die Modellierungsfunktionalität von Together erweitert und kein eigenständiges Modellierungswerkzeug entwickelt. Abbildung 8.2 zeigt ein Komponentendiagramm des Pausenplaners, das gerade mit Together bearbeitet wird. Die von uns entwickelten Diagramme enthalten zusätzliche In-

formationen, die man in Together nicht direkt darstellen kann. Bei jedem grafischen Modellierungselement kann man aber in Together diese zusätzlichen Informationen hinterlegen. So kann die Bedingung und das Verhalten der Invariante `observableBehavior` der Schnittstelle `Teacher` in dem abgebildeten Properties-Dialog bearbeitet werden. Die Bedingung wird in dem Feld für Vorbedingungen (*preconditions)* eingegeben und das Verhalten in dem nicht sichtbaren Tableau für zusäzliche Beschreibungen (*Description)* (vgl. Abbildung 8.2).

Mit diesem Werkzeug kann der Entwickler innerhalb seiner gewohnten Umgebung Architekturspezifikationen komponentenbasierter Systeme erstellen, entsprechend der Syntax die in den vorhergehenden Kapiteln 6 und 7 erarbeitet wurde.

8.3 Konsistenz- und Integrationsüberprüfung

Hat der Entwickler eine Architekturspezifikation evolutionär weiter entwickelt oder eine neue erstellt, so sind einige Konsistenz- und Integrationsüberprüfungen durchzuführen, bevor die Spezifikation weiter verarbeitet werden kann. Dabei unterscheiden wir die folgenden Prüfmechanismen:

- Konsistenzinspektor (*Consistency Inspector*)
- Evolutionsinspektor (*Evolution Inspector*)
- Integrationsinspektor (*Integration Inspector*)

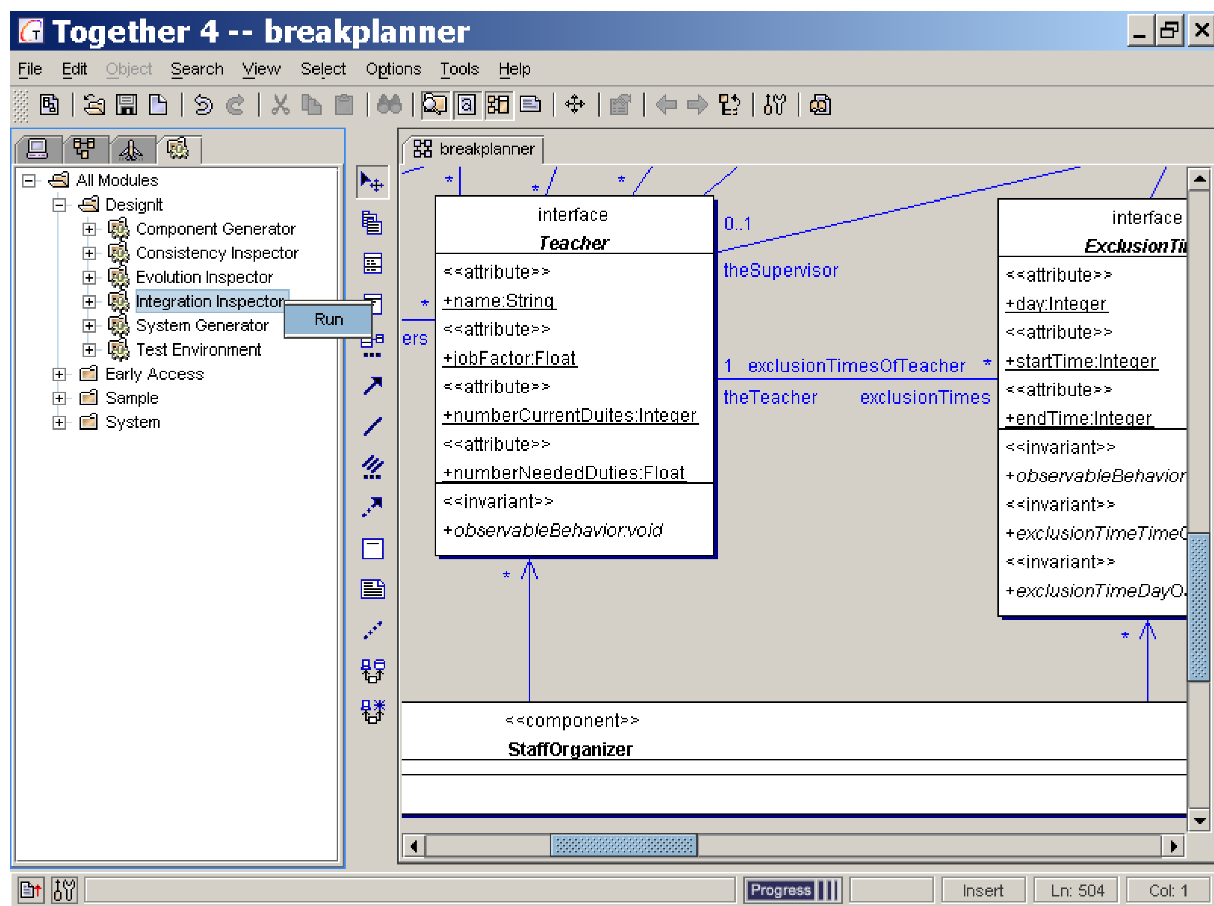

Abbildung 8.3: DesignIt Prüfmechanismen für Architekturspezifikationen

Mit DesignIt können diese Prüfmechanismen direkt aus dem Modellierungswerkzeug Together aufgerufen werden. In der Modulansicht im Projektfenster (links in Abbildung 8.3)

stehen unter dem Modulordner DesignIt alle Prüfmechanismen als eigenständige Modele in Together zur Verfügung und können direkt ausgeführt werden.

Wie im vorhergehenden Kapitel 8.2 beschrieben, verwendet DesignIt als Modellierungswerkzeug direkt Together. Ein Entwickler kann somit auch Spezifikationen entwerfen, die syntaktisch nicht den in Kapitel 6 und 7 definierten Diagrammarten entsprechen. Mit dem Konsistenzinspektor kann der Entwickler die syntaktische Konsistenz von Architekturspezifikationen überprüfen. Darüber hinaus werden noch zusätzliche Konsistenzbedingungen kontrolliert, wie zum Beispiel, dass alle Schnittstellenbezeichner, die von Verbindungsspezifikationen verwendet werden, auch definiert und Komponenten zugeordnet sind.

Der Evolutionsinspektor zeigt dem Entwickler die Veränderungen zwischen zwei Architekturmodellversionen. Dem Entwickler wird jeweils eine Liste der gelöschten, veränderten und neuen Komponenten und Schnittstellen präsentiert. Zu jedem Element in der Liste kann der Entwickler sich das entsprechende Spezifikationsartefakt anzeigen lassen und so detaillierte Informationen über den Evolutionsschritt bekommen.

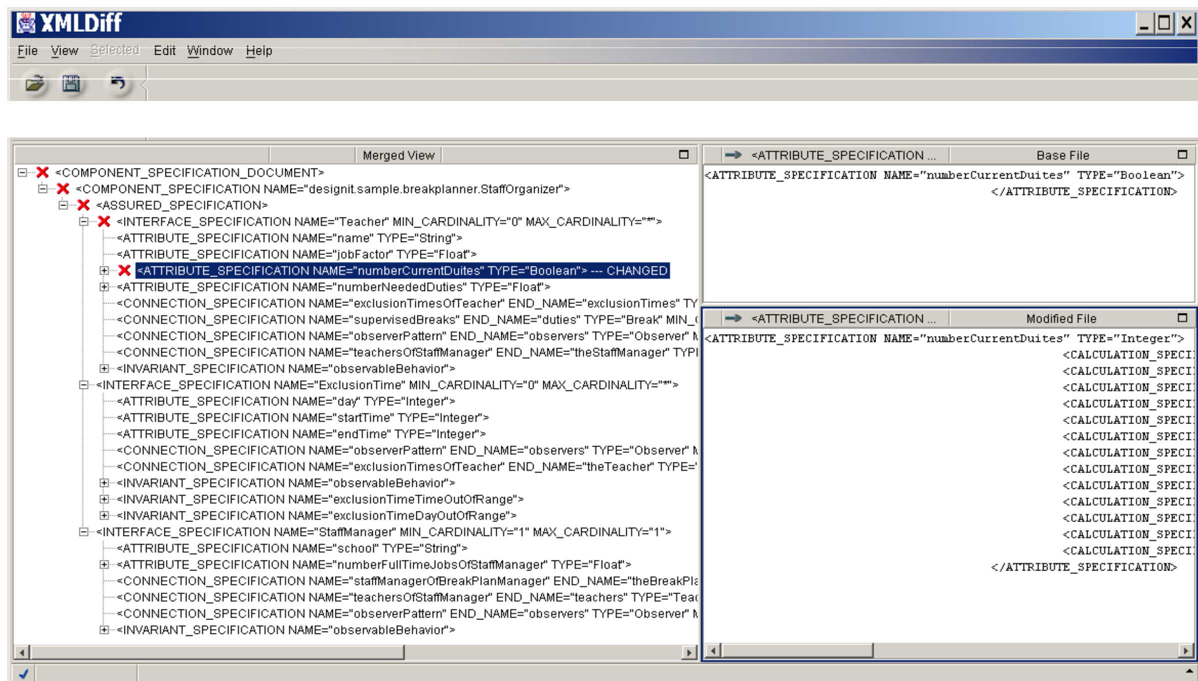

Abbildung 8.4: Evolutionsinspektor für Architekturspezifikation mit XMLDiff

Darüber hinaus ist es auch möglich die Veränderungen im Rahmen eines Evolutionsschrittes bis auf die Ebene einzelner Methoden und Attribute darzustellen. Hierzu eigenen sich speziell XML-basierte Technologien, mit denen relativ einfach Differenzen in XML-Bäumen berechnet und dargestellt werden können (siehe auch [HM01]). Die textbasierten Spezifikationstechniken aus Kapitel 6 und 7 entsprechen bis auf die XML spezifische Erweiterungen direkt XML-basierten Spezifikationen. Abbildung 8.4 illustriert den Einsatz dieser Technik anhand des Werkzeuges XMLDiff [XMLD01]. Links in der Abbildung wird der aus dem Vergleich zweier Spezifikationen resultierende XML-Baum dargestellt. Das „x“ an einem Knoten in dem Baum zeigt, dass innerhalb des Teilbaumes dieses Knotens eine Veränderung zwischen der ursprünglichen und der neuen Spezifikation erkannt wurde. Markiert man diesen Knoten, so kann man in den zwei linken Fenstern jeweils die entsprechenden Artefakte aus der ursprünglichen und der neuen Spezifikation sehen und so den Evolutionsschritt nachvollziehen. In Abbildung 8.4 hat sich hierbei der Typ des Attributes `numberCurrentDuties` von `Boolean`

auf `Integer` geändert. Außerdem ist noch eine Berechnungsspezifikation für das Attribut hinzugefügt worden. Mit diesem Verfahren lassen sich die stabilen, neuen und veränderten Eigenschaften von evolutionär weiter entwickelten Spezifikationen ermitteln (vgl. Definition 3.13, Definition 3.14 und Definition 3.15).

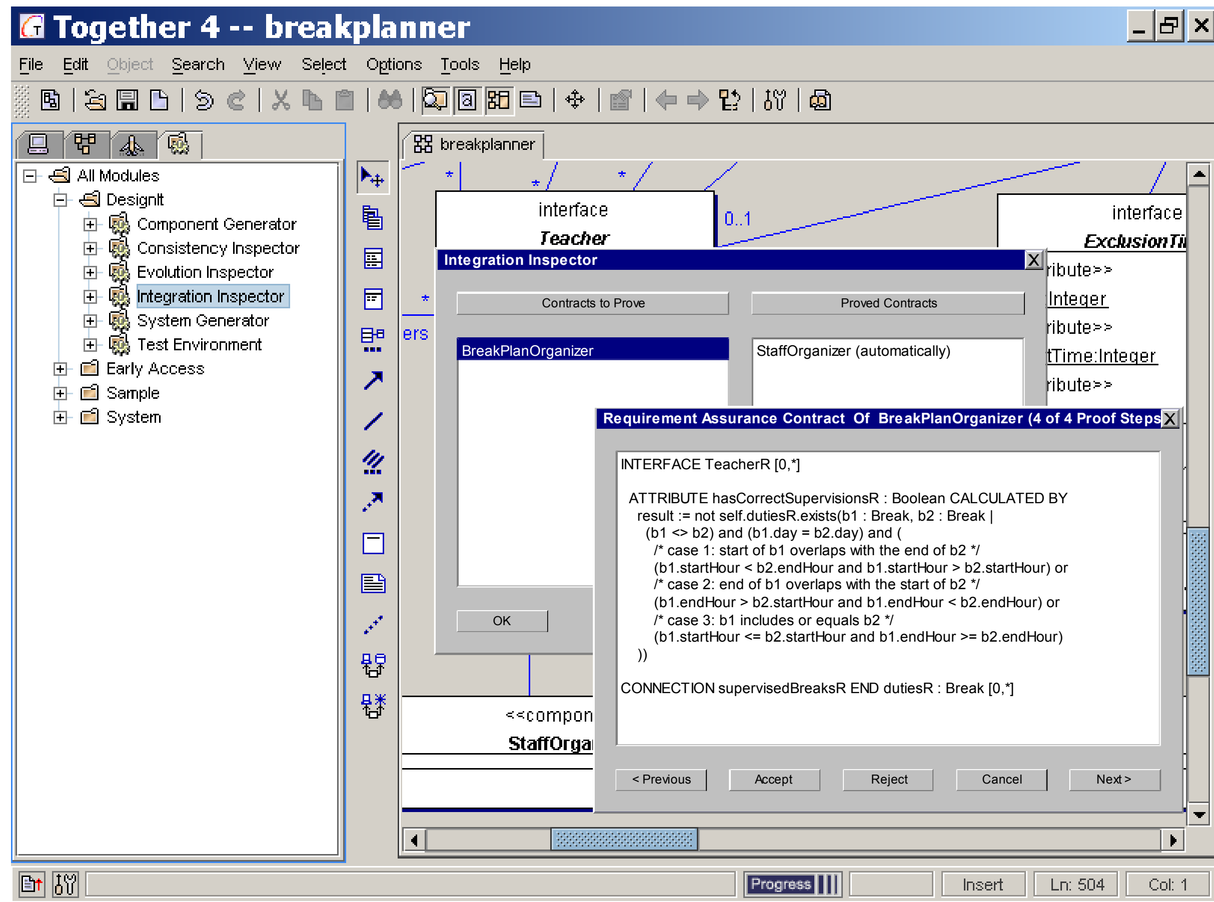

Abbildung 8.5: DesignIt Integrationsinspektor für Architekturspezifikation

Schließlich, mit Hilfe des Integrationsinspektors kann der Entwickler die Annahme-/Zusicherungsverträge einer Architekturspezifikation überprüfen. Der Integrationsinspektor versucht entsprechend dem Schema aus Kapitel 7 die syntaktische Validierung von Annahme-/Zusicherungsverträgen durchzuführen. Zu diesem Zweck führt er zuerst die Spezifikationsbezeichnerersetzungen beim Beweisziel und beim Beweisaxiom durch. Sind die resultierenden Spezifikationsartefakte äquivalent, so ist der Vertrag gültig. Ansonsten werden, wie in Abbildung 8.5 illustriert, die einzelnen Beweisschritte dem Entwickler gezeigt. Er kann dann die Korrektheit des Beweis und somit die Gültigkeit des Vertrages bestätigen oder ablehnen. Sind alle Verträge gültig, so ist die Architekturspezifikation vollständig. Der Integrationsinspektors realisiert somit die in Definition 3.11 definierten Prädikate über Spezifikationen: $\mathsf{consistent}(\mathsf{eas})$ und $\mathsf{complete}(\mathsf{eas})$.

8.4 Generierung von Programmcode

Wurde eine Architekturspezifikation mit einem Modellierungs- und Spezifikationswerkzeug entworfen und alle Konsistenz- und Integrationsüberprüfungen erfolgreich durchgeführt, so kann aus der Spezifikation Programmcode generiert werden. Dieser wird dann in einer Ausführungs- und Testumgebung getestet. Wie in Abbildung 8.3 bereits dargestellt ist, wird bei

DesignIt der Programmcode von zwei unabhängigen Generatoren erzeugt, dem Komponentengenerator (*Component Generator*) und dem Systemgenerator (*System Generator*).

Der Komponentengenerator erzeugt aus Komponentendiagrammen (vgl. Abbildung 7.10) den Programmcode für die Komponenten, die zugehörigen Schnittstellen und ihre Attribute, Verbindungen, Nachrichten sowie Invarianten. Der Systemgenerator hingegen erzeugt aus Systemdiagrammen (vgl. Abbildung 7.11) den Programmcode für ein komponentenbasiertes System, der die Instanzierung und Initialisierung des Systems implementiert.

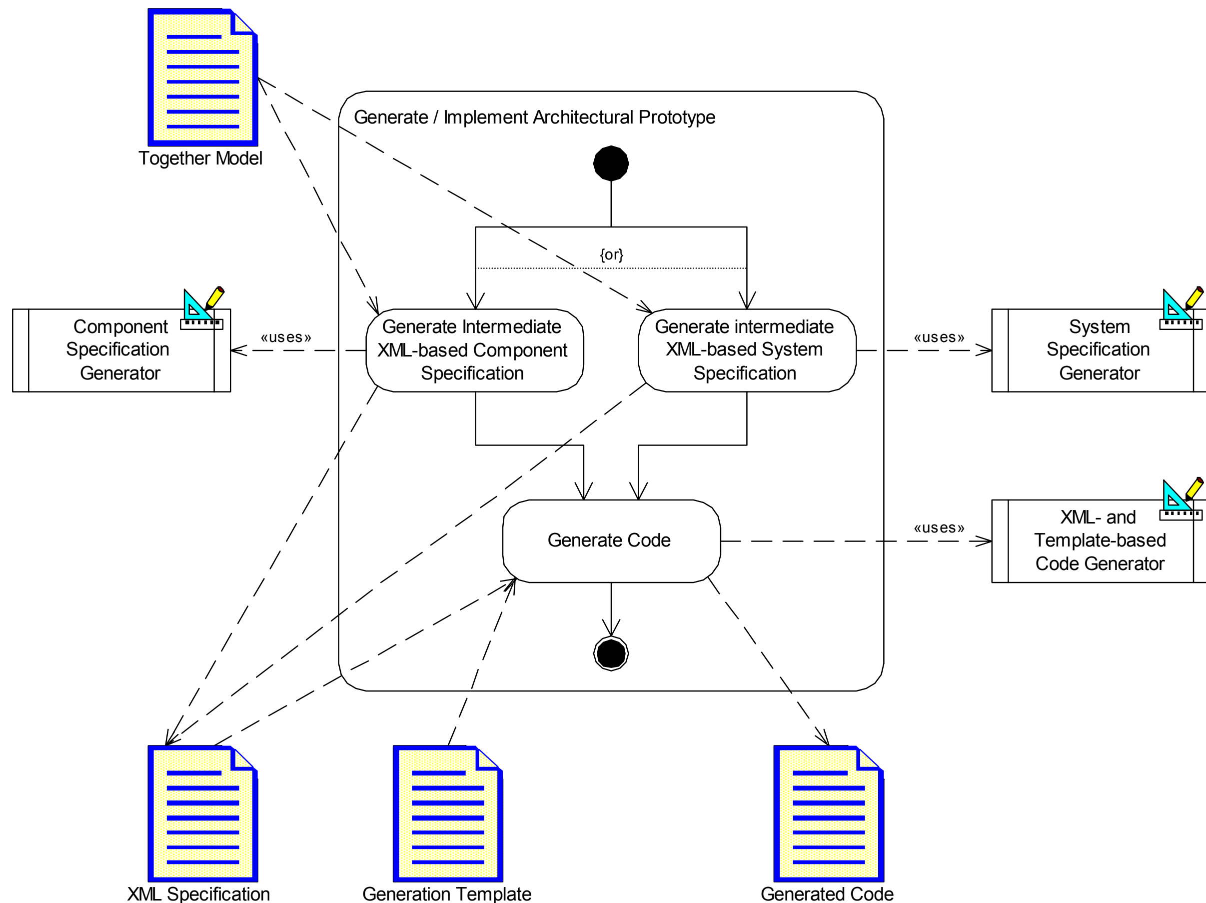

Abbildung 8.6: DesignIt Generatoren für Architekturspezifikationen

Generatoren und Compiler verwenden häufig Zwischenformate. Die Komponenten für die Code-Optimierung und Code-Erzeugung können so für mehrere Eingabeformate verwendet werden [ASU88a, ASU88b]. Bei der Entwicklung der DesignIt Generatoren haben wir diese Technik ebenfalls angewendet.

Wie in Abbildung 8.6 dargestellt ist, erzeugen sowohl der Komponentengenerator als auch der Systemgenerator jeweils aus einer Architekturspezifikation, die mit Together erstellt wurde, eine XML-basierte Spezifikation. Diese XML-basierten Spezifikationen sind identisch mit den textbasierten Spezifikationstechniken aus Kapitel 6 und 7, bis auf die zusätzlich notwendigen XML-Elemente (siehe auch [HM01]).

Mit Hilfe des XML- und Template-basierten Code-Generators kann aus den XML-basierten Spezifikationen der eigentliche Programmcode generiert werden. Dazu ersetzt der Code-Generator die in den Code-Templates enthaltenen Platzhalter mit Elementen aus der XML-basierten Spezifikation (vgl. Abbildung 8.6).

Durch die Templates und das XML-basierte Eingabeformat ist der XML- und Template-basierte Code-Generator so flexibel, dass er auch für beliebige andere Generierungen verwendet werden kann. Insbesondere kann auch Programmcode für eine andere Plattform, Programmiersprache und Zielumgebung erzeugt werden. Hierfür müssen nur die entsprechenden Code-Templates geändert werden. Eine detaillierte Beschreibung des XML- und Template-basierten Code-Generators ist in [Kivl00] und [Raus01a] zu finden.

8.5 Ausführungs- und Testumgebung

Nach erfolgreicher Generierung kann der Architekturprototyp in einer Ausführungsumgebung kontrolliert ausgeführt und getestet werden. So wird eine frühzeitige Rückkopplung des Entwurfs erreicht, ein wesentlicher Bestandteil des evolutionären Architekturentwurfs. Die Ausführung des Architekturprototypen können wir dabei in unterschiedliche Kategorien einteilen:

- Produktiv-Ausführung
- Simulations-Ausführung
- Debug-Ausführung
- Testfall-Ausführung

Bei der Produktiv-Ausführung wird der generierte Architekturprototyp oder eine modifizierte Version des Prototypen innerhalb eines produktiven Systems ausgeführt. Dieses System enthält dann meist noch eine zusätzliche Komponenten, wie zum Beispiel eine Benutzeroberfläche oder eine Datenbankanbindung.

Bei der Simulations-Ausführung hingegen wird der Architekturprototyp innerhalb einer simulierten Umgebung ausgeführt. Die Benutzeroberfläche beispielsweise könnte durch ein Design-Werkzeug generiert werden und die Datenbankanbindung könnte im Speicher simuliert werden. Wesentlich dabei ist, dass während der Simulation die Ausführung durch den Benutzer kontrolliert und inspiziert werden kann. So kann der Benutzer beispielsweise die Simulation an bestimmten Stellen anhalten und den Zustand des Systems inspizieren und unter Umständen sogar verändern (vgl. auch [HMR+98]).

Die Debug-Ausführung ist ähnlich zu der Simulations-Ausführung. Allerdings wird dabei keine spezifische Umgebung simuliert, sondern der Architekturprototyp in einer generischen Debug-Umgebung ausgeführt. Der Benutzer kann dabei Haltepunkte definieren, Einzelschritte ausführen, das System inspizieren und verändern. Die Debug-Ausführung wird von den Entwicklern meist zur Analyse von Fehlern und zur Durchführung von Ad-hoc-Testfällen verwendet.

Unter der Testfall-Ausführung verstehen wir die automatische Durchführung und Überprüfung von Regressionstests. Dies ist speziell beim evolutionären Entwurf von entscheidender Bedeutung. Wenn hierbei nicht automatisiert wird, müssen die Testfälle mehrfach von Hand durchgeführt werden, was einen gesteigerten Aufwand zur Folge hat. Erfolgreich absolvierte Regressionstests garantieren, dass alte Funktionalität auch nach einem evolutionären Entwurfsschritt noch in der gewünschten Weise zur Verfügung steht (vgl. auch [HMR+98, BFR00]).

Gleichgültig, welche Ausführungsvariante man wählt, das Verhalten des Systems muss stets identisch sein. Alle Ausführungsvarianten sollten deshalb auf ein und derselben Ausführungsumgebung basieren. Im Rahmen des Werkzeugprototyps DesignIt haben wir eine entsprechende Ausführungsumgebung implementiert. Abbildung 8.7 zeigt die Debug-

Ausführung des Pausenplaners innerhalb von DesignIt. Der Entwickler kann die Debug-Ausführung starten und anhalten, einzelnen Ausführungsschritte ausführen, den Systemzustand inspizieren, Attributwerte verändern und Nachrichten an Schnittstellen anlegen.

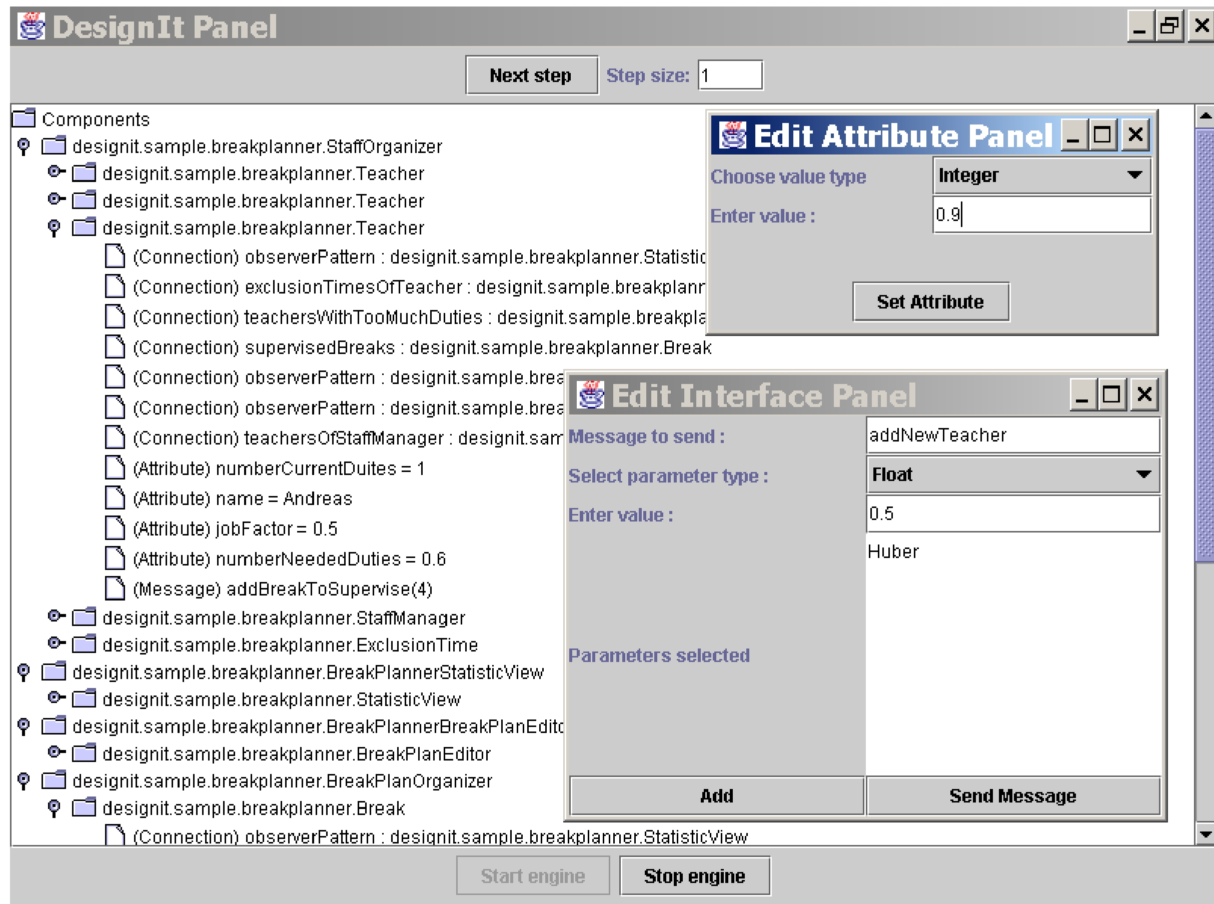

Abbildung 8.7: DesignIt Debug-Ausführung der Pausenplanerspezifikation

Wie bereits erwähnt basiert diese Debug-Ausführung, wie jede andere Ausführungsvariante auf einer einheitlichen Ausführungsumgebung. Diese Ausführungsumgebung ist eine Implementierung des Systemmodells aus Kapitel 5.

Abbildung 8.8 zeigt das Klassendiagramm der Ausführungsumgebung von DesignIt. Die Debug-Ausführung (*DebugExecution*) von DesignIt in Abbildung 8.7 ist dabei ein spezieller Ausführungsinspektor (*ExecutionInspector*). Ein Ausführungsinspektor führt ein System kontrolliert innerhalb einer Laufzeitumgebung (*RuntimeEnvironment*) aus. Die Laufzeitumgebung besteht aus einer Menge von Ausführungsknoten (*ExecutionNode*). Ein Ausführungsknoten ist nur für die Verwaltung einer entfernt ausgeführten Menge von Komponenten (*Component*) verantwortlich. Jeder Komponente ist eine Menge von Schnittstellen (*Interface*) zugeordnet. Eine Schnittstelle besitzt eine Menge von Attributen (*Attribute*), hat Verbindungen (*Connection*) zu anderen Schnittstellen und es können eine Menge von Nachrichten (*Message*) zur Verarbeitung anliegen.

Entsprechend des Systemmodells aus Kapitel 5 verarbeitet jede Komponente in einen Ausführungsschritt alle Nachrichten, die an ihren Schnittstellen anliegen. Bei der Abarbeitung dieser Nachrichten können neue Nachrichten versendet, Werte von Attributen verändert und Komponenten, Schnittstellen sowie Verbindungen erzeugt bzw. gelöscht werden. Entsprechend dem Systemmodell aus Kapitel 5 kann die Komponente diese Operationen nicht selbst durch-

führen. Deshalb erzeugen die Komponenten entsprechende Spezialisierungen der Klasse *Operation* und übergeben diese Aufträge der Laufzeitumgebung. Haben alle Komponenten ihre Nachrichten abgearbeitet, so beginnt die Laufzeitumgebung alle anstehenden Operationen durchzuführen. Dies kann wiederum dazu führen, dass neue Nachrichten an Schnittstellen angelegt werden, die dann im nächsten Ausführungsschritt von den zugehörigen Komponenten bearbeitet werden.

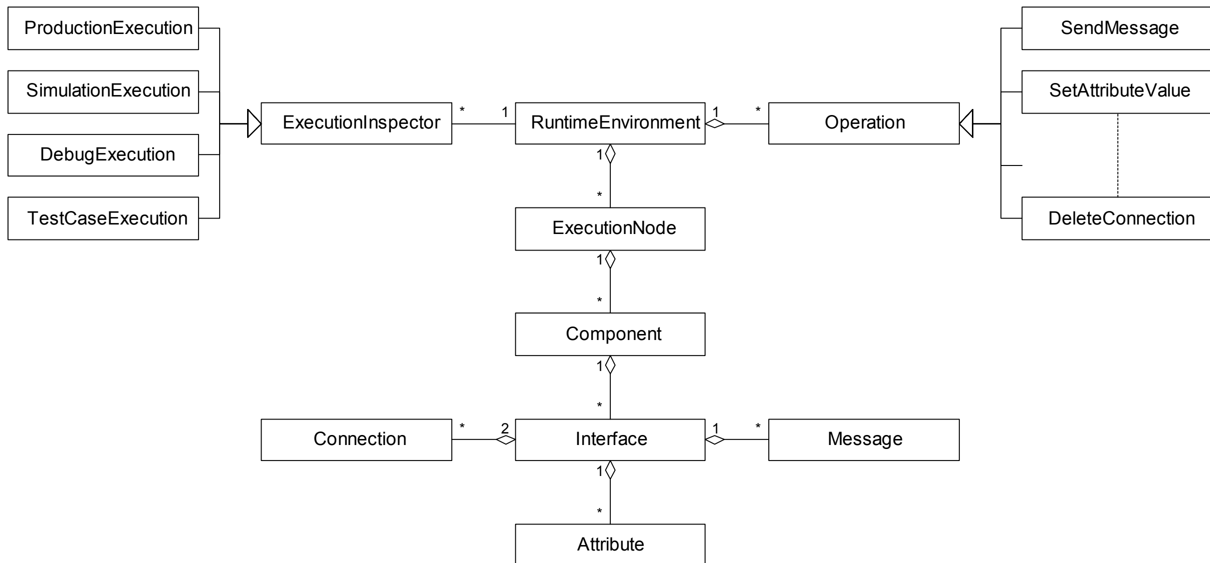

Abbildung 8.8: UML Klassendiagramm der Ausführungsumgebung von DesignIt

Die aktuelle Implementierung der Ausführungsumgebung von DesignIt basiert auf der Java 2 Enterprise Edition [SUN01] und CORBA [OMG01b]. Die grundlegenden Konzepte sind aber von genereller Natur, so dass die Ausführungsumgebung relativ einfach auch auf Java Enterprise Beans oder CORBA Components abgebildet werden kann (vgl. auch [Haas00]).

Eine detaillierte Beschreibung der Implementierung der Laufzeitumgebung und der Debug-Ausführungsumgebung sind den Arbeiten [Lave00] und [HP01] zu entnehmen. Die Umsetzung einer Produktiv- oder Simulations-Ausführung basiert ebenfalls auf diesen Konzepten und unterscheidet sich nicht wesentlich von der Debug-Ausführung (vgl. auch [HMR+98]).

Die Testfall-Ausführung hingegen ist die einzige Variante bei der die Ausführung und deren Auswertung automatisiert durchgeführt werden. Eine Benutzerinteraktion während der Ausführung des Architekturprototypen ist nicht notwendig und nicht erwünscht. Grundlage der Testfall-Ausführung sind Testfallspezifikationen. Eine Testfallspezifikation beinhaltet folgende Elemente:

- Die zu testende Komponente bzw. das zu testende System, der Testling, wird festgelegt.
- Der Initialzustand des Testlings wird definiert.
- Die Interaktion zwischen Umgebung und Testling, das Testszenario, wird beschrieben.
- Der erwartete Endzustand des Testlings wird spezifiziert.

Für die eigentliche Testfall-Ausführung muss der Testling in den Initialzustand gebracht, das Testszenario auf dem Testling ausgeführt und schließlich der tatsächliche Endzustand mit dem erwarteten Endzustand verglichen werden. Der Testverlauf und das Testergebnis sind dann in einem Testbericht festzuhalten. Insbesondere sollten hier Abweichungen des tatsächlichem Verhaltens vom erwarteten Verhalten dokumentiert werden.

In [BFR00] haben wir eine entsprechende Testfall-Ausführungsumgebung konzipiert und realisiert. Als Ausgangsbasis diente dabei das Test-Framework JUnit von Kent Beck und E-rich Gamma [BG01]. Diese Testfall-Ausführungsumgebung ist allerdings noch nicht in DesignIt integriert. Allerdings sollte die Integration relativ einfach möglich sein. Eine entsprechende Beschreibungstechnik für Testfallspezifikationen basierend auf Sequenzdiagrammen haben wir bereits in Abbildung 6.7 vorgestellt. Der XML- und Template-basierten Code Generator aus Kapitel 8.4 sollte es mit vertretbarem Aufwand ermöglich aus diesen Testfallspezifikationen Programmcode für die Testfall-Ausführung zu generieren. Die generierten Testfälle könnten dann direkt in der Ausführungsumgebung von DesignIt ausgeführt werden.

8.6 Versions- und Migrationsunterstützung

Abgesehen von einer funktionierenden Entwicklungsumgebung ist ein Werkzeug für das Versions- und Konfigurationsmanagement eine Grundvoraussetzung für jedes Projekt. Nur wenn man mehrere Versionen einer Datei verwalten kann, lässt sich die Historie nachzeichnen. Dies ist zum Nachvollziehen von Entscheidungen wichtig. Wird von einem Kunden ein Fehler gemeldet, so ist es essentiell, dass man exakt die Systemversion wieder herstellen kann, bei der beim Kunden der Fehler aufgetreten ist. Generell, bei Problemen in der Entwicklung ist es hilfreich, wieder auf einem definierten Punkt aufzusetzen zu können

Gerade beim evolutionären Vorgehen entstehen sehr viele Versionen der einzelnen Spezifikations-, Modellierungs- und Programmierungsartefakte, die ohne Versions- und Konfigurationsmanagement nicht mehr verwaltet werden können. Zwischen diesen versionierten Artefakten existieren Beziehungen, die wiederum auch der Versionierung unterliegen. Eine Menge solcher Artefakte und deren Beziehungen, jeweils in einer spezifischen Version, ergeben zusammen konsistente Modelle, sogenannte Konfigurationen.

Für die Versionsverwaltung existieren eine Reihe freier und kommerzieller Produkte, die bereits in vielen Projekten erfolgreich eingesetzt wurden. Die bekanntesten Vertreter auf Unix-Systemen sind sccs [Poch75] und rcs [Tich85]. PVCS von Intersolv [Inte01] und Clear Case von Rational [Rati01] sind neuere kommerzielle Produkte, die ebenfalls häufig eingesetzt werden. Einige Konzepte zur Integration dieser Produkte in eine übergreifende Entwicklungsumgebung wurden in [BHRS97] bereits vorgestellt. Eine tiefergreifende Diskussion bezüglich Versions- und Konfigurationsmanagement ist in [Rass98] zu finden.

Ist eine neue Version eines Modells entstanden, so hat dies unter Umständen Auswirkungen auf andere, davon abhängige, Modelle. Ändert sich beispielsweise das Datenmodell eines Systems, so muss mit der Auslieferung des neuen Systems das Schema einer beim Kunden existierenden Datenbank angepasst werden, sonst würde er unter Umständen seine Daten verlieren. Für die Akzeptanz des Systems wäre dies nicht gerade förderlich.

Die selbe Problematik tritt bereits während des evolutionären Architekturentwurfs auf: Wurde beispielsweise in einem Evolutionsschritt eine neue Version der Architekturspezifikation entwickelt, so müssen die Beschreibungen der Initialzustände in allen Testfallspezifikationen entsprechend dem neuen Architekturmodell angepasst werden.

Die Migration von Modellen bzw. Modellartefakten, die abhängig von anderen Modellen bzw. Modellartefakten sind, kann durch entsprechende Werkzeuge unterstützt werden. Für eine automatisierte Migration sind neben einer Beschreibung des ursprünglichen und des neuen Modells auch eine Beschreibung des Migrationsweges notwendig. Der Migrationsweg be-

steht aus einer Reihe von primitiven Operationen, die man auf Modellen ausführen kann, wie zum Beispiel das Hinzufügen, Löschen oder Verändern von Attributen.

Dieser Migrationsweg kann im allgemeinen nicht aus dem ursprünglichen und dem neuen Modell berechnet werden [Kim91, Wall91]. Ob ein Attribut verändert, oder ob das Attribut gelöscht und dann ein neues, mit dem selben Namen, hinzugefügt wurde kann nicht von einem Werkzeug entschieden werden. Diese Information muss der Entwickler in der Beschreibung des Migrationsweges hinterlegen. Der Migrationsweg bestimmt dabei den Effekt der Migration. Wird ein Attribut verändert, so müssen beispielsweise bei der Datenbankmigration die Daten des Attributes migriert werden. Wird das Attribut gelöscht und ein neues hinzugefügt, so müssen die Daten des Attributes gelöscht und die Daten das neue Attribut initialisiert werden. Obwohl diese zwei unterschiedlichen Migrationswege verschiedene Auswirkungen auf die Datenbank haben, ist das resultierende Datenbankschema identisch.

Diese grundlegenden Konzepte eines entsprechenden Migrationswerkzeugs haben wir bereits in [Keie99a] und [KR01] vorgestellt und in entsprechenden Prototypen umgesetzt. Diese Werkzeuge sind allerdings noch nicht in DesignIt integriert. Spätestens mit der Integration der Testfallablaufumgebung sollte ein entsprechendes Werkzeug für die automatisierte Migration der Testfälle vorhanden sein. Darüber hinaus sind die vorgestellten Konzepte für die Migrationsunterstützung von genereller Natur. Ein entsprechendes Werkzeug kann für die verschiedensten Modellmigrationen im Rahmen des evolutionären Architekturentwurfs hilfreich sein. Die in der Literatur unter dem Schlagwort „Refactoring“ bekannten Techniken zur Modellevolution könnten mit einem Werkzeug zur Migrationsunterstützung automatisiert durchgeführt werden (vgl. auch [Opdy92, Fowl99]).

8.7 Zusammenfassung

Der Erfolg einer Methodik basiert nicht zuletzt auf der Qualität und Durchgängigkeit der Werkzeugunterstützung. Für eine optimale und weitreichende Unterstützung des evolutionären Architekturentwurfs sind die folgenden Werkzeuge besonders hilfreich: Modellierungs- und Spezifikationswerkzeuge, Konsistenz- und Integrationsüberprüfung, Generierung von Programmcode, Ausführungs- und Testumgebung sowie Versions- und Migrationsunterstützung.

Mit einem Modellierungs- und Spezifikationswerkzeug bearbeitet und verwaltet der Architekt Projekte und die darin enthaltenen Architekturspezifikationen. Für die Realisierung eines entsprechenden Modellierungs- und Spezifikationswerkzeuges kann man entweder auf bestehende CASE-Tools zurückgreifen, oder einen eigenständiges Werkzeug entwickeln. Bei der Implementierung unseres Werkzeugprototypen DesignIt haben wir uns für die erste Variante entschieden und das kommerziell verfügbare Modellierungswerkzeug Together entsprechend erweitert. Der Entwicklungsaufwand für DesignIt hat sich dadurch enorm reduziert. Der wesentliche Nachteil ist jedoch, dass der Architekt jetzt auch Spezifikationen erstellen kann, die syntaktisch nicht korrekt sind.

Deshalb muss ein zusätzlicher Prüfmechanismus realisiert werden: Der Konsistenzinspektor überprüft die Syntax von Architekturspezifikationen. Speziell für den evolutionären Architekturentwurf sind zwei weitere Konsistenz- und Integrationsüberprüfungen wichtig. Der Evolutionsinspektor visualisiert dem Entwickler die einzelnen Veränderungen in einem Architekturmodell, die im Laufe eines Evolutionsschrittes durchgeführt wurden. Der Integrationsinspektor validiert in Interaktion mit dem Benutzer die Annahme-/Zusicherungsverträge einer Architekturspezifikation. Damit wird die Integrationsproblematik bereits auf der Modellebene angegangen.

Hat eine Architekturspezifikation allen Überprüfungen stand gehalten, so kann daraus ein Architekturprototyp generiert werden. Bei DesignIt besteht die Generierung aus zwei Stufen. Zuerst wird aus den grafischen UML Diagrammen eine XML-basierte Spezifikation erzeugt. Mit dem XML- und Template-basierten Code-Generator wird dann aus den XML-basierten Spezifikationen der eigentliche Programmcode generiert. Dieser Code-Generator ermöglicht es durch Veränderung der Templates Programmcode für andere Programmiersprachen, Technologien und Plattformen zu erzeugen.

Der generierte Architekturprototyp kann dann in einer Ausführungs- und Testumgebung ausgeführt werden. In dieser Umgebung kann der Entwickler den Prototypen kontrolliert ausführen, inspizieren und verändern. Fehlersuche, Fehleranalyse, Durchführung von Ad-hoc-Testfällen und Ausführung von Regressionstests sind damit möglich.

Und schließlich sind gerade beim evolutionären Architekturentwurf eine effektive Versions- und Migrationsunterstützung von entscheidender Bedeutung. Für das Versions- und Konfigurationsmanagement existieren ausgereifte Produkte die in den Projekten erfolgreich eingesetzt werden. Für die Migrationsunterstützung haben wir die Konzepte eines Werkzeuges vorgestellt, das es uns erlaubt den Migrationsweg als eine Sequenz von primitiven Operationen auf einem Modell oder einem Teilmodell zu beschreiben und dann automatisiert durchzuführen.

9 Zusammenfassung und Ausblick

Die Entwicklung von Softwaresystemen findet heute in einem hochgradig dynamischen Umfeld statt. Typischerweise lässt es sicht nicht vollständig verhindern, dass sich die Anforderungen an das zu entwickelnde Softwaresystem ändern. Zusätzlich werden die technologischen Innovationszyklen immer kürzer, was dazu führt, dass neue Technologien in die laufenden Entwicklungen integriert werden müssen. Selbst die konzeptionellen und organisatorischen Rahmenbedingungen eines Projektes können sich noch während der Systementwicklung verändern.

Die Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme beugt den Risiken vor, die mit diesem Wandel verbunden sind. In iterativen Evolutionsschritten wird eine Softwarearchitektur entworfen, die einen stabilen und gut dokumentierten Rahmen bietet und Performanz und Zuverlässigkeit garantiert. Indem die Softwarearchitektur die Auswirkungen der Veränderung von Anforderungen, Technologien und Rahmenbedingungen so weit wie möglich abdämpft, unterstützt sie Flexibilität und Erweiterbarkeit des Systems. Formal fundierte pragmatische Beschreibungstechniken ermöglichen die Generierung von ablauffähigen Architekturprototypen. Mit der Ausführung dieser Prototypen in entsprechenden Testumgebungen kann bereits frühzeitig die Qualität des Architekturmodells untersucht und beurteilt werden.

Die vorliegende Dissertation stellt eine durchgängige Methodik für den evolutionären Architekturentwurf komponentenbasierter Systeme bereit, die den Softwarearchitekten bei der täglichen Arbeit nachhaltig unterstützt. Die Schlüsselkonzepte dieser Methodik sind die evolutionäre Modellierung von Softwarearchitekturen und die frühzeitige Rückkopplung durch Prototyping.

Der Architekturentwurf ist ein zentraler wenn auch kleiner Ausschnitt einer vollständigen und umfassenden Methode der Softwareentwicklung. Damit die vorliegenden Ergebnisse weiter entwickelt und andere Arbeiten integriert werden können, haben wir ein methodisches Rahmenwerk erarbeitet und dieser Arbeit zu Grunde gelegt. Elementare Bestandteile dieses Rahmenwerks sind ein Produktmodell, das alle Produkte der Softwareentwicklung und deren Beziehungen beschreibt, und Prozessmuster, die methodische Vorgehensweisen zur Verfügung stellen, um diese Entwicklungsprodukte zu erzeugen und zu bearbeiten.

Im Kontext dieses methodischen Rahmenwerks haben wir den Aufbau und die Struktur von Architekturspezifikationen definiert. Das Prozessmuster des evolutionären Architekturentwurfs beschreibt das methodische Vorgehen der inkrementellen Erstellung einer derartigen Architekturspezifikation.

In einem ersten Schritt werden ausgehend von einer Architekturskizze und einer Testspezifikation die Komponenten- und Schnittstellenspezifikationen sowie entsprechende Realisierungsspezifikationen für hierarchische Komponenten erstellt. Nach den notwendigen Konsistenz- und Integrationsüberprüfungen wird aus diesen Spezifikationen ein Architekturprototyp generiert. Dieser kann dann in einer Ausführungsumgebung anhand der Testspezifikation geprüft werden. Das Ergebnis wird in einem Testbericht festgehalten. Hat das Architekturmodell noch nicht alle Anforderungen erfüllt oder haben sich Anforderungen, Technologien oder Rahmenbedingungen verändert, so wird die nächste Iteration des evolutionären Architekturentwurfs gestartet.

Beschreibungstechniken mit einer systemmodellbasierten formalen Semantik ermöglichen eine weitgehend vollständige Generierung von lauffähigen Prototypen aus konsistenten Beschreibungen. Ist eine Spezifikation jedoch inkonsistent, so sind spezifischere Aussagen nicht mehr möglich. Gerade beim evolutionären Architekturentwurf komponentenbasierter Systeme sind aber weitergehende Fragestellungen essentiell, wie zum Beispiel die Frage, welche Veränderungen an einzelnen Komponentenbeschreibungen zu einem inkonsistenten Gesamtmodell geführt haben.

Eine Lösung dieses Problems liefert der neue, verbesserte Ansatz einer prädikatenbasierten formalen Semantik für Beschreibungstechniken. Dabei werden jeder Architekturspezifikation eine Menge von Eigenschaften in Form von logischen Ausdrücken mit freien Variablen zugeordnet. Ein System implementiert genau dann die Spezifikation, wenn sich alle Ausdrücke durch die Belegung der freien Variablen mit dem System in wahre Ausdrücke überführen lassen.

Darüber hinaus unterscheiden wir zwei Gruppen von Eigenschaften, die Komponentenspezifikationen zugeordnet werden: Die Menge der angenommenen Eigenschaften und die Menge der zugesicherten Eigenschaften. Die zugesicherten Eigenschaften müssen genau dann von der Komponente gewährleistet werden, wenn die angenommenen Eigenschaften durch die Umgebung bereit gestellt werden.

Mit sogenannten Annahme-/Zusicherungsverträgen werden die angenommenen Eigenschaften einer Komponente durch zugesicherte Eigenschaften anderer Komponenten gedeckt. Ändert sich nun eine Komponentenspezifikation im Zuge der evolutionären Modellierung, so ändern sich auch die Eigenschaften dieser Komponente. Durch erneutes Überprüfen der Annahme-/ Zusicherungsverträge kann man feststellen, ob sich die einzelnen Komponentenspezifikationen noch zu einem konsistenten Gesamtmodell integrieren lassen. Die Integrationsproblematik kann damit bereits auf der Modellebene angegangen und eine aufwendige nachfolgende Fehlerbehebung vermieden werden.

Das Systemmodell, das die Menge der komponentenbasierten Systeme charakterisiert, die wir in dieser Arbeit betrachten, basiert auf dieser formalen Grundkonzeption. Ein komponentenbasiertes System besteht dabei aus einer Menge von Komponenten. Jeder Komponente sind eine Menge von Schnittstellen zugeordnet. Schnittstellen haben Attribute mit einer aktuellen Wertebelegung. Zwischen Schnittstellen existieren Verbindungen. Schnittstellen verarbeiten Nachrichten und haben Invarianten. Bei der Abarbeitung von Nachrichten und Invarianten werden neue Nachrichten verschickt, Invarianten ausgelöst, Attributwerte verändert, und Komponenten, Schnittstellen und Verbindungen erzeugt und gelöscht. Das Verhalten eines komponentenbasierten Systems kann durch eine Folge von Systemzuständen beschrieben werden, die jeweils aus Daten-, Struktur- und Kommunikationszustand bestehen.

Das Komponentenverhalten entspricht einer Zustandsübergangsrelation. Aus den Zustandsübergangsrelationen der Komponenten eines Systems und dem aktuellen Systemzustand kann der nächste Systemzustand konstruktiv berechnet werden. Das Verhalten von hierarchischen Komponenten wird durch ein weiteres, untergeordnetes komponentenbasiertes System beschrieben. Dabei legen die Sichtbarkeitsregeln der hierarchischen Komponente fest, welche Instanzen des übergeordneten Systems in dem untergeordneten verwendet werden können und umgekehrt.

Für die Architekturbeschreibung derartiger komponentenbasierter Systeme haben wir sowohl textbasierte als auch UML-basierte, grafische Komponentenbeschreibungen und Systembe-

schreibungen entwickelt und formal fundiert. Komponentenbeschreibungen spezifizieren die zugesicherten Eigenschaften einer Komponente. Systembeschreibungen definieren die Komposition einer Menge von Komponenten zu einem komponentenbasierten System.

Darüber hinaus haben wir erweiterte Beschreibungstechniken für die explizite und präzise Spezifikation der Abhängigkeiten zwischen den einzelnen Komponenten eines komponentenbasierten Systems entworfen und formal fundiert. Erweiterte Komponentenbeschreibungen enthalten zusätzlich die angenommenen Eigenschaften einer Komponente. Erweiterte Systembeschreibungen enthalten für jede Komponente des Systems einen Annahme-/Zusicherungsvertrag. In einem Annahme-/Zusicherungsvertrag ist festgelegt, durch welche zugesicherten Eigenschaften von Komponenten die angenommenen Eigenschaften einer Komponente erfüllt werden.

Anhand eines durchgängigen Fallbeispiels, dem Entwurf und der Entwicklung eines verteilten Pausenplaneditors, wurden die erarbeiteten Konzepte illustriert und deren Anwendbarkeit demonstriert. Abschließend haben wir die grundlegende Konzeption einer durchgängigen und weitreichenden Werkzeugunterstützung des evolutionären Architekturentwurfs komponentenbasierter Systeme diskutiert. Der Entwurf und die Implementierung eines ersten Prototypen namens DesignIt wurden präsentiert. Eine Reihe von Erweiterungsmöglichkeiten für diesen Werkzeugprototyp haben wir vorgestellt.

Im einem nächsten Schritt sollte man diese aufgreifen und die Werkzeugunterstützung sukzessive ausbauen und verbessern. Gleichzeitig sind aussagekräftige Fallbeispiele mit der vorgestellten Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme umzusetzen. Das resultierende Feedback kann direkt in den weiteren Ausbau der Werkzeugunterstützung einfließen. Die entstandene Rückkopplung kann aber auch richtungsweisend für die Weiterentwicklung dieser Arbeit in einigen weiteren, in der Folge vorgestellten Bereichen sein.

Das methodische Rahmenwerk, das wir in dieser Arbeit vorgestellt haben, bietet genügend Potential für Erweiterungen in allen Richtungen. Die Methodik des evolutionären Architekturentwurfs komponentenbasierter Systeme kann schrittweise ausgebaut werden. Ausgehend von den existierenden Artefakten bestehender Ansätze und den Erfahrungen aus der Praxis kann so eine integrierte und umfassende Methodik der Softwareentwicklung erarbeitet werden. In der ersten Stufe wäre eine Integration der Anforderungsanalyse von besonderem Nutzen. Darüber hinaus könnte eine tiefergreifende Diskussion der spezifischen methodischen Vorgehensweisen bei der Transformation von Architekturmodellen, wie zum Beispiel beim Refactoring oder beim musterbasierten Entwurf, von Nutzen sein.

Die Verhaltensbeschreibungen in den vorgestellten Spezifikationstechniken basieren auf der Sprache OCL, die ein Bestandteil der UML ist. Alternative Beschreibungstechniken könnten hierfür angedacht werden, wie zum Beispiel Automatendiagramme, Sequenzdiagramme oder Aktivitätsdiagramme. Ein besonderer Schwerpunkt würde dabei in der Integration und Fundierung dieser unterschiedlichen Beschreibungstechniken liegen. Außerdem müsste die Werkzeugunterstützung für die neuen Beschreibungstechniken entsprechend erweitert werden.

Ein weiteres denkbares Einsatzgebiet der erarbeiteten Konzepte ist der Architekturentwurf mobiler komponentenbasierte Systeme. An einigen Stellen haben wir die notwendigen Erweiterungen an Systemmodell, Beschreibungstechniken und Werkzeugunterstützung bereits angedeutet, so dass sie relativ leicht angegangen werden könnten.

Und schließlich könnten die Konzepte und Techniken des evolutionären Architekturentwurfs so erweitert werden, dass nicht nur die Architektur des Systems spezifiziert, generiert und getestet wird, sondern auch die Benutzungsoberfläche. In der Konsequenz würde dies zu einer Methodik des evolutionären Systementwurfs führen. Ausgestattet mit geeigneten grafischen Spezifikationssprachen und einer durchgängigen Werkzeugunterstützung wäre dann die Entwicklung einer Programmiersprache der nächsten Generation in greifbarer Nähe. Da diese Sprache auf einem wesentlich höheren Niveau als die heute verfügbaren und verwendeten Sprachen wäre, könnte damit ein gewaltiges Potential für Produktivitäts- und Qualitätssteigerungen in der industriellen Softwareentwicklung erschlossen werden.

Literaturverzeichnis

[AC96] Martin Abadi, Luca Cardelli. A Theory Of Objects. Springer Verlag. 1996.

[AF01] AutoFocus. Die AutoFocus Homepage. Technische Universität München, Fakultät für Informatik, Lehrstuhl von Professor Dr. Manfred Broy, http://autofocus.informatik.tu-muenchen.de/. 2001.

[AG90] Caroline Ashworth, Mike Goodland. SSADM, A Practical Approach. McGraw-Hill. 1990.

[Alex79] Christopher Alexander. The Timeless Way of Building. Oxford niversity Press. 1979.

[Alle00] Paul Allen. Realizing eBusiness with Components. Addison Wesley Professional. 2000.

[Ambl98] Scott W. Ambler. Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press. 1998.

[Ambl99] Scott W. Ambler. More Process Patterns: Delivering Large-Scale Systems Using Object Technology. Cambridge University Press. 1999.

[ASU88a] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilerbau, Teil 1. Addison Wesley Publishing Company. 1988.

[ASU88b] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilerbau, Teil 2. Addison Wesley Publishing Company. 1988.

[Balz00] Helmut Balzert. Lehrbuch der Software-Technik, 2 Bände. Heidelberger Verlag Spektrum - Akademischer Verlag. 2000.

[BBR+00] Klaus Bergner, Manfred Broy, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Formal Model for Componentware. In Foundations of Component-Based Systems, Cambridge University Press. 2000.

[BBR+99] Klaus Bergner, Bernd Brügge, Andreas Rausch, Marc Sihling, Christoph Vilsmeier. Anbindung eines ressourcenbeschränkten, prozessorbasierten Systems an einen Mechanismus zur Signalisierung von Nachrichten. Patentanmeldung am Deutschen Patent- und Markenamt, Nummer 199 06 134.3. 1999.

[BCKB98] Len Bass, Paul Clements, Rick Kazman, Ken Bass. Software Architecture in Practice (Sei Series in Software Engineering). Addison-Wesley Publishing. 1998.

[BDD+92] Manfred Broy, Franz Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, Rainer Weber. The Design of Distributed Systems – an Introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München. 1992.

[Beck99] Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley. 1999.

[Berg97] Klaus Bergner. Spezifikation großer Objektgeflechte mit Komponentendiagrammen. Dissertation, Technische Universität München. 1997.

[BFR00] Thomas Bonfig, Rainer Frömming, Andreas Rausch. Goal - Eine Testinfrastuktur für unternehmensweite Anwendungen. In OBJEKTspektrum 4/2000. 2000.

[BG01] Kent Beck, Erich Gamma. JUnit, http://www.xprogramming.com/software.htm. 2001.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, Wolfgang Schwerin. Systems, Views and Models of UML. In The Unified Modeling Language,

Technical Aspects and Applications, Editors Martin Schader, Axel Korthaus. Physica Verlag, Heidelberg. 1998.

[BGR+99] Klaus Bergner, Radu Grosu, Andreas Rausch, Alexander Schmidt, Peter Scholz, Manfred Broy. Focusing on Mobility. In Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences (HICSS 32), IEEE Computer Society. 1999.

[BHH00] Leonor Barroca, Jon Hall, Patrick A. V. Hall. Software Architecture: Advances and Applications. Hall Springer Verlag. 2000.

[BHK+97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, Monika Schmidt. Using Extended Event Traces to Describe Communication in Software Architectures. In Proceedings of the Asia-Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong, IEEE Computer Society. 1997.

[BHRS97] Klaus Bergner, Franz Huber, Andreas Rausch, Marc Sihling. Component-Oriented Redesign of the CASE-Tool AutoFocus. Technical Report TUM-I9752, Technische Universität München. 1997.

[BJR98] Grady Booch, Ivar Jacobson, James Rumbaugh. The Unified Modeling Language User Guide. Addison Wesley Publishing Company. 1998.

[BKR98] Klaus Bergner, Karsten Kuhla, Andreas Rausch. Schnelle Schichten: Transparenter Zugriff auf ODBMS über CORBA. iX No. 11. 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons.. 1996.

[BMS96] Manfred Broy, Stephan Merz, Katharina Spies. The RPC-Memory Specification Problem: A Synopsis. In Formal Systems Specification – The RPC-Memory Specification Case Study, Lecture Notes in Computer Science 1169, Manfred Broy, Stefan Merz, Katherina Spies (eds.), Springer Verlag. 1996.

[Boeh86] Barry Boehm. A Spiral Model of Software Development and Enhancement. ACM Sigsoft Software Engineering Notes, Vol. 11, No. 4. 1986.

[Booc94] Grady Booch. Object-Oriented Analysis and Design With Applications. Addison Wesley Publishing Company. 1994.

[Bosh00] Jan Bosch. Design and Use of Software Architectures. Addison-Wesley Publishing. 2000.

[Bott01] Dick Botting. A Methods and Standards Page. http://www.csci.csusb.edu/dick/methods.html. 2001.

[Broo75] Frederik P. Brooks. The Mythical Man-Month. Addision Wesely. 1975.

[Brow00] Alan W. Brown. Large Scale Component Based Development. Prentice Hall. 2000.

[Broy96] Manfred Broy. A Functional Solution to the RPC-Memory Specification Problem. In Formal Systems Specification – The RPC-Memory Specification Case Study, Lecture Notes in Computer Science 1169, Manfred Broy, Stefan Merz, Katherina Spies (eds.), Springer Verlag. 1996.

[Broy97] Manfred Broy. Towards a Mathematical Concept of a Component and its Use. In Software-Concepts and Tools 18, pp. 137-148. 1997.

[BRS+99] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, Manfred Broy. A Formal Model for Componentware. In Proceedings des 9. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme, FBT `99,

Herbert Utz Verlag. 1999.

[BRS97] Klaus Bergner, Andreas Rausch, Marc Sihling. Using UML for Modeling a Distributed Java Application. Technical Report TUM-I9735, Technische Universität München. 1997.

[BRS98a] Klaus Bergner, Andreas Rausch, Marc Sihling. Componentware – The Big Picture. In Proceedings of the International Workshop on Component-Based Software Engineering. 1998.

[BRS98b] Klaus Bergner, Andreas Rausch, Marc Sihling. A Component-Oriented Architecture for the CASE-Tool AutoFocus. In Proceedings of the IASTED Conference on Software Engineering, ACTA Press. 1998.

[BRS99] Klaus Bergner, Andreas Rausch, Marc Sihling. AutoFocus – ein CASE-Tool für verteilte System. In Erfahrungen mit Java: Projekte aus Industrie und Hochschule, dpunkt Verlag. 1999.

[BRSV00] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. Putting the Parts Together – Concepts, Description Techniques, and Development Process for Componentware. Proceedings of the 33th Annual Hawaii International Conference on System Sciences, IEEE Computer Society. 2000.

[BRSV98a] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Componentware Development Methodology based on Process Patterns. Proceedings of the 5th Annual Conference on the Pattern Languages of Programs. 1998.

[BRSV98b] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Componentware Methodology based on Process Patterns. Technical Report TUM-I9823, Technische Universität München. 1998.

[BRSV98c] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Componentware Methodology Based on Process Patterns. In GI Workshop, Anwendung von objektorientierten Entwicklungsstrategien und deren Unterstützung durch Vorgehensmodelle. 1998.

[BRSV98d] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. Component-Oriented GUI Architectures: The Redesign of a Medical Patient Browser. Internal Report for Siemens. 1998.

[BRSV99a] Klaus Bergner, Andreas Rausch, Marc Sihling, Christoph Vilsmeier. CORBA and the Java Card – Connecting Small Devices to a Standard Event Service. In: SCI 99, Proceedings of the World Multiconference on Systems, Cybernetics, and Informatics., SCI `99. 1999.

[BRSV99b] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. Componentware – Methodology and Process. In Proceedings of the International Workshop on Component-Based Software Engineering. 1999.

[BRSV99c] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. An Integrated View On Componentware – Concepts, Description Techniques, and Development Process. In Proceedings of IASTED Conference on Software Engineering, ACTA Press. 1998.

[BS00] Peter Brössler, Johannes Siedersleben. Softwaretechnik: Praxiswissen für Softwareingenieure. Carl Hanser Verlag. 2000.

[BS01] Manfred Broy, Ketil Stølen. Specification and Development of Interactive Systems. Springer Verlag. 2001.

[BW97] Martin Büchi, Wolfgang Weck. A Plea for Grey-Box Components. Technical Report 122, Turku Centre for Computer Science, Turku, Finland. 1997.

[CD00] John Cheesman, John Daniels. UML Components: A Simple Process for Specifying Component-Based Software (The Component Software Series). Addison-Wesley Publishing. 2000.

[Chen76] P. P. S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. In ACM Transactions on Database Systems, 1(1), 1976.

[Cole93] Derek Coleman. Object-Oriented Development: The Fusion Method. Prentice Hall. 1993.

[CY90] Peter Coad, Edward Yourdon. Object-Oriented Analysis. Yourdon Press. 1990.

[CY91] Peter Coad, Edward Yourdon. Object-Oriented Design. Yourdon Press. 1991.

[DACH01a] oo D.A.CH. Gruppe von Forschern und Forscherinnen aus den Ländern Deutschland, Österreich und der Schweiz, die sich mit objektorientierten Fragestellungen auseinandersetzen. http://swt-www.informatik.uni-hamburg.de/research/projects/oodach/index.html. 2001.

[DACH01b] oo D.A.CH. Historie und Einführung in den Pausenplaner. http://set.gmd.de/~sylla/DACH/. 2001.

[DACH01c] oo D.A.CH. Alle Versionen der Aufgabenstellungen des Pausenplaners. http://set.gmd.de/~sylla/DACH/pap-aufgabe.html. 2001.

[DACH01d] oo D.A.CH. Erste Version der Pausenplaneraufgabe. http://set.gmd.de/~sylla/DACH/pap-v1.html. 2001.

[DACH01e] oo D.A.CH. Zweite Version der Pausenplaneraufgabe. http://set.gmd.de/~sylla/DACH/pap-v2.html. 2001.

[Deif01] Bernhard Deifel. Requirements Engineering komplexer Standardsoftware. Promotionsarbeit, Technische Universität München. 2001.

[DeMa79] Tom DeMarco. Structured Analysis and System Specification. New York, Yourdon Press. 1979.

[DeMa97] Tom DeMarco. The Deadline: A Novel About Project Management. Dorset House. 1997.

[Dene91] Ernst Denert. Software-Engineering. Berlin, Heidelberg, New York, Springer Verlag. 1991.

[Desi01] DesignIt. Die DesignIt Homepage. Technische Universität München, Fakultät für Informatik, Lehrstuhl von Prof. Dr. Manfred Broy, http://designit.informatik.tu-muenchen.de/. 2001.

[DKW00] David M. Dikel, David Kane, James R. Wilson. Software Architecture: Organizational Principles and Patterns. Prentice Hall. 2000.

[DL99] Tom DeMarco, Timothy Lister. Peopleware, Productive Projects and Teams, Second Edition Featuring Eight All-New Chapters. Dorset House Publishing Corporation. 1999.

[DSB99] Desmond D'Souza, Aamond Sane, Alan Birchenough. First-Class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns. In Proceedings of the UML ´99, Lecture Notes in Computer Science (pp. 265-278), 1723, Berlin, Heidelberg, New York, Springer Verlag. 1999.

[DSV99] Bernhard Deifel, Wolfgang Schwerin, Sascha Vogel . Work Products for Integrated Software Development. Technischer Bericht, TUM-I9921, Technische Universität München. 1999.

[DW98] Desmond Francis D'Souza, Alan Cameron Wills. Objects, Components, and Frameworks With UML: The Catalysis Approach. Addison Wesley Publishing Company. 1998.

[DW99] Wolfgang Dröschel, Manuela Wiemers. Das V- Modell 97. Oldenbourg. 1999.

[EB01] Alexander Egyed, Robert Balzer. Unfriendly COTS Integration – Instrumentation and Interfaces for Improved Plugability. White Paper from Teknowledge Corporation, http://www.teknowledge.com. 2001.

[EFLR99] Andy Evans, Robert France, Kevin Lano, Bernhard Rumpe. The UML as a Formal Modeling Notation. In The Unified Modeling Language – Workshop UML'98: Beyond the Notation. Springer Verlag Berlin, LNCS. 1999.

[EFT92] Heinz-Dieter Ebbinghaus, Jörg Flum, Wolfgang Thomas. Einführung in die mathematische Logik. BI-Wissenschaftsverlag. 1992.

[EK99] Andy Evans, Stuart Kent. Core Meta-Modeling Semantics of UML: The pUML Appraoch. In Proceedings of the UML ´99, Lecture Notes in Computer Science (pp. 265-278), 1723, Berlin, Heidelberg, New York, Springer Verlag. 1999.

[FO95] Brain Foote, William F. Opdyke. Lifecycle and Refactoring Patterns that Support Evolution and Reuse. In Pattern Languages of Programm Design, edited by J. Coplien and D. Schmidt, Addision-Wesley. 1995.

[FORS01] FORSOFT: Bayerische Forschungsverbund Software-Engineering (FORSOFT). http://www.forsoft.de/. 2001.

[Fowl97] Martin Fowler. Analysis Patterns, Reusable Object Models. Addision-Wesley. 1997.

[Fowl99] Martin Fowler. Refactoring, Improving the Design of Existing Code. Addison Wesley Publishing Company. 1999.

[FRS00] Rainer Frömming, Andreas Rausch, Hans Stadtherr. ShapeShifter – Automatisierte Migration von Datenbanken. Internal Report der 4Soft GmbH, München. 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley Publishing Company. 1995.

[GMP+01a] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, Wolfgang Schwerin. Towards a Living Software Development Process based on Process Patterns. In the Proceedings of the Eighth European Workshop on Software Process Technology. 2001.

[GMP+01b] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, Wolfgang Schwerin. Modular Process Patterns supporting an Evolutionary Software Development Process. In the Proceedings of the Third International Conference on Product Focused Software Process Improvement. 2001.

[GNR+00] Lindsay Groves, Ray Nickson, Greg Reeve, Steve Reeves, Mark Utting. A Survey of Software Development Practices in the New Zealand Software Industry. In Proceedings of the International Australian Software Engineering Conference 2000. 2000.

[Goos95] Gerhard Goos. Vorlesungen über Informatik, Band 1: Grundlagen und funktionales Programmieren. Springer Verlag. 1995.

[GR95] Adele Goldberg, Kenneth Rubin. Succeeding With Objects : Decision Frameworks for Project Management. Addison-Wesley. 1995.

[Grif98] Frank Griffel. Componenteware. dpunkt Verlag. 1998.

[Haas00] Stephan de Haas. E-Business-Anwendungen mit J2EE, 1. Teil: Überblick. OBJEKTspektrum 3/2000. 2000.

[HDF00] Heinrich Hussmann, Birgit Demuth, Frank Finger. Modular Architecture for a

Toolset Supporting OCL. In Proceedings of UML 2000 Conference, York, UK. October, 2000.

[HHG90] Richard Helm, Ian M. Holland, Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In Proceedings of OOP-SLA/ECOOP `90. 1990.

[HK99] Martin Hitz, Gerti Kappel. UML@Work: Von der Analyse zur Realisierung. dpunkt.verlag GmbH. 1999.

[HM01] Elliotte Rusty Harold, W. Scott Means. XML in a Nutshell: A Desktop Quick Reference (Nutshell Handbook). O'Reilly & Associates. 2001.

[HMR+98] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, Oscar Slotosch. Tool supported Specification and Simulation of Distributed Systems. Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society. 1998.

[HNS99] Christine Hofmeister, Robert Nord, Dilip Soni. Applied Software Architecture. Addison Wesley Publishing Company. 1999.

[Hoar85] Charles A. Hoare. Communicating Sequential Processes. Prentice Hall. 1985.

[Holl93] Ian M. Holland. The Design and Representation of Object-Oriented Components. PhD Thesis, Northeastern University. 1993.

[HP01] Lucien Hoogendoorn, Sylvia Pohlmann. Konzeption und Realisierung einer verteilten Componentware-Umgebung basierend auf J2EE und CORBA und Konzeption und Realisierung eines generischen Code-Generators für UML-basierte CASE-Tools. Systementwicklungsprojekt, Institut für Informatik, Technische Universität München. 2001.

[HP98] David Harel, Michal Politi. Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill. 1998.

[HR00] David Harel, Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Technical Report, The Weizmann Institute of Science, Rehovot, Israel, MCS00-16. 2000.

[HS99] Peter Herzum, Oliver Sims. Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons. 1999.

[HU88] John E. Hopcroft, Jeffrey D. Ullman. Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley. 1988.

[Huck01] Thomas Huckle. Kleine BUGs, große GAUs . http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html. 2001.

[IBM97] IBM Object-Oriented Technology Center. Developing object-oriented software: An experience-based approach. Prentice Hall. 1997.

[IEEE99] IEEE Software Engineering Standards, 1999 Edition Volume Two: Processes. IEEE. 1999.

[Inte01] Intersolv. Die Intersolv Homepage. http://www.intersolv.com. 2001

[Isem00] David Iseminger. COM+ Developer's Reference Library (Windows Programming Reference Series). Microsoft Press. 2000.

[ITU94] International Telecommunication Union. Message Sequence Charts. ITU-T Recommendation Z.120. Geneva. 1994.

[IW99] InformationWeek 500 Europa. Der Club der Innovatoren (InformationWeek Studie). InformationWeek, Januar 1999.

[Jaco92] Ivar Jacobson. Object-Oriented Software Engineering: A Use Case Driven Ap-

proach. Addison Wesley Publishing Company. 1992.
[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh. Unified Software Development Process. Addison Wesley Publishing Company. 1999.
[Jone86] Cliff B. Jones. Systematic Software Development Using VDM. Prentice Hall. 1986.
[JRLL00] Mehdi Jazayeri, Alexander Ran, Frank Van Der Linden, Philip Van Der Linden. Software Architecture for Product Families: Principles and Practice. Addison-Wesley Publishing. 2000.
[Keie99a] Frank Keienburg. Schema Evolution for a Distributed UML Software Engineering Workbench based on CORBA, XML and Java. Systementwicklungsprojekt, Institut für Informatik, Technische Universität München. 1999.
[Keie99b] Frank Keienburg. Modellierung und Organisation technischer Daten am Beispiel einer Turbinenscheibe. Diplomarbeit an Institut für Informatik, Technische Universität München. 1999.
[Kim91] Hyoung-Joo Kim. Algorithmic and Computational Aspects of OODB Schema Design. In Object-Oriented Databases with Applications to CASE, Prentice Hall. 1991.
[Kivl00] Daire Kivlehan. DesignIt: Code Generator based on XML and Code Templates. Dissertation of Master of Science in Computer Science and Applications in the College of Engineering, The Queen's University of Belfast. 2000.
[KN93] Benjamin J. Keller, Richard E. Nance. Abstraction Refinement: A Model of Software Evolution. Journal of Software Maintenance: Research and Practice, Vol. 5 123-145. 1993.
[KR01] Frank Keienburg, Andreas Rausch. Using XML/XMI for Tool Supported Evolution of UML Models. In the Proceedings of the Thirty-Four Annual Hawaii International Conference on System Sciences, IEEE Computer Society. 2001.
[Kroh97] Petra Kroha. Softwaretechnologie. Prentice Hall. 1997
[KRSW01] Cornel Klein, Andreas Rausch, Marc Sihling, Zhaojun Wen. Extension of the Unified Modeling Language for Mobile Agents. In: Unified Modeling Language: Systems Analysis, Design and Development Issues, edited by Keng Siau and Terry Halpin. Idea Group Publishing Book. 2001.
[Kruc00] Philippe Kruchten. The Rational Unified Process: An Introduction, Second Edition. Addison Wesley Publishing Company. 2000.
[Krüg00] Ingolf Krüger. Distributed System Design with Message Sequence Charts. Dissertation, Technische Universität München. 2000.
[Lamp89] Leslie Lamport. A simple approach to specifying concurrent systems. Communications of the ACM. ACM. 1989.
[Lave00] Karen Lavery. DesignIt: Componentware Runtime Environment. Dissertation of Master of Science in Computer Science and Applications in the College of Engineering, The Queen's University of Belfast. 2000.
[LBR99] Gary T. Leavens, Albert L. Baker, Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06d, Iowa State University, Department of Computer Science. 1999.
[LS00] Gary T. Leavens, Murali Sitaraman. Foundations of Component-Based Systems. Cambridge University Press. 2000.
[MEH01] Mark Maier, David Emery, Rich Hilliard. Introducing IEEE 1471. In IEEE Computer, colume 34, number 4. 2001.

[Meye88] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall. 1988.
[Meye92] Bertrand Meyer. Applying ´design by contract`. In Computer (IEEE), Vol. 25, No. 10. Oktober 1992.
[Meye97] Bertrand Meyer. Object-Oriented Software Construction (Second Edition). Prentice Hall. 1997.
[MLS98] Tom Mens, Carine Lucas, Patrick Steyart. Supporting Reuse and Evolution of UML Models. In Proceedings of the First Interational Workshop on UML, UML `98 – Beyond the Notation. 1998.
[MO92] James Martin, James J. Odell. Object-Oriented Analysis and Design. Prentice Hall. 1992.
[MSL98] Tom Mens, Patrick Steyart, Carine Lucas. Giving Precise Semantics to Reuse and Evolution in UML. In Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques, Technical Report TUM-I9803, Technische Universität München. 1998.
[Müll99] Gunter Müller-Ettrich. Objektorientiertet Prozeßmodelle: UML einsetzen mit OOTC, V-Modell, Objectory. Addision Wesley Longman Verlag GmbH. 1999.
[OFM+94] Anders Olsen, O. Færgemand, B. Møller-Pedersen, Rick Reed and J. R. W. Smith. Systems Engineering Using SDL-92. North-Holland. 1994.
[OHJ+99] Bernd Oestereich, Peter Hruschka, Nicolai Josuttis, Hartmut Kocher, Hartmut Krasemann, Markus Reinhold. Erfolgreich mit Objektorientierung. Oldenbourg. 1999.
[OMG00a] Object Management Group (OMG). OMG Unified Modeling Language Specification (Version 1.3). http://www.omg.org, document number: formal/2000-03-01. 2000.
[OMG00b] Object Management Group (OMG). Meta Object Facility (MOF) Specification (Version 1.3). http://www.omg.org, document number: formal/2000-04-03. 2000.
[OMG00c] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification (Version 1.1). http://www.omg.org, document number: formal/2000-11-02. 2000.
[OMG01a] Object Management Group (OMG). Die Object Management Group Homepage. http://www.omg.org. 2001.
[OMG01b] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification (Version 2.4). http://www.omg.org, document number: formal/2001-02-33. 2001.
[OMG97] Object Management Group (OMG). OMG Unified Modeling Language Specification (Version 1.1). http://www.omg.org, document number: formal/1997-08-05. 1997.
[OMG99] Object Management Group (OMG). CORBA Components. http://www.omg.org, document number: orbos/99-07-01, 02 and 03 (joint submission to OMG's Components RFP). 1999.
[OOSE01] oose.de Dienstleistungen für innovative Informatik GmbH. Object Engineering Process (OEP). http://www.oose.de/oep/. 2001.
[Opdy92] William F. Opdyke. Refactoring Object-Oriented Frameworks. Ph.D. diss., University of Illinois at Urbana-Champaign. 1992.
[Ould95] Martyn A. Ould. Business Processes: Modeling and Analysis for Reengineering and Improvement. John Wiley & Son. 1995.

[Parn94] David Lorge Parnas. Software aging. In Proceedings of 16th International Conference on Software Engineering, pages 279-287, Sorrento, Italy, 16-21 May, IEEE. 1994.

[PB96] Gustav Pomberger, Günther Blaschek. Software-Engineering: Prototyping und objektorientierte Software-Entwicklung. Carl Hanser Verlag. 1996.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model for Software, Version 1.1. Software Engineering Institute, CMU/SEI-93-TR-24, DTIC Number ADA263403. 1993.

[Poch75] Marc J. Pochkind. The source code control system. IEEE Transactions on Software Engineering. SE-1(4), pp. 364-470. 1975.

[Rass98] Thomas Rassmann. Ein Vorgehensmodell für das Web-Site Engineering und Konzepte für das Konfigurationsmanagement bei der Entwicklung und Verwaltung von Web-Sites. Diplomarbeit, Institut für Informatik, Technische Universität München. 1998.

[Rati01] Rational. Die Rational Homepage. http://www.rational.com. 2001.

[Raus00a] Andreas Rausch. A Proposal for Software Evolution in Componentware. In Proceedings of the 4th European Conference on Software Maintenance and Reengineering, IEEE Computer Society. 2000.

[Raus00b] Andreas Rausch. Software Evolution in Componentware – A Practical Approach. In Proceedings of the 2000 Australian Software Engineering Conference, IEEE Computer Society. 2000.

[Raus00c] Andreas Rausch. Software Evolution in Componentware Using Requirements/Assurances Contracts. In Proceedings of the 22th International Conference on Software Engineering. 2000.

[Raus01a] Andreas Rausch. A Proposal for a Code Generator based on XML and Code Templates. In the Proceedings of the Workshop of Generative Techniques for Product Lines, 23rd International Conference on Software Engineering. 2001.

[Raus01b] Andreas Rausch. Towards a Software Architecture Specification Language based on UML and OCL. In the Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering. 2001.

[Raus99] Andreas Rausch. Executive Summary: Software Evolution in Componentware – A Practical Approach. In the Proceedings of the International Workshop on Software Change and Evolution. 2000.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen. Object-Oriented Modeling and Design. Addison Wesley Publishing Company. 1991.

[RJB98] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Addison Wesley Publishing Company. 1998.

[RKB95] Bernhard Rumpe, Cornel Klein, Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender System – Syslab Systemmodell. Technical Report TUM-I9510, Technische Universität München. 1995.

[RKP00] Bert Röge, Josef Krumbahcer, Ulrich Pfaffenberger. Wandel in den Köpfen. w-info, Wirtschaftsmagazin für München und Oberbayern 19/2000. 2000.

[Royc70] Winston W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In WESCON Technical Papers, Western Electronic Show and Convention, Los Angeles, Aug. 25-28, number 14. 1970.

Reprinted in Proceedings of the Ninth International Conference on Software Engineering, Pittsburgh, PA, USA, ACM Press, 1989, pp.~328--338.

[Rump96] Bernhard Rumpe. Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Dissertation, Technische Universität München. 1996.

[Sche00] Manuela Scherer. Towards a Sound Foundation of the UML Constraint Language OCL in Isabelle/HOL. Technische Universität München, Diplomarbeit. 2000.

[SG96] Mary Shaw, David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall. 1996.

[SGW94] Bran Selic, Garth Gullekson, Paul T. Ward. Real-Time Object-Oriented Modeling. John Wiley & Sons. 1994.

[SM89] Sally Shlaer, Stephen J. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data. Yourdon Press. 1989.

[Somm00] Ian Sommerville. Software Engineering. Addison-Wesley. 2000.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons. 2000.

[Støl96] Ketil Stølen. Using Relations on Streams to Solve the RPC-Memory Specification Problem. In Formal Systems Specification – The RPC-Memory Specification Case Study, Lecture Notes in Computer Science 1169, Manfred Broy, Stefan Merz, Katherina Spies (eds.), Springer Verlag. 1996.

[SUN01] Sun Microsystems (SUN). Java 2 Platform Enterprise Edition Specification (Version 1.3). Sun Microsystems (SUN), http://sun.java.com/j2ee/docs.html. 2001.

[Szyp97] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison Wesley Publishing Company. 1997.

[Tich85] Walter F. Tichy. RCS – a system for version control. Software – Practice and Experience, 15(7), pp. 637-654. 1985.

[Toge01] Togethersoft. Die Togthersoft Homepage. http://www.togethersoft.com. 2001.

[TS00] Amir Tomer, Stephen R. Schach. The Evolution Tree: A Maintenance-Oriented Software Development Model. In Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000), Zurich, Switzerland, February/March 2000, pp. 209-214. 2000.

[Wall91] Emmanuel Waller. Schema Updates and Consistency. In DOOD'91 Proceedings, Springer Verlag. 1991.

[Wirt86] Niklaus Wirth. Compilerbau. Teubner Verlag. 1986.

[WK98] Jos B. Warmer, Anneke G. Kleppe. The Object Constraint Language: Precise Modeling With UML. Addison Wesley Publishing Company. 1998.

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. Designing Object-Oriented Software. Prentice Hall. 1990.

[XMLD01] XML Diff and Merge Tool. IBM Alphaworks, http://alphaworks.ibm.com/aw.nsf/frame?ReadForm&/aw.nsf/techmain/CB2EF938D7532F338825671B0068244F. 2001.

[YC79] Edward Yourdon, Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall. 1978.

[Your89] Edward Yourdon. Modern Structured Analysis. Prentice-Hall. 1989.

[ZL99] Ron Zahavi, David S. Linthicum. Enterprise Application Integration with

CORBA Component and Web-Based Solutions. John Wiley & Sons. 1999.

Abbildungsverzeichnis

Abbildung 2.1: Grundbausteine der methodischen Softwareentwicklung 13
Abbildung 2.2: Grundlegende Konzepte in Vorgehensmodellen 14
Abbildung 2.3: Zusammenhang zwischen Prozessmodellen und Beschreibungstechniken 16
Abbildung 2.4: Softwarearchitektur und Laufzeitumgebung 17
Abbildung 2.5: Die Projektsicht auf den prozessmusterbasierten Ansatz 21
Abbildung 2.6: Die Methodiksicht auf den prozessmusterbasierten Ansatz 23
Abbildung 2.7: Die verschiedenen Ebenen der Methoden und Vorgehensmodelle 24
Abbildung 2.8: Prozessmusterbasiertes Metamodell für Entwicklungsmethoden 25
Abbildung 2.9: Die drei Dimensionen im Lebenszyklus der Modelle eines Systems 27
Abbildung 2.10: Die Verschiedenen Sichten eines Architekturmodells 29
Abbildung 2.11: UML Klassendiagramm des Architekturmodells im Projektschrank 32
Abbildung 2.12: Vorgehen beim evolutionären Architekturentwurf 41
Abbildung 3.1: Evolution als Vergröberung und Verfeinerung 52
Abbildung 3.2: Prädikatenbasierte Semantik und Annahme-/Zusicherungsverträge 57
Abbildung 4.1: Systemvision des Pausenplaners (aus [DACH01d]) 69
Abbildung 4.2: UML Anwendungsfalldiagramm des Pausenplaners aus [BRS97] 70
Abbildung 4.3: UML Aktivitätsdiagramm des Pausenplaners aus [BRS97] 71
Abbildung 4.4: UML Kollaborationsdiagramm des Observer Patterns 73
Abbildung 4.5: Komponenten des Pausenplaners 74
Abbildung 5.1: Eine Folge von Schnappschüssen des Pausenplaners 79
Abbildung 5.2: Struktur komponentenbasierter hierarchischer Systeme 91
Abbildung 5.3: Sichtbarkeitsregeln und Abbildungen der Systemzustände 93
Abbildung 5.4: Erweitertes Zeitmodell und Verhalten hierarchischer Systeme 96
Abbildung 6.1: Textbasierte Spezifikation der Komponente `StaffOrganizer` 103
Abbildung 6.2: Textbasierte Spezifikation der Komponente `BreakPlanOrganizer` 104
Abbildung 6.3: Textbasierte Spezifikation des Systems `BreakPlanner` 106
Abbildung 6.4: UML-basiertes Komponentendiagramm 128
Abbildung 6.5: Komponentendiagramm mit integrierter Verhaltensbeschreibung 129
Abbildung 6.6: UML-basiertes Systemdiagramm 130
Abbildung 6.7: Sequenzdiagramm eines komponentenbasierten Systems 132
Abbildung 7.1: Komponentendiagramm des Pausenplaners mit Pausenstatistik I 136
Abbildung 7.2: Komponentendiagramm des Pausenplaners mit Pausenstatistik II 137
Abbildung 7.3: Erweiterte textbasierte Spezifikation der Komponente `StaffOrganizer` 140
Abbildung 7.4: Erweiterte textbasierte Spezifikation der Komponente `BreakPlanOrganizer` 141
Abbildung 7.5: Erweiterte textbasierte Spezifikation des Systems `BreakPlanner` 142
Abbildung 7.6: Ausgangsaxiom des Beweises 143
Abbildung 7.7: Axiom des Beweises mit ersetzten Spezifikationsbezeichnern 143
Abbildung 7.8: Beweisziel des Beweises 143
Abbildung 7.9: Beweisziel des Beweises mit ersetzten Spezifikationsbezeichnern 144
Abbildung 7.10: Erweitertes UML-basiertes Komponentendiagramm 151
Abbildung 7.11: Erweitertes UML-basiertes Systemdiagramm 152
Abbildung 8.1: Werkzeugunterstützung für den evolutionären Architekturentwurf 156
Abbildung 8.2: Architekturspezifikation des Pausenplaners in Together 159
Abbildung 8.3: DesignIt Prüfmechanismen für Architekturspezifikationen 160

Abbildung 8.4: Evolutionsinspektor für Architekturspezifikation mit XMLDiff 161
Abbildung 8.5: DesignIt Integrationsinspektor für Architekturspezifikation 162
Abbildung 8.6: DesignIt Generatoren für Architekturspezifikationen 163
Abbildung 8.7: DesignIt Debug-Ausführung der Pausenplanerspezifikation 165
Abbildung 8.8: UML Klassendiagramm der Ausführungsumgebung von DesignIt 166

Definitionsverzeichnis

Definition 3.1: Systemmodell 48
Definition 3.2: Syntaktisches Modell 48
Definition 3.3: Systemmodellbasierte formale Semantik von Dokumentenmengen 48
Definition 3.4: Evolution von Dokumentenmengen 49
Definition 3.5: Instanzen und Spezifikationsbezeichner eines semantischen Modells 55
Definition 3.6: Prädikatenbasierte formale Semantik von Spezifikationsbezeichnern 56
Definition 3.7: Eigenschaften von Spezifikationsbezeichnern 56
Definition 3.8: Annahme-/Zusicherungsvertrag (Requirement Assurance Contract) 56
Definition 3.9: Spezifikation eines komponentenbasierten Softwaresystems 58
Definition 3.10: Eigenschaften von Spezifikationen komponentenbasierter Systeme 58
Definition 3.11: Konsistenz und Vollständigkeit von Spezifikation 59
Definition 3.12: Inkonsistenzen und Unvollständigkeiten in Spezifikationen 60
Definition 3.13: Stabile Eigenschaften in Spezifikationen und Spezifikationsbezeichnern 61
Definition 3.14: Neue Eigenschaften in Spezifikationen und Spezifikationsbezeichnern 61
Definition 3.15: Entfernte Eigenschaften in Spezifikationen und Spezifikationsbezeichnern 62
Definition 3.16: Verfeinerung 63
Definition 3.17: Vergröberung 63
Definition 3.18: Strikte Evolution 63
Definition 5.1: Instanzen in komponentenbasierten Systemen 80
Definition 5.2: Struktur komponentenbasierter Systeme 81
Definition 5.3: Datenzustand komponentenbasierter Systeme 81
Definition 5.4: Kommunikation in komponentenbasierten Systemen 82
Definition 5.5: Schnappschüsse eines komponentenbasierten Systems 83
Definition 5.6: Gezeitete Ströme 84
Definition 5.7: Verhalten eines komponentenbasierter Systeme 85
Definition 5.8: Erzeugen und Löschen von Instanzen 85
Definition 5.9: Zuordnung von Komponenten, Schnittstellen und Attributen 86
Definition 5.10: Unabhängigkeit der Systeme 86
Definition 5.11: Zustandsübergangsrelationen von Komponenten eines Systems 86
Definition 5.12: Projektion auf Mengen 88
Definition 5.13: Zustandsübergangsrelation der aktiven Komponenten in einem System 88
Definition 5.14: Menge der schaltbaren Zustandsübergänge in einem System 88
Definition 5.15: Operator für das Ersetzen von Tupeln in Mengen 88
Definition 5.16: Einfaches Ausführungsmodell komponentenbasierter Systeme 89
Definition 5.17: Struktur komponentenbasierter hierarchischer Systeme 90
Definition 5.18: Sichtbarkeit in komponentenbasierten hierarchischen Systemen 92
Definition 5.19: Operator für die Durchsetzung der Sichtbarkeitsregeln 94
Definition 5.20: Abbildung des Systemzustandes vom Supersystem in das Subsystem 94
Definition 5.21: Abbildung des Systemzustandes vom Subsystem in das Supersystem 95
Definition 5.22: Ausführungsmodell komponentenbasierter hierarchischer Systeme 97
Definition 6.1: Grammatik und Sprache von Architekturspezifikationen 116
Definition 6.2: Spezifikationsbezeichner einer Architekturspezifikation 117
Definition 6.3: Instanzen eines Systems 117
Definition 6.4: Von Instanzen zu den Spezifikationsbezeichnern 118

Definition 6.5: Struktur und Anzahl von Schnittstellen an Komponenten 118
Definition 6.6: Struktur und Anzahl von Attributen an Schnittstellen 119
Definition 6.7: Struktur und Anzahl von Verbindungen zwischen Schnittstellen 120
Definition 6.8: Versenden von Nachrichten an Schnittstellen 120
Definition 6.9: Interpretation von Attributwertberechnungen 121
Definition 6.10: Attributwertberechnung 122
Definition 6.11: Interpretation von Bedingungen in Invarianten 122
Definition 6.12: Bedingungsauswertung von Invarianten 123
Definition 6.13: Interpretation von Verhaltensbeschreibungen 123
Definition 6.14: Ausführung von Invarianten und Nachrichten 124
Definition 6.15: Komponenteninstanzen in Systemen 125
Definition 6.16: Startzustand eines Systems 125
Definition 6.17: Subsysteme eines Supersystems 125
Definition 6.18: Instanzierung des Wurzelsystems 126
Definition 7.1: Grammatik und Sprache von erweiterten Architekturspezifikationen 147
Definition 7.2: Zusammenhang zwischen erweiterten und einfachen Spezifikationen 147
Definition 7.3: Struktur und Anzahl von Schnittstellen an Komponenten 148
Definition 7.4: Annahme-/Zusicherungsverträge von Komponenten 149
Definition 7.5: Gültigkeit von Annahme-/Zusicherungsverträgen 150

Weitere Informationen im Internet

Grammatiken der Syntax der erarbeiteten Spezifikationstechniken als XML DTD:

- Einfache Komponentenspezifikationen entsprechend Kapitel 6:
  http://designit.informatik.tu-muenchen.de/dtd/ComponentSpecification.dtd
- Einfache Systemspezifikationen entsprechend Kapitel 6:
  http://designit.informatik.tu-muenchen.de/dtd/SystemSpecification.dtd
- Erweiterte Komponentenspezifikationen entsprechend Kapitel 7:
  http://designit.informatik.tu-muenchen.de/dtd/ExtendedComponentSpecification.dtd
- Erweiterte Systemspezifikationen entsprechend Kapitel 7:
  http://designit.informatik.tu-muenchen.de/dtd/ExtendedSystemSpecification.dtd

Beispiele für die Anwendung der vorgestellten Spezifikationstechniken:

- Intro Beispiel:
  http://designit.informatik.tu-muenchen.de/sample/Intro.zip
- BreakPlanner Beispiel:
  http://designit.informatik.tu-muenchen.de/sample/BreakPlanner.zip

Werkzeugprototyp DesignIt:

- DesignIt:
  http://designit.informatik.tu-muenchen.de/tool/DesignIt.zip

Weitere aktualisierte Informationen zu dieser Arbeit und zu DesignIt:

- DesignIt Homepage:
  http://designit.informatik.tu-muenchen.de/